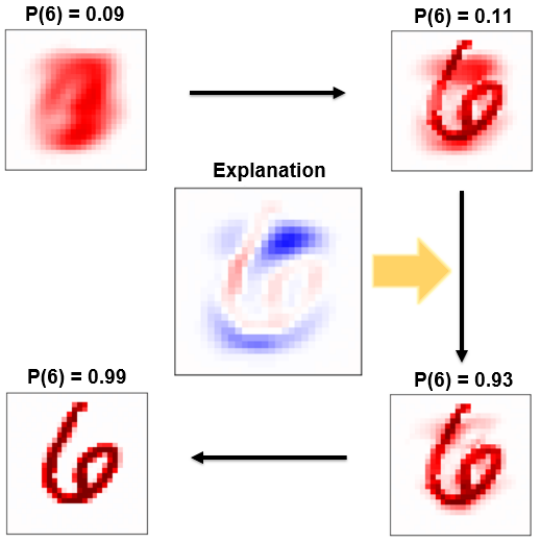
Understanding Deep Learning Decisions: the Explanatory Vector Decomposition (EVD) method

Master Thesis by Winfried van den Dool
Supervisors:
Dr. Sjoerd Dirksen (Utrecht University)
Pieter-Jan van Kessel (PwC)
Second Reader:
Dr. Tristan van Leeuwen (Utrecht University)

February 4, 2020



Abstract

Although successful in terms of prediction accuracy, the Artificial Neural Network has a notable drawback, namely the lack of explainability of its outcomes.

We propose a mathematical definition for the concept of an explanation in the context of understanding deep learning decisions. We put forward the Explanatory Vector Decomposition (EVD) method for computing such explanations, based on optimizing *explanation strength*. This is defined as the difference in model output probability caused by a movement in input space, divided by the vector length of this movement.

We also propose a technique for quantitatively comparing existing explainability methods that compute feature importance, using this newly found definition of explanation strength.

Implementation of this technique on LIME and RDE points to a higher average explanation strength achieved by the latter method, while the EVD method outperforms both according to this measure.

Contents

1 Introduction

Ever since the development of the first modern computers people have fantasised about Artificial Intelligence. With the increasing availability of large amounts of data, as well as more efficient methods to store and process it, Machine Learning became a major research field that brought previously unreachable AI ideas to life. It tackles one of the fundamental goals of AI, namely the ability to autonomously learn from outside environments.
More specifically, machine learning is defined as the study of algorithms that automatically learn and improve from experience without being explicitly programmed. In practice this study focuses on the development of computer programs that use data to independently learn prediction or classification functions. Although the first machine learning algorithms were little more than statistical methods, developments in computing power contributed to increasingly creative model ideas. Encouraged by the original ambition of using computers to mimic intelligence, a new model has been developed inspired by the biological brain: the Artificial Neural Network (ANN) [FC54].
The ANN is based on a layered collection of connected units or nodes called artificial neurons, which loosely model the neurons of the brain. When a large number of layers is used, the technique is also referred to as *deep learning*[GBC16]. While originally intended to solve problems in the same way as the human brain would, over time attention moved to performing specific tasks. The largely empirical research on performance optimization of the (artificial) neural network on these tasks lead to deviations from biology.

Over the last decades, neural networks have steadily achieved higher prediction accuracies, making them attractive in an increasingly broad field of applications. As a result, the interest in deep learning has spread from academic and distinct technological environments to the broader business sector and society as a whole.[Sch14] This has lead to new requirements and preferences for using machine learning models, exposing an important obstacle: although the neural network is very successful as a prediction model, it does not give any explanation as to *why* a certain label is outputted. In fact, the architecture of large neural networks can be so complex that understanding the model's behavior is practically impossible.
The "black box" nature of machine learning models limits their applications. This has incited a new wave of research, focusing on *explaining the outputs* of these models - a topic that largely lagged behind the efforts to maximize prediction accuracy. Consequently many of the latest developments in the ANN research field are related to so-called *explainability methods*, that aim to make the models and their decision-making process understandable.[AB18]

The literature on machine learning explainability as well as the practical experience with explaining model outputs shows an important caveat: a formal mathematical definition for an "explanation" in this context is missing. The

many different existing explainability methods in effect do not directly return explanations, but rather give diverse proxy measures related to feature importance or input sensitivity.
Although these methods are often successful in their prime objective of "opening the black box" by delivering greater understanding of model decision-making, we note that comparing the explanatory effectiveness among methods is highly subjective due to the different approaches. Also, the outputted proxy measures may score high on the goals set by the explanatory methods themselves, but may not be useful at all when it comes to forming an actual explanation.
A notable example of this is the LIME method [RSG16] applied to neural network classification of hand-written digit images. We show that in some cases the method, designed to return a form of feature (pixel) importance, simply highlights the pixels that together form the digit. While these pixels are indeed important (removing them would remove the entire digit) they do not *explain* why a digit image is labeled the way it is.

In this thesis we introduce a new method, the *Explanatory Vector Decomposition* method, for generating explanations of neural network outputs. The purpose of the newly developed technique is two-fold. Firstly, whereas most explainability methods aim for an interpretable outcome directly, we separate the concepts of explainability and interpretability and define the notion of "explanation strength". This measure enables us to compare other explainability methods, by first translating feature importance maps to explanations and then comparing average explanation strength.
Secondly, by optimizing explanation strength we arrive at a new explainability method, that can be regularized for interpretability to become a practical method by itself.
Hopefully, this thesis will open further avenues of research into this important and interesting topic.

1.1 Structure of this work

In this thesis we first thoroughly explain the mathematical background of deep learning (Chapter 2), before investigating the more recent research on explainability methods (Chapter 3). Three topics from this latter field, namely the Bayesian Neural Network (Section 3.2) as well as the LIME and RDE explainability methods (Sections 3.3 and 3.4) receive extra attention: We implement a Bayesian neural network image classifier on the MNIST dataset of hand-written digits and apply the LIME and RDE methods to retrieve explanations of the model's outputs.
In Chapter 4 we implement the EVD method on a Bayesian neural network trained on the MNIST data-set, so that we can compare the three explainability methods (EVD, LIME and RDE) in Chapter 5. We find that EVD method achieves the highest explanation strength, followed by RDE and finally LIME, supporting our initial observations of the outputs of these methods.

2 Artificial Neural Networks

In this section we give general background information on (artificial) neural networks, introducing the notations that will be used throughout the rest of this text. Although much more theory exists on this topic, we only present the what is required for understanding the subsequent chapters, and only mention the specific techniques that are used in our implementation.
In the first subsection we treat neural networks only globally, in the more general context of supervised machine learning. Most of what is discussed in this section also applies to other machine learning techniques. When this is the case we nonetheless treat these topics from a neural network perspective: the purpose is to give a light top-down introduction into neural networks, only gradually diving deeper into the mathematical structure and techniques.

2.1 Supervised Machine Learning

Let $d, N \in \mathbb{N}$, $N \geq 2$. Let $X \subseteq \mathbb{R}^d$ be a given feature space, and $Y \subseteq \mathbb{R}^N$ a label space. Under the assumption that there is some, possibly probabilistic, unknown model $g : X \to Y$, the general goal of machine learning algorithms is to produce a function f that mimics g, i.e., it takes elements $x \in X$ and outputs the correct[1] label $y \in Y$ given by the unknown model. For this task a data set $S \subset X \times Y$ of correctly labeled data points is collected, in the sense that $y = g(x)$ for $(x, y) \in S$, or y is the result of a random draw of $g(x)$ if g is probabilistic. We write S_X when referring to the set of data points without their labels. Both subsets of X and subsets of $X \times Y$ are sometimes referred to as data, depending on whether data points include labels or not.
Of course, fitting a function f to a data set S for prediction purposes is a very general concept. Distinguishing machine learning from ordinary statistical or econometric methods can be complicated due to a large gray area in this field of research. However, it might help to observe that in machine learning, as opposed to other methods, the function that is to be fitted to the data is so complicated that the optimal parameters cannot be found analytically. An iterative training procedure is used, slightly improving parameters of the function step by step. The fact that these function parameters are not determined or computed by the programmer beforehand, is the reason why this technique is called *machine learning*, as the algorithm itself *learns* from data that the programmer supplies. The popularity of machine learning, compared to other methods of fitting functions, then largely arises from the freedom of functional forms that can be chosen, as it is no longer necessary to be able to find the parameters analytically or by some simple statistical method.[2]

[1] In case of a probabilistic model g, the notion of a "correct" label can in practice be interpreted in different ways, e.g. the correct label can be interpreted as being reasonably close to either $\mathbb{E}[g(x)|x]$ or $\arg\max\{\mathbb{P}(g(x) = y|x) : y \in Y\}$.

[2] Simple models, on data with only a few features, can often easily be fitted using a small number of parameters. However, if we want to formulate a function labeling image data as either cats or dogs, for example, where images are represented by the different pixel intensities, a much more complex functional form is required.

We denote the function that is to be fitted to the data by $f = f(x; p)$, where $x \in X$ is a data point, and $p \in \mathcal{P}$ represent the function parameters, with the parameter space $\mathcal{P}$ depending on the specific form of f. We describe this parameter space in detail in Section 2.2.1. We occasionally omit the dependency on x or p depending on whether we want to explicitly show f is a function of x or p. Based on the information obtained from observing S, the machine learning algorithm now tries to find the best suitable function parameters p, i.e., those leading to the highest percentage of accurately predicted labels by $f(p)$ on (some subset of) S_X.
The approach described above is called *Supervised* machine learning, because the data set contains elements $x \in X$ as well as their corresponding labels $y \in Y$. Sometimes there are no such labels available, and the goal is not to train a labeling function (which would be impossible) but rather to train a function that recognizes some pattern or different categories or clusters in the data. Different elements can be found to belong to different categories based on some distinct features in X, without knowing what such a category really represents (there is no label set Y). This type of learning is called *unsupervised* machine learning. We will not treat such learning here, and always assume our training data set S also contains proper labels.

2.1.1 Cost functions and Stochastic Gradient Descent

Simple trial and error to find good parameters p, i.e., aiming for high accuracy on (some subset of) the data set S, is inefficient when the dimensionality of the input space is large. Instead, for neural networks, a technique called *stochastic gradient descent* is used. For this technique a proxy measure of the parameter quality is required: given a data set S, the performance of the algorithm on this set is monitored by computing the *cost function* $C(S, f(p))$, or $C(S, p)$. This function has a general behavior of returning a larger value when the labels outputted by $f(p)$ on S_X deviates more (frequently) from the correct labels observed in S. It is additive in S in the sense that $C(S, p) = \sum_{s \in S} C(s, p)$. We slightly abuse notation and write $C(s, p)$ rather than $C(\{s\}, p)$ when it is clear one datum from S is taken, rather than a subset.
A well-known example of a cost function is the *least squares* cost function,

$$C(S, p) = \sum_{(x,y) \in S} ||f(x; p) - y||^2,$$

often used in regression analysis.
The cost function must be a smooth function in parameter space, so that derivatives $\frac{\partial C}{\partial p}$ can be taken. (When working with derivatives we use the notation $\frac{\partial C}{\partial p} = \nabla_p C$ in the coming sections.) These derivatives are then used to move through parameter space, guiding the algorithm towards better parameters, i.e., towards lower values of the function $C(S, p)$.

More precisely, let the data elements be represented by s_t for $t = 1, \ldots, |S|$ and let $p_1 \in \mathcal{P}$ be some (random) initialization of the parameters. The training procedure is an iterative process, given by

$$p_{t+1} = p_t - \lambda \frac{\partial C(x_t, p_t)}{\partial p}, \text{ for } t = 1, \ldots, |S|, \tag{1}$$

where $\lambda > 0$ represents the *learning rate*.
Note that on each iteration instead of computing the true gradient, i.e., the gradient with respect to the full data set: $\frac{\partial C(S,p)}{\partial p}$, which would be computationally inefficient, only the derivative for one "random" element s_t is used. For this reason this procedure is called *stochastic gradient descent* (SGD). It is common that the input (training) data S is shuffled before use, i.e., s_t is randomly sampled from S without replacement. A compromise between computing the true gradient and the gradient for only a single sampled element s_t is to compute the gradient against a "mini-batch" $B \subset S$ at each step. This mini-batch is again selected uniformly at random, justifying the name mini-batch stochastic gradient descent.

2.1.2 Over-fitting and test data

Even when the data S is assumed to be a reasonable representation of the "true" data $\{(x, y) \in X \times Y : y = g(x)\}$, care must still be taken on how to train on it. Perfectly fitting this data, having found parameters that lead to a global minimum of the cost function, is often suboptimal. This is referred to as over-fitting, and is easiest explained visually (Figure 1).

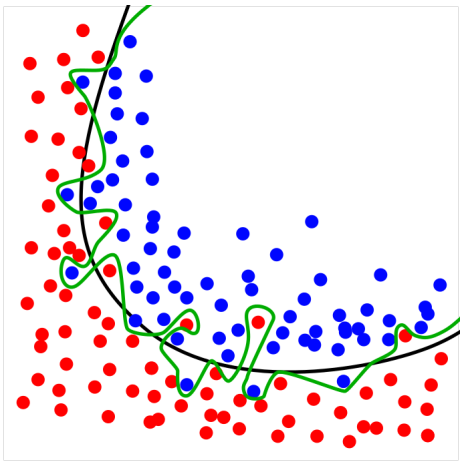


Figure 1: A binary classification task. The green line represents an over-fitted model, while the black line shows a good fit on the underlying pattern. Even though the green line fits the training data better (perfect), it is more likely to lead to higher errors when confronted with unseen new data points.

Usually it is not so clearly visible when over-fitting occurs, especially in high dimensional data sets, with complicated cost functions for which it is difficult to find a global minimum.
Actually, after having used all the training data to optimize the parameters, there is no way of telling how good the model really is when confronted with new unseen data points. The performance on the training data can be a bad measure for the algorithms quality, because we have specifically trained it to perform well on this data, hence training data cannot be assumed to be unbiased data any more.
For this purpose the available data is divided into a training and a test data set. The training procedure uses only a part of the data, say 90% of what is actually available, so that the remaining 10% can give an unbiased measure of the performance. The difference between the error on the unseen testing set and the (smaller) error on the training set is called the *generalization error*.

2.1.3 Hyper-parameters and Cross-validation

Apart from the regular parameters $p \in \mathcal{P}$ that need to be optimized, there are also hyper-parameters. Hyper-parameters determine structural choices about the algorithm or the training procedure. These choices often influence what other parameters there are to optimize, and the optimization itself (e.g. the learning rate λ or the form of cost function used). Because of this we cannot train the hyper-parameters and normal parameters at the same time, as there is no well-defined (let alone differentiable) cost function that combines all of these parameters. This is why *validation data* is used. Having already removed part of the data (which will later be used as test data) we separate yet another part of it that will be used to rate and compare different hyper-parameters. The approach is the following:

1. For a series of possible configurations of hyper-parameters, use training data to optimize the (normal) parameters.
2. For each of the possible configurations of hyper-parameters, together with the optimized parameters, test the algorithm on the validation data.
3. Having found what hyper-parameters lead to the best algorithm, the validation data is no longer needed. It can be merged with the rest of the training data, and the algorithm with the optimal hyper-parameters can be further trained on this set.
4. Finally the algorithm is tested on the test set, which has not been used up until now.

The procedure of partitioning data into complementary subsets, performing some analysis on one subset (the training set) and validating the analysis on the other (the validation set) is called *cross-validation*.
Obviously, a lot depends on how we actually partition into training and validation sets. If we shuffle our data set so that all elements are in a different order,

the training set and validation set will be different, and the resulting algorithm and performance as well. One way to reduce this unwanted variability is to use k-fold cross validation: The original data (excluding test data) is randomly partitioned into k equal sized sub-samples. One round of cross-validation is then performed, each time using a different sub-sample as validation set, and the rest as training set. In the end the validation results are combined (e.g. averaged) over the k rounds to give a more robust estimate of the algorithm's performance for each selection of hyper-parameters.

2.2 The Neural Network

Although artificial neural network ideas exist since the early 1950s, with the first successful implementations published in the 1960s [Iva70], only recent developments in computing power have led to the current frequent use of this type of machine learning algorithm [Sch14]. For high dimensional data involving many non-linear complex patterns, a neural network's high accuracy now makes it the algorithm of choice. Given sufficient training data, the number of parameters can be increased almost arbitrarily compared to previous decades, making this technique beat other algorithms. Neural networks have achieved tremendous success in image and sound recognition, as well as text and time series analysis[Kai+17].
In this section the neural network as a specific machine learning method is described in detail. We start out by describing the functional form of a neural network and the intuitive interpretation from which the name "neural network" originates. After that we give more formal mathematical definitions of the network as we go into the theoretical background that tries to explain its empirical success. However, in light of what is to come, the most important topic of this chapter is the description of the network architecture as a series of consecutive layers of different types.
Although neural networks are also suitable for regression problems, here we specifically treat the neural network as a classification function, and define the output space accordingly.

2.2.1 Mathematical Structure

We again let the input space be $X \subseteq \mathbb{R}^d$, with $d \in \mathbb{N}$ the dimension of the input data.[3] Let Y be the output space. For a classifier the output is usually given as a score for each of the possible labels (e.g. "cat","dog","lizard" etc). If there are N labels we would have $Y \subseteq [0,1]^N$. Although we may at first restate this mathematical set-up whenever it is relevant, we will gradually leave these chosen notations implicit in subsequent chapters.

[3]This can be very high for some learning tasks. Take for example the case of image recognition. The input data might consist of pictures of size 100 by 200 pixels, with pixel color given by a further separation into red, green and blue intensity. In that case we have $d = 100 \times 200 \times 3 = 60000$.

A neural network of $L \in \mathbb{N}$ layers, with respective sizes $d_n \in \mathbb{N}$ for $n \in [L]$ and $d_0 = d$, $d_L = N$, is a function $f : X \to Y$ of the form

$$\begin{aligned} f(x) &= l_L \circ l_{L-1} \circ \cdots \circ l_1(x), \text{ where} \\ l_n &: \mathbb{R}^{d_{n-1}} \to \mathbb{R}^{d_n} \text{ for } n \in [L]. \end{aligned}$$

The *layer functions* l_n are parameterized by weight matrices $W_n \in \mathbb{R}^{d_n \times d_{n-1}}$ and bias vectors $b_n \in \mathbb{R}^{d_n}$. Given a nonlinear, coordinate-wise acting [4] *activation function* σ_n, their form is given by

$$l_n(a) = \sigma_n(W_n a + b_n) \text{ for } a \in \mathbb{R}^{d_{n-1}}. \tag{2}$$

The parameter space of layer $n \in [L]$ can thus be given by $\mathbb{R}^{d_n \times d_{n-1}} \times \mathbb{R}^{d_n}$, so that the total number of parameters is given by

$$P := \sum_{n \in [L]} d_n(d_{n-1} + 1),$$

and the parameter space can be represented as $\mathcal{P} \subset \mathbb{R}^P$. In many of the sections that follow, we refer to the neural network only as a function $f : X \to Y$, with parameters $p = (p_j)_{j \in [P]}$ or $p \in \mathcal{P}$. Thus in its shortest form the neural network is written $f = f(x, p)$, as any other function.
As the last layer of the network is in fact the only layer of which the output is seen, the other layers are usually referred to as *hidden layers*. It is common to also refer to a network of L layers as a network of $L - 1$ hidden layers.
Sometimes the input of the first layer is referred to as the *input layer*. This can be helpful for understanding and visualising the network, even though the input layer is merely a place-holder, and does not have the same mathematical functionality as the other layers.

2.2.2 Neuron interpretation

The first artificial neural network research was inspired by the human brain, which is where the algorithm derives its name from. In the human brain information is passed from one neuron to the other based on electrical and chemical signals. Depending on the strength of incoming signal a neuron can become "activated" and fire a signal to the next neurons in line. This concept is mathematically simulated by an *activation function*. Let $z \in \mathbb{R}$ be some incoming signal reaching a neuron. A simple form of activation function can be given by

$$\sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

In a network, as in the brain, the incoming signal z can actually originate from a large number of different previous neurons connected to the neuron under consideration.

[4] When we refer to coordinate-wise functions σ acting on vectors of varying dimensions, we slightly abuse notation, implicitly assuming to have defined $\tilde{\sigma} : \mathbb{R} \to \mathbb{R}$, and $\sigma(x) = (\tilde{\sigma}(x_1), \ldots, \tilde{\sigma}(x_n))^T$, where n is the dimension of x.

Let there be $m \in \mathbb{N}$ prior neurons with their combined outputs summarized by $a \in \mathbb{R}^m$. Some of the prior neurons may be more strongly connected to the neuron under consideration than others. This can be represented by *weights* $w \in \mathbb{R}^m$, so that the dot product $w \cdot a$ is the full incoming signal.
Activation of the neuron under consideration occurs when a certain threshold is exceeded. This threshold need not be 0, as in (3), but can be shifted up or down by adding some *bias* $b \in \mathbb{R}$, to the incoming signal: $z = w \cdot a + b$. Eventually, this signal then passes through an activation function like (3), and forms the "activated" value a', that can subsequently become (part of) the input of other neurons. Thus, summarizing the above,

$$a' = \sigma(z) = \sigma(w \cdot a + b). \tag{4}$$

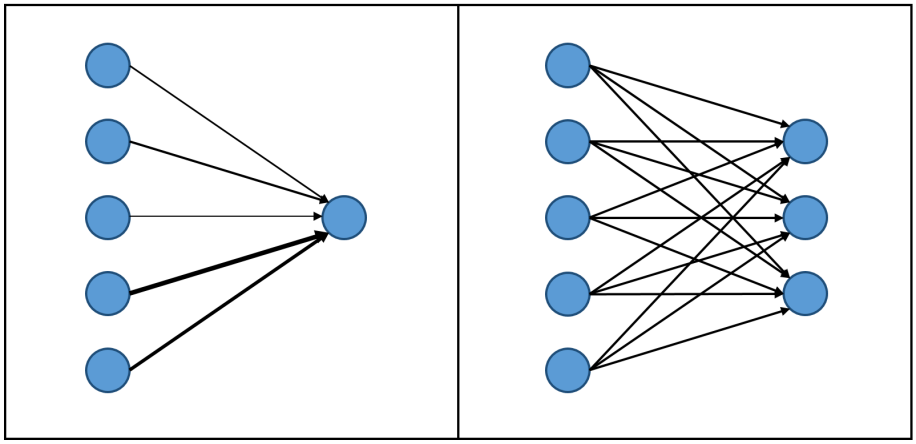


Figure 2: The figure on the left shows 5 neurons connected to one other neuron, showing different connection strengths by the thickness of the arrows. On the right are 2 layers of neurons stacked one after the other, with the 5 neurons of the previous layer each contributing to all 3 neurons of the next layer.

We now imagine the neurons as stacked in layers, depicted in Figure 2, and focus on two adjacent layers, say layer "0" consisting of m neurons and layer "1" consisting of n neurons. We can write an equation like (4) for each of the output neurons separately. Let $a^0 \in \mathbb{R}^m$ be the outgoing signals of the neurons of the prior layer, and let the respective connections to the neurons of the next layer (weights) and thresholds (biases) be given by $w_i \in \mathbb{R}^m$ and $b_i \in \mathbb{R}$, respectively, for $i \in [n]$. The full incoming signal of neuron i is then given by $z_i^1 = w_i \cdot a^0 + b_i$. This signal passes through some activation function σ, to form the input signal of the following layer,

$$a_i^1 = \sigma(z_i^1) = \sigma(w_i \cdot a^0 + b_i), \tag{5}$$

and so on. The above equations can be written in matrix notation, by letting $W \in \mathbb{R}^{n\times m}$ be the *weight matrix* with w_i on the $i-$th row for $i \in [n]$. In the following notation we let σ be a coordinate-wise acting vector function, evaluating and outputting the value for each coordinate separately. Describing all signals moving from layer 0 to layer 1 we then get

$$a^1 = \sigma(Wa^0 + b), \tag{6}$$

which generalizes to (2) for a *deep learning* network, i.e., a network consisting of many layers.
In order to use stochastic gradient descent, the cost function $C(p) = C(S, f(p))$ must have nonzero derivatives with respect to p almost everywhere on $\mathcal{P}$, and consequently, by the chain rule, the same holds for the network function $f(p)$ itself. Hence the activation function σ used in practice is different from equation 3. An intuitive example is to use sigmoidal functions, like the logistic function,

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad z \in \mathbb{R},$$

with the common S-shaped graph depicted in Figure 3.

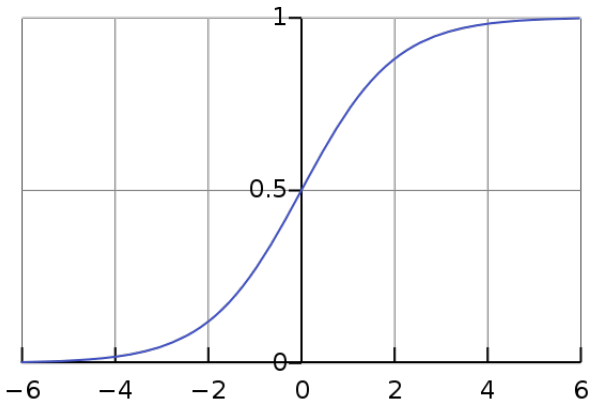


Figure 3: Logistic function

However, in what is known as the "saturated regime" of such an activation function, i.e., for very large $|z|$, the line is flat and the derivative gets close to zero. Small derivatives lead to small steps in the gradient descent iterative process (1), slowing down the learning process. If the derivatives are too small, the algorithm will get stuck in this saturated regime. This is known as the *vanishing gradient problem*. One way to prevent this is to experiment with different activation functions. Due to its empirical success, the most common activation function currently used is the *rectified linear unit*, or *ReLU*, activation function

$$\sigma(z) = \max\{z, 0\}. \tag{7}$$

2.2.3 Theoretical Background

Most of the research on neural networks has been empirical, focusing on getting high accuracies on real datasets and finding new interesting applications. In contrast, viewing the neural network as a family of functions, its underlying mathematical properties have been researched significantly less.
We briefly mention the most important theoretical results. These are mainly on the subject of *approximation*: a reasonable objection for a certain method in machine learning to be used, is if the family of functions that it can approximate is too small, therefore likely not containing the underlying unknown model that the researcher is interested in. A simple linear regression will never be able to recognize and label images of cats and dogs. However, as the theorems below show, no matter how difficult the pattern or function in the data, a neural network can be designed that fits it up to an arbitrarily small error.
In this section we define the neural network more formally, giving two definitions that will be used for further reference. We state without proof two important approximation properties of neural networks that support their popularity and give a partial explanation for how successful deep learning is in many prediction tasks.

Definition 2.1 *Let $n, m \in \mathbb{N}$, $W \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^n$, $p = (W, b)$ and σ a real coordinate-wise function. A **layer** $\mathcal{L}_{(m,n,p,\sigma)}$ is a function $\mathcal{L} : \mathbb{R}^m \to \mathbb{R}^n$ given by*

$$\mathcal{L}(x) = \sigma(Wx + b). \tag{8}$$

*We refer to n as the **size** of the layer, and σ as its **activation function**.*

Definition 2.2 *Let $d, N, l \in \mathbb{N}$, $X \subseteq \mathbb{R}^d$ and $Y \subseteq \mathbb{R}^N$. Furthermore, for $1 \leq i \leq l$, let $L_i = \mathcal{L}_{(n_{i-1}, n_i, p_i, \sigma_i)}$ be a layer of the form (8). A **neural network** $f_{(X,Y,L)}$ is a function $f : X \to Y$ given by*

$$f(x) = L_l \circ L_{l-1} \circ \cdots \circ L_1(x). \tag{9}$$

*The function domain and range imply $n_0 = d$ and $n_L = N$. We call $p := (p_i)_{i \in [l]}$ the **parameters** of the network, l its **depth**, and $max_{i \in [l]}(n_i)$ its **width**.*

For $s, t \in \mathbb{N}$ we define a function class $\mathcal{F}_{(X,Y,L(s,t))}$ consisting of all neural networks with depth $\leq s$ and width $\leq t$.

Theorem 2.3 *[Cyb89] [Hor91] Let $d \in \mathbb{N}$. Let σ be a non-constant, bounded, and continuous real function. Then $\mathcal{F}_{(X,Y,L(2,\infty))}$ is dense in $C(\mathbb{R}^d)$ with respect to the supremum norm.*
In other words, for any $\epsilon > 0$ and any function $g \in C(\mathbb{R}^d)$, there is a $t \in \mathbb{N}$ and a neural network $f \in \mathcal{F}_{(\mathbb{R}^d, \mathbb{R}, L(2,t))}$ such that

$$|f(x) - g(x)| < \epsilon, \, \forall x \in \mathbb{R}^d.$$

Theorem 2.4 *[Lu+17] Let $n \in \mathbb{N}$. For any Lebesgue-integrable function $g : \mathbb{R}^n \to \mathbb{R}$ and any $\epsilon > 0$, there is a maximum depth $s = s(n, g, \epsilon) \in \mathbb{N}$ such that there exists some neural network $f \in \mathcal{F}_{(\mathbb{R}^n, \mathbb{R}, L(s, n+4))}$ for which*

$$\int_{\mathbb{R}^n} |f(x) - g(x)| dx < \epsilon.$$

This result holds even if the layer activation functions are ReLU functions.

These theorems can be further extended to hold for output spaces of arbitrary dimensions by adding independent networks on top of each other, i.e., multiplying the width with the dimension of the output space. Even for this naive way of stacking neural networks, the results are the same: almost any high-dimensional function on a high-dimensional output space can be approximated by a neural network in one of two ways. Either by setting some restriction on the network width, and allowing arbitrary depth, or restricting the depth, and allowing arbitrary width. Even though a good approximations exists, there is no guarantee that a particular training method will find a neural network that is a good approximation.
From the popularity of deep (as opposed to shallow) learning, i.e., networks using many layers, and the results of previous theorems, a natural question arises: is depth or width more important? Empirical work largely indicates preferences for going deeper, see for instance [He+15a]. However, despite the practical success, like on the subject of neural networks in general, there is only little theoretical foundation concerning the role of depth. We briefly discuss two of the most important approaches in this field.

In [Tel16] and [Lu+17], the main concern is the analysis of *depth efficiency* and *width efficiency*, respectively. This comes roughly down to asking how big an increase (i.e., exponential, polynomial) in the total number of nodes is needed after reducing and fixing the width (or depth) of the network, in order to have the same representation power as the original network.
Telgarsky [Tel16] has proved the existence of deep neural networks that cannot be approximated arbitrarily close by any shallow network whose size is exponentially larger: for every positive integer k, there exist neural networks with $\Theta(k^3)$ layers, $\Theta(1)$ nodes per layer, which can not be approximated by networks with $\mathcal{O}(k)$ layers and $o(2^k)$ nodes. This might indicate a stronger representational power of depth over width. However, as Lu et al. [Lu+17] point out, this is not necessarily the case if the same result can be proven the other way around, i.e., if there are also shallow networks that cannot be realized by deep networks whose size is exponentially larger. Lu et al. show that there exist networks such that reducing width requires an increase of the depth that is at least polynomial. Note that this is still not satisfactory for concluding that depth plays the more important role, as the actual necessary increase of depth might still be exponential, rather than polynomial. Proving the importance of depth versus width using this measure of efficiency is posed as a formal open

problem.[Lu+17] Either an exponential lower bound can be found for width efficiency, not indicating any of the two is more important than the other, or a polynomial upper bound is found for width efficiency. The latter case would mean, more concretely, that every wide network can be approximated by a narrow network whose size increase is no more than a polynomial.

Whereas the above theory focused on (mutual) approximation qualities of different neural networks, a different measure of a neural network's strength comes from the number of linear response regions that the input can be separated in.[5] As compositions and sums of linear functions are equally linear, in this sense any deep neural net based on piece-wise linear activation functions can itself be considered a piece-wise linear function. One way to measure its complexity is then by counting the number of linear regions.

It is shown[PMB13; Mon+14] that in the asymptotic limit of many layers, deep networks with ReLU activations are able to separate their input space into exponentially more linear response regions than their shallow counterparts, despite using the same number of computational units. By viewing single neurons as a way of mapping different input spaces into the same output space, they show that activation functions can exploit similar patterns across different input space hyperplanes in the function that is to be fitted.

This last idea is crucial for the current generally accepted explanation as to why deep networks are so successful. In short, adding layers gives the neural network the opportunity to exploit patterns across different subsets of the input space simultaneously, i.e., without the need of extra parameters for each subspace of the input space. As simpler patterns are accumulated to form more complex patterns in deeper layers, this hierarchical structure can be seen to use the same exploitation trick iteratively, and as such deep learning is especially efficient because it makes more efficient use of each parameter.

Consider as a simple example a uni-variate function, symmetric with respect to the origin. Assuming n parameters are required for some model (say a neural net with one hidden layer) to reasonably fit the function for values of $x > 0$, a naive way (not recognizing the symmetry) of fitting the full function would require $2n$ parameters. This can be seen as equivalent to doubling the width of the network. However, symmetry can be exploited by adding an extra layer after the input with only two nodes, noting that $|x| = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)$. Rather than finding parameters to fit input values both larger than and smaller than zero, only one side of the y-axis needs to be fitted, which can then be mirrored to the other side.

[5] A linear response region is a region where changes to the input linearly affect the output. For example, the ReLu activation function contains two linear regions for input z: $z \geq 0$ and $z < 0$.

This reasoning obviously applies to complex hyperplanes and different types of (compositions of) activation functions as well. Prior to training it is not known where any symmetry resides, but the algorithm is allowed to find and exploit this by itself. The intuition is thus that using multiple layers allows the network to summarize complex patterns (often more complex than mere symmetry) in fewer parameters than a non-hierarchical single layer network would need.

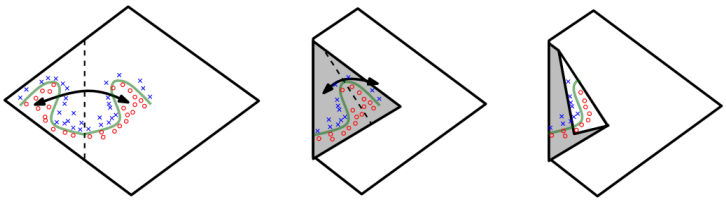


Figure 4: Space folding of 2-D space in a non-trivial way. Note how the folding can potentially identify symmetries in the boundary that it needs to learn. Only the line on the rightmost figure needs to be properly described by parameters. The entire input space on the left can then be fitted using only a few parameters for the folding of 2-D space. (Picture taken from [Mon+14].)

In 2017, Serra et al.[STR17] have further investigated the complexity of deep neural networks, improving both upper and lower bounds on the number of linear regions that a piecewise linear function represented by a deep neural net can attain.
Although results on the number of linear regions do give an indication of the power of deep learning, these results do not immediately explain why deep learning performs so well in practice. The training procedure, although empirically optimized, has been insufficiently studied to give a true theoretical basis for the conjectured strength of this learning method. The consensus that deep learning is one of the best methods in machine learning currently available is largely due to overwhelming empirical evidence.

2.2.4 More Complicated Layers

In most literature, the layers such as defined in (2.1) are referred to as *fully connected* or *dense* layers, as each neuron of a layer is connected (allowed to contribute) to each neuron of the next layer, as visualised in Figure 2. However, as neural networks became more widely used, people have started experimenting with more creative architectures. We discuss two types of layers, namely pooling and convolutional layers, and show how these can in fact be rewritten as special cases of regular (fully connected) layers.
Before giving mathematical definitions of the layers, we give more informal intuitive descriptions. The main characteristic of convolutional and pooling layers is the fact that they are used in problems where input data elements can be attributed a location, e.g. pixels of an image, values in a time series, words in

a text paragraph, and that the local patterns or correlations in the input can be exploited. For convolutional layers the main advantage comes from sharing parameters for recognizing patterns across different locations in the input, thus reducing computation time. Note that ordinary fully connected layers do not actually use this information. As each node is connected to each node of the next layer, due to this symmetric build-up the nodes of any layer (even the input layer, i.e., the place-holder for input data elements) might have just as well been shuffled prior to training, with no effect on the algorithm's performance.

In cases where the relative locations of input elements are exploited, it becomes practical to change the way layers are depicted. Although a layer is theoretically equivalent to a vector, it is in these cases more insightful to depict it as a grid of nodes, rather than as a simple column. Take the example of a neural net training on square images of 30 by 30 pixels. Rather than simply depicting the first layer (i.e., the input layer) as a column of 900 pixel values, it can be depicted as a grid of 30×30 pixel values, where subsequent matrix multiplications or other operations can implicitly treat this grid as a vector in $\mathbb{R}^{900}$.
Sticking to this 2-D notation of layers, pooling and convolutional layers can be seen as sliding a pooling or convolutional window over this grid, and computing a function on the inputs falling in this window. For example, a common type of pooling, known as average-pooling, takes a window of size, say, 2×2, and reproduces a new layer consisting of all the average values found in squares of size 2×2 of the original grid. For a layer shaped as a 30×30 grid, this results in a next layer of shape 15×15, where each element of the next layer represents a local average found in the previous grid. Similarly, max-pooling layers return maximum values rather than average values of the elements in each window. Both types of pooling are depicted in Figure 5.

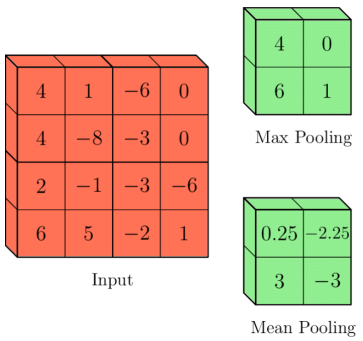


Figure 5: Max-pooling and average-pooling layer examples. The pooling layers have 4 nodes, and are fed input data from a previous layer of 16 nodes. In this case the pooling windows are 2×2 squares, laid over the four corner regions of the previous layer. Each corner of the output layer corresponds to a corner of the input layer.

Note that in contrast to regular layers, pooling layers have no parameters. The computation (taking the average or maximum value) as well as the structure (the size of the window and how it shifts through the grid) is the same before and after training, independent of the data fed to the algorithm. Pooling layers learn nothing new from the data, but just conveniently summarize information. This is different for convolutional layers. Here too a window shifts through the grid, but this time the function applied on input values falling in that window can be compared to taking a weighted average, where the weights are layer parameters. More generally, the function that is locally applied is actually represented by taking the inner product between window values, *the kernel*, and the underlying grid values (layer inputs). The resulting value is stored in a node of the next layer. An example is given in Figure 6.

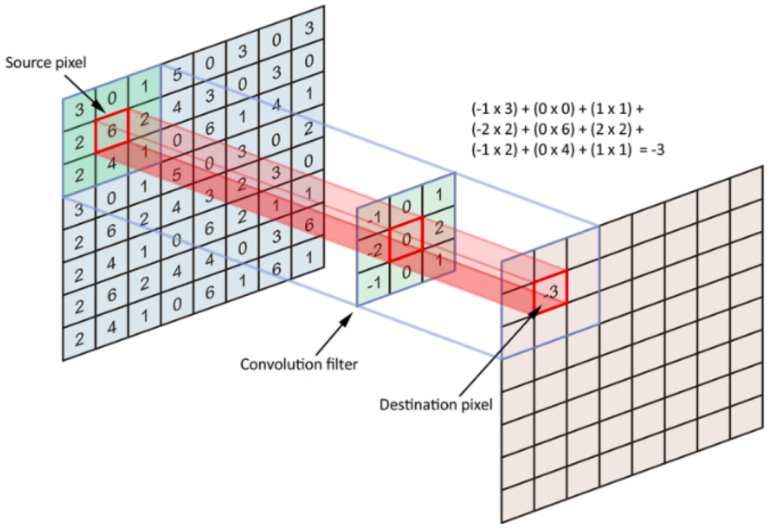


Figure 6: An example of one computation in a convolutional layer. The same 3×3 convolutional filter is used for computing every grid point of the layer. This convolutional filter is also called a kernel. The precise values of the kernel change during training: these values are parameters that can be optimized.

The intuition behind convolutional layers is that the convolutional window returns information about whether or not a certain pattern has been discovered at a certain location. For instance, in the example of Figure 6, the kernel is specialized in recognizing horizontal contrast: a high value in the destination pixel is associated with high values to the right and low values to the left of the source pixel. On the other hand, if there is no horizontal contrast, the outcome on the destination pixel is 0. Rather than fixing what type of pattern the layer should recognize, e.g. horizontal contrast, vertical lines, dog snouts, and so on, that choice has been left for the algorithm to figure out, by letting the kernel be given by parameters that can change during training.

Similar to normal layers, convolutional layers usually also have an activation function and a bias which are applied after the dot product with the kernel is taken.
Finally, it is often the case that a single convolutional layer is used to search for multiple distinct patterns simultaneously. For example, an entire image is scanned for vertical lines as well as horizontal and diagonal lines in one layer. In this case not just one kernel is used, but a large number of say, K independent kernels each move through the entire grid with the same convolutional window, but different kernel values and biases. The outcome is then depicted as a 3-d grid: each location on the 2-d grid gets a column of K outcome values, representing the extent in which each of the K patterns have been found on that specific location. To make matters even more complex, searching for a single pattern in this new "3 dimensional" layer by means of *another* convolutional layer, now requires a 3-d convolutional window, i.e., a convolutional bar. This convolutional bar moves through the input in the original 2 dimensions of the first 2-d grid, but overlaps (i.e., takes the inner product) with the input in 3 dimensions.[6]
It can happen that the input grid of the pooling or convolutional layers cannot be evenly divided into the given fixed shapes of the pooling or convolutional windows. For example, a 5×5 grid cannot be split into separate 2×2 pooling windows. There are two concepts for dealing with these issues. The first is called **stride**, which determines if pooling or convolutional windows should overlap, and by how many grid points. It is common for pooling windows to have no overlap, whereas convolutional windows always overlap. The second concept is called **padding**. This refers to the practice of adding dummy zeros around the borders of the grid in order to always have a full convolutional window with which a dot product with the kernel can be taken.

We now give formal definitions of pooling and convolutional layers. These definitions are more general then what was just discussed in order to include possible padding and stride decisions. Although the definitions may thus allow very exotic and illogical pooling/convolutional windows to be built, it is mostly important that the above described layers are indeed cases of the following definitions.

Definition 2.5 *Let $n, m \in \mathbb{N}$, $n \leq m$. Let the indices of the **pooling window** at location $j \in [n]$ be given by $I_j \subset [m]$. An **average pooling layer** is a function $\pi_{avg} : \mathbb{R}^m \to \mathbb{R}^n$ given by*

$$\pi_{avg}(x)_j = \frac{\sum_{i \in I_j} x_i}{|I_j|} \text{ for } x \in \mathbb{R}^m.$$

*We refer to n as the **size of the layer** and $max_j |I_j|$ as the **size of the pooling window.***

[6] A popular coding package for implementing neural networks in Python is called *Tensorflow*. This name is inspired by the fact that layers can be interpreted as 3-d grids of nodes, i.e., tensors, and the signal of these tensors flows through a network of multiple layers.

Definition 2.6 *Let $n, m, s, K \in \mathbb{N}$. Let the indices of the* convolutional window *corresponding to the target location $j \in [n]$ be given by $I_j \subset [m]$ with $|I_j| \leq s$. Let kernels be given by weights $w_k \in \mathbb{R}^s$ and biases $b_k \in \mathbb{R}$, for all $k \in [K]$. Let the indices of the window corresponding to location $j \in [n]$ be matched to the kernel (weight) indices by the injective mapping $g_j : I_j \to [s]$. Summarize the parameters by stacking the weight vectors as rows, $w = (w_k)_{k\in[K]}$, and write $p = (w, b)$. A* convolutional layer *with activation function σ, window $I = (I_j)_{j\in[n]}$ and K kernels of size s, is a function $c_{(p,I,\sigma)} : \mathbb{R}^m \to \mathbb{R}^{n\times K}$ given by*

$$c(x)_{jk} = \sigma(\sum_{i\in I_j} w_{kg_j(i)}x_i + b_k),\ \forall j \in [n],\ k \in [K], x \in \mathbb{R}^m.$$

The choice of I is fixed, but the layer parameters *p can change during training. For the purpose of representing the layer consistently as a single column vector, rather than a matrix or grid, any bijection $h : [n] \times [K] \to [nK]$ can be used, and the definition can be changed accordingly:*

$$c(x)_{h(j,k)} = \sigma(\sum_{i\in I(j)} w_{kg_j(i)}x_i + b_k),\ \forall j \in [n],\ k \in [K].$$

Starting from these definitions, we now prove that average-pooling and convolutional layers are special cases of regular layers as defined in 2.1.

Theorem 2.7 *Any average-pooling layer following definition 2.5 can be written as a layer following definition 2.1.*

Proof 2.7.1 *We give a proof by construction. Let $\pi_{avg} : \mathbb{R}^m \to \mathbb{R}^n$ be an average pooling layer with pooling windows I_j for all $j \in [n]$. Let $W \in \mathbb{R}^{n\times m}$,$b = \mathbf{0} \in \mathbb{R}^n$ and σ be the identity function. Let $W_{ji} = \mathbb{1}_{i\in I_j} 1/|I_j|$. Let $p = (W, b)$. Then the layer $\mathcal{L} = \mathcal{L}_{(m,n,p,\sigma)}$ is identical to the average pooling layer π_{avg}:*

$$\begin{aligned}
\mathcal{L}(x)_j &= (\sigma(Wx + b))_j \\
&= (Wx)_j \\
&= \sum_i W_{ji}x_i \\
&= \sum_i \mathbb{1}_{i\in I_j} x_i/|I_j| \\
&= \sum_{i\in I_j} \frac{x_i}{|I_j|} \\
&= \pi_{avg}(x)_j.
\end{aligned}$$

Theorem 2.8 *Any convolutional layer according to definition 2.6 can be written as a layer following definition 2.1.*

Proof 2.8.1 *We give a proof by construction. Let $c_{(p,I)} : \mathbb{R}^m \to \mathbb{R}^{nK}$ be a convolutional layer of K kernels with parameters $p = (w, b)$ and windows I_j for $j \in [n]$, with window indices mapped to kernel weight indices using g_j, in accordance with definition 2.6. Let $h : [n] \times [K] \to [nK]$ be a bijection. Let $W \in \mathbb{R}^{nK \times m}$, $B \in \mathbb{R}^{nK}$ and $P = (W, B)$. Let $B_{h(j,k)} = b_k$ and $W_{h(j,k)i} = \mathbb{1}_{i \in I(j)} w_{kg_j(i)}$ for all $k \in [K]$, $j \in [n]$ and $i \in [m]$. Then the layer $\mathcal{L} = \mathcal{L}_{(m,nK,P,\sigma)}$ is identical to the convolutional layer $c_{(p,I,\sigma)}$. We show this by letting $k \in [K]$ and $j \in [n]$, and let $l = h(j, k)$. As h is a bijection, and k and j have been chosen arbitrarily, it suffices to show that the layers are identical for l:*

$$\begin{aligned}
\mathcal{L}(x)_l &= \sigma((Wx)_l + B_l)) \\
&= \sigma(\sum_i W_{li} x_i + B_l) \\
&= \sigma(\sum_i W_{h(j,k)i} x_i + B_{h(j,k)}) \\
&= \sigma(\sum_i \mathbb{1}_{i \in I(j)} w_{kg_j(i)} x_i + b_k) \\
&= \sigma(\sum_{i \in I(j)} w_{kg_j(i)} x_i + b_k) \\
&= c(x)_{h(j,k)} \\
&= c(x)_l.
\end{aligned}$$

We summarize the importance of theorems 2.7 and 2.8 in the following corollary.

Corollary 2.8.1 *Any neural network consisting of a combination of fully connected, average pooling and convolutional layers can be remodeled as an identical neural network of only fully connected layers, in accordance with definition 2.2. This is true for the network when all parameters are set. However, during the training procedure, some parameters of the fully connected layers that mimic pooling and convolutional layers should be coupled or held constant for the identical properties to hold.*

2.3 Training Neural Networks

When creating the network $f = f(x, p)$, it is common for the parameters p to be randomly initialized, and the model clearly is a bad predictor or fit of the underlying function. As discussed in Section 2.1.1, a differentiable proxy measure of the parameter quality, i.e., the cost function, is used together with stochastic gradient descent to change the parameters. In this section we derive the proper form of the cost function and describe the optimal techniques and issues arising during training. We assume a classification task of N labels and apply *one-hot encoding* to our data labels. This means that the labels are represented by standard basis vectors of $\mathbb{R}^N$, rather than ordinary numbers.[7] Let $E_N := \{e_i\}_{i\in[N]}$ be the standard basis vectors of $\mathbb{R}^N$, then it holds for all $(x, y) \in S$ that $x \in \mathbb{R}^d$ and $y \in E_N$.

Although the outputs of the unknown labeling function g are restricted to E_N, the same does not hold for the neural network function $f : X \to Y$. Note that f is differentiable almost everywhere by design, and it should be, as stochastic gradient descent requires the cost function $C(S, f(p))$ to have existing derivatives almost everywhere.

As a result of these properties of f, there is never a simple binary result stating whether $f(x)$ is a good fit or a bad fit of the true label y. Rather, some fits seem better than others, e.g. $f(x) = (0.03, 0.02, 0.89)$ seems a better fit than $f(x) = (0.45, 0.69, 0.91)$ for the true label $y = (0, 0, 1)$. The challenge of putting an actual value on the quality of the fit is dealt with by finding a proper cost function $C(S, f(p))$.

2.3.1 The Cross-entropy loss function

As the research field of neural networks is for a big part empirical, there are many different cost functions used in practice, each fine-tuned to get the best prediction results. However, we focus on only one specific cost function, called the *cross-entropy loss function*, because it has a solid mathematical foundation in statistical inference.

One particularly convenient way to make sense of values $f(x) \in Y\backslash E_N$ is to choose to interpret $f(x)$ as a categorical distribution on E_N. For this purpose we choose the last layer's activation function, σ_L, in such a way that a proper probability vector is outputted, thus arranging that $\sum_{i\in[N]} f_i(x) = 1$. A common way to do this is to use the *softmax* function

$$(\sigma(z))_i = \frac{e^{z_i}}{\sum_j e^{z_j}},\ z \in \mathbb{R}^N. \tag{10}$$

We can then interpret f by letting $f_i(x, p) = \mathbb{P}(e_i|x, p)$ for all $i \in [N]$.

[7] The reason for this is to overcome some treacherous situations, i.e., if the labels for 'cat', 'dog', and 'lizard' are 0, 1 and 2 for instance, it might happen that some unclear image resembling both a cat and a lizard is labeled as a dog (the label 1 being the average of 0 and 2). Representing labels as independent basis vectors prevents such peculiar and meaningless relations.

Interpreting $f(x,p)$ as a distribution on E_N now allows us to use maximum likelihood estimation to find the best parameters p^*: the optimal parameters are those corresponding to the highest probability of exactly reproducing the labels S_Y when performing the random independent experiment of selecting a label $y \sim f(x,p)$ for all $x \in S_X$.
In other words, if the labels S_Y *were* in fact generated by a categorical distribution of the form $f(x,p)$ for $x \in S_X$, then the the probability of observing S_Y is highest for the parameters

$$\begin{aligned} p^* &= \text{argmax}_p \mathbb{P}[S_Y | S_X, f(p)] \\ &= \text{argmax}_p \prod_{(x,y)\in S} \mathbb{P}[y|x,p] \\ &= \text{argmax}_p \prod_{(x,y)\in S} \prod_{i\in[N]} f_i(x,p)^{y_i} \\ &= \text{argmax}_p \log\Big(\prod_{(x,y)\in S} \prod_{i\in[N]} f_i(x,p)^{y_i} \Big) \\ &= \text{argmax}_p \sum_{(x,y)\in S} \sum_{i\in[N]} \log(f_i(x,p)^{y_i}) \\ &= \text{argmax}_p \sum_{(x,y)\in S} \sum_{i\in[N]} y_i \log(f_i(x,p)). \end{aligned}$$

Finding the exact maximum of the above expression is practically impossible, hence we define a cost function as follows:

$$C(S, f(p)) := -\sum_{(x,y)\in S} \sum_{i\in[N]} y_i \log(f_i(x,p)), \tag{11}$$

and use batch stochastic gradient descent to find its minimum. This cost function is called the *cross-entropy* cost function. We conclude with the summarized approach for batch stochastic gradient descent using this cost function.
Starting with some random initial $p_1 \in \mathcal{P}$, a learning rate $\lambda \in \mathbb{R}_{>0}$ and a partion of the training data set S_X into M batches $B_t \subset S_X$, for $t \in [M]$, we search for the optimal parameters by the iterative process

$$p_{t+1} = p_t - \lambda \frac{1}{|B_t|} \frac{\partial C(B_t, f(p_t))}{\partial p}, \text{ for } t \in [M]. \tag{12}$$

2.3.2 Back-Propagation

Taking the derivative of equation 11 requires finding the derivative of $f(x,p)$ with respect to p, which is a tedious task. At every iteration t the derivative needs to be computed again for new p and B_t. Training a deep network can thus take a long time, especially if there is a lot of data. Therefore it is important to efficiently compute the derivatives at each step, using a minimal number of computations. The method developed for this purpose is called *back-propagation*.

Let, in accordance with definition 2.2 a neural net of L layers be given by

$$\begin{aligned} f(x) &= l_L \circ \cdots \circ l_1(x) \\ l_n(a) &= \sigma(W^n a + b^n) \text{ for } a \in \mathbb{R}^{d_{n-1}}, \end{aligned}$$

where we now use superscripts denoting the layer number, so as not to confuse with individual matrix or bias elements. We also omit the layer dependencies of the activation functions for clarity, assuming if you will that all activation functions are of the same type σ. We now use introduce some new notation. Let

$$a^n(x) := l_n \circ \cdots \circ l_1(x)$$

be the (intermediate) output of the n-th layer of the network for $n \in [L]$, with $a^0 = x$. We also refer to a^n as the *activations* of layer n. Furthermore let

$$z^n(x) := W^n a^{n-1}(x) + b^n \tag{13}$$

be the value of the n-th layer *before* it passes through the activation function σ, thus having

$$a^n(x) = \sigma(z^n(x)). \tag{14}$$

Let p_n be all parameters of the n-th layer. Since we need all parameters $p = (p_n)_{n\in[L]}$, we need the following derivative for each $n \in [L]$, omitting the dependence on x for clarity[8]:

$$\begin{aligned} \frac{\partial C}{\partial p_n} &= \frac{\partial C}{\partial f}\frac{\partial f}{\partial p_n} \\ &= \frac{\partial C}{\partial f}\frac{\partial f}{\partial a^n}\frac{\partial a^n}{\partial p_n} \\ &= \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial a^n}\frac{\partial a^n}{\partial z^n}\frac{\partial z^n}{\partial p_n} \\ &= \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial a^{L-2}} \cdots \frac{\partial a^{n+1}}{\partial a^n}\frac{\partial a^n}{\partial z^n}\frac{\partial z^n}{\partial p_n}. \end{aligned}$$

After memorizing a single forward pass through the network, i.e., inputting x and computing all $a^n(x)$ sequentially, each derivative can then be obtained in $O(1)$ time, and computing $\frac{\partial a^L}{\partial a^n}$ requires $O(L)$ computations. To get all derivatives for all parameters $p = (p_n)_{n\in[L]}$ then seems to require $O(L^2)$ computations. However, using a simple memorization trick, the computational time can be reduced significantly. Rather than computing each derivative independently, we save the intermediate value

$$\delta^n(x) = \frac{\partial C}{\partial z^n}(x),$$

[8]For differentiating functions of multiple variables in multiple dimensions, i.e., $f : \mathbb{R}^n \to \mathbb{R}^m$, we use the following convention: $(\partial f/\partial x)_{ij} = \partial f_i/\partial x_j$. Using this convention means that if $g : \mathbb{R}^m \to \mathbb{R}^k$ and $h(x) = g(f(x))$ we can simply write $\partial h/\partial x = \partial g/\partial f \cdot \partial f/\partial x$ as we would in one dimension. Note that matrix multiplication is implicit.

and apply the following recursion:

$$\begin{aligned}\delta^n &= \frac{\partial C}{\partial z^{n+1}}\frac{\partial z^{n+1}}{\partial z^n} \\ &= \delta^{n+1}\frac{\partial z^{n+1}}{\partial a^n}\frac{\partial a^n}{\partial z^n} \\ &= (\delta^{n+1}W^{n+1}) \odot \sigma'(z^n), \end{aligned} \tag{15}$$

where we used 13 and 14 to compute the derivatives. We use the notation $\odot$ to represent the hadamard product (pointwise multiplication of matrices). The recursion starts with

$$\begin{aligned}\delta^L &= \frac{\partial C}{\partial z^L} \\ &= \frac{\partial C}{\partial f}\frac{\partial a^L}{\partial z^L} \\ &= \frac{\partial C}{\partial f} \odot \sigma'(z^L). \end{aligned} \tag{16}$$

Finding the remaining δ^n for $n = L-1, \ldots, 1$ requires $O(L)$ steps. This method is called back-propagation, because the recursion starts at the end of the network, and moves backwards, at each step back-propagating the sensitivity of the cost function with respect to a certain layer.
From δ^n we can now easily retrieve the derivatives with respect to the actual parameters, as

$$\begin{aligned}\frac{\partial C}{\partial p_n} &= \frac{\partial C}{\partial z^n}\frac{\partial z^n}{\partial p_n} \\ &= \delta^n\frac{\partial z^n}{\partial p_n}. \end{aligned}$$

Using once more equation 13 produces the following elegant equations:

$$\begin{aligned}\frac{\partial C}{\partial b_i^n} &= \tfrac{\partial C}{\partial z^n}\tfrac{\partial z^n}{\partial b_i^n} &&= \delta_i^n, \\ \frac{\partial C}{\partial W_{ij}^n} &= \tfrac{\partial C}{\partial z^n}\tfrac{\partial z^n}{\partial W_{ij}^n} &&= a_j^{n-1}\delta_i^n. \end{aligned}$$

We conclude with the pseudo-code for the backpropagation algorithm:

Input x, i.e., set the corresponding activation $a^0 = x$ for the input layer.

for $n = 1, \ldots, L$ **do**
  $z^n = W^n a^{n-1} + b^n$
  $a^n = \sigma(z^n)$
end for

Compute $\delta^L = \frac{\partial C}{\partial f} \odot \sigma'(z^L)$

for $n = L-1, \ldots, 1$ **do**
$\delta^n = (\delta^{n+1} W^{n+1}) \odot \sigma'(z^n)$
end for

The gradient of the cost function is given by
$\frac{\partial C}{\partial b_i^n} = \delta_i^n$ and $\frac{\partial C}{\partial W_{ij}^n} = a_j^{n-1} \delta_i^n$.

2.3.3 Parameter initialization

Prior to starting the stochastic gradient descent algorithm, an initial value for the parameters must be selected. If initial values are chosen identical per layer, from the back-propagation and gradient descent algorithms we can see that these parameters remain identical: the neural net does not perform symmetry-breaking. Hence already in the initialization of parameters some predetermined variation is required.
The current accepted approach is to let the biases have initial value of zero, and break the symmetry by randomly initialize the weights with mean 0. The chosen variance of the initial weights distribution, by determining the average size of the parameters, then heavily influences the course of the training procedure.
To understand why the average size of the parameters is relevant, we note that the gradients of the cost function with respect to parameters of layer n are proportional in size to δ^n, and focus on the recursion of the back-propagation algorithm:

$$\delta^n = (\delta^{n+1} W^{n+1}) \odot \sigma'(z^n). \tag{17}$$

If δ^n is on average slightly smaller or larger than δ^{n+1} for all n, this difference builds up as the back-propagation recursion moves deeper towards the beginning of the network. The gradients with respect to the parameters of the first layers of the network might then be either very small or very large, which is known as the vanishing or exploding gradient problem, respectively. As can be seen in equation 17 the choice of activation function σ has some influence on this. However, a proper initialisation of the weights is equally important to keep the gradients at practical levels.
Assume that the weights in W^n (i.e., each element W_{ij}^n) are initialized i.i.d. with mean zero and variance v_n^2. Note that from equation 17 it follows that $\mathbb{E}[\delta^n] = 0$. As a proper measure of the size we therefore look at $\mathbb{E}[(\delta_j^n)^2]$, for some $j \in [d_n]$ (Remember that the number of nodes in layer n is given by d_n,

so that also $\delta^n \in \mathbb{R}^{d_n}$.) We now find that:

$$\begin{aligned}
\mathbb{E}[(\delta_j^n)^2] &= \mathbb{E}[(\sum_{i=1}^{d_{n+1}} \delta_i^{n+1} W_{ij}^{n+1})^2 (\sigma'(z_j^n))^2] \\
&= \mathbb{E}[\sum_{i,k=1}^{d_{n+1}} \delta_i^{n+1} \delta_k^{n+1} W_{ij}^{n+1} W_{kj}^{n+1} (\sigma'(z_j^n))^2] \\
&= \mathbb{E}[\sum_{i=1}^{d_{n+1}} (\delta_i^{n+1})^2 (W_{ij}^{n+1})^2 (\sigma'(z_j^n))^2] \\
&= \sum_{i=1}^{d_{n+1}} \mathbb{E}[(\delta_i^{n+1})^2] v_{n+1}^2 \mathbb{E}[(\sigma'(z_j^n))^2] \\
&= d_{n+1} \mathbb{E}[(\delta_j^{n+1})^2] v_{n+1}^2 \mathbb{E}[(\sigma'(z_j^n))^2].
\end{aligned}$$

In the last steps we assumed independence among δ^{n+1}, W^{n+1} and $\sigma'(z^n)$. Even if this is true after initialization, it is of course not guaranteed to remain this way during training. However, assuming the covariance terms will at least remain relatively small, there is some insight to be gained from this derivation, as has also been shown empirically[GB10]. To stabilize the gradients we now use the result

$$\frac{\mathbb{E}[(\delta_j^n)^2]}{\mathbb{E}[(\delta_j^{n+1})^2]} = d_{n+1} v_{n+1}^2 \mathbb{E}[(\sigma'(z_j^n))^2]. \tag{18}$$

More explicitly put, to make sure the gradients for all parameters are similar in size, the right-hand side of the above equation should remain close to 1. As mentioned, choosing a convenient activation function, like the ReLU, who's derivative is either 1 or 0, already helps. Note that sigmoid type activation functions have close to zero derivatives in their saturated regimes, which form the majority of their domain.
We thus take σ to be the ReLU function. From equation 13, and the initialisation of b^n as zero and W^n with mean zero, we find that z_j^n has a symmetric distribution. The probability for this value falling in the flat or in the linear domain of σ are then equal, so that $\mathbb{E}[(\sigma'(z_j^n))^2] = \frac{1}{2}$. Filling in this value in equation 18 leads to proposing the initialisation variance

$$v_n^2 = \frac{2}{d_n}.$$

Although exact results on how these equations remain valid *during* training are lacking, the derivation does shed some light on the standard approach and empirical success of initialising parameters such that $b^n = 0$ and $\mathrm{Var}[W_{ij}^n] = 2/d_n$ for every layer n in a ReLU network. [GB10] [He+15b]

2.4 Methods to prevent neural networks from overfitting

Overfitting was briefly explained in Section 2.1.2. We list a few examples of approaches that help against over-fitting.

- Using an ensemble of networks can give a smoother result than a single, possibly better trained, network.
- Dropout is the procedure of temporarily removing individual nodes of the network during each training step with some probability.
- Regularization refers to adding another term to the cost function, penalizing the complexity of the model by regulating for the size of the parameters. A common type of regularization is $L2$ regularization, referring to the type of norm used. The new cost function C' is then given by $C' = C + \alpha \sum_i p_i^2$ for some constant $\alpha > 0$ determining the regularization strength. It can be shown that this is equivalent to assuming a normal prior distribution with mean 0 on the parameters, and applying maximum likelihood estimation. (See appendix A.)
- Expansion of training data can help a network reducing over-fitting by making the training data itself more general.
- Early stopping is one of the simplest methods to prevent over-fitting. This usually involves careful monitoring of algorithm performance as a function of training iterations.

As most of these methods are self-explanatory or relatively simple, we only elaborate further on the first two topics: dropout and network ensembles.

2.4.1 Ensemble of networks

Instead of spending all the capacity (time and/or computing power) on training one network, it can be helpful to split the attention towards training multiple networks, and combining these for a better result. This is known as an ensemble method in machine learning, as an ensemble of learned models is used instead of just one model. The two main ways to do this are called *boosting* and *bagging*. Boosting trains a number of constrained (e.g. limited in complexity) or weaker models *in sequence*, where each model learns from the mistakes of the one before it. All weak learners are then combined into a single strong learner. On the other hand, bagging uses complex models and trains them separetely, combining the stronger models *in parallel*. While boosting uses simple models and tries to "boost" their aggregate complexity, bagging uses complex models and tries to smoothen out their predictions. Especially the latter method is commonly used to prevent overfitting. We use some elementary statistics to show how.
For neural networks one type of the bagging approach can for example be achieved by training multiple networks on random different, preferably disjunct, subsets of the training data S, or by using different initial parameters. Having trained M networks, say, we then get a series f_m, $m \in [M]$.

For the final result we average those networks such that

$$f(x) = \frac{1}{M} \sum_{m=1}^{M} f_m(x). \tag{19}$$

To understand how this helps against over-fitting, we need to understand where the error in a model can come from.[9] Let $g(x)$ be the unknown true model generating the labels y. If $g(x)$ is probabilistic, assume that $g(x) - \mathbb{E}[g(x)|x]$ is independent of x. Note that there is randomness in the observation of a new data point x as well as in the subsequent labeling $g(x)$. Given a model f, the expected size of the error of this model takes into account both sources of randomness. The mean squared error (MSE) of the actual model f is then given by

$$\begin{aligned}
\mathbb{E}[(f(x) - g(x))^2] &= \mathbb{E}[(f(x) - \mathbb{E}[g(x)|x] + \mathbb{E}[g(x)|x] - g(x))^2] \\
&= \mathbb{E}[(f(x) - \mathbb{E}[g(x)|x])^2] + \mathbb{E}[(g(x)|x] - g(x))^2] \\
&\quad +2\mathbb{E}[(f(x) - \mathbb{E}[g(x)|x])(\mathbb{E}[g(x)|x] - g(x))] \\
&= \mathbb{E}[(f(x) - \mathbb{E}[g(x)|x])^2] + \mathbb{E}[(g(x)|x] - g(x))^2],
\end{aligned}$$

where the last equality follows from the assumption that $g(x) - \mathbb{E}[g(x)|x]$ is independent of x and the fact that this expression has expectation zero. We refer to the first term as the *reducible* error, and the second term as the *irreducible* error. Note that indeed the second term is independent of the model $f(x)$, and only represents noise of the labeling function.
We now let the reducible error of a network f be given by $\epsilon(x) := f(x) - \mathbb{E}[g(x)|x]$, and let ϵ_m be similarly defined for networks f_m of the ensemble. Assume that the networks are unbiased independent estimators, i.e., $\mathbb{E}[\epsilon_m(x)] = 0$ and $\mathbb{E}[\epsilon_m(x)\epsilon_n(x)] = \mathbb{E}[\epsilon_m(x)]\mathbb{E}[\epsilon_n(x)] = 0$. In that case we see that the reducible error of the ensemble average network f (Equation 19) is given by

$$\begin{aligned}
\mathbb{E}[\epsilon(x)^2] &= \frac{1}{M^2}\mathbb{E}[(\sum_{m=1}^{M} \epsilon_m(x))^2] \\
&= \frac{1}{M^2}\mathbb{E}[\sum_{m=1}^{M}\sum_{n=1}^{N} \epsilon_m(x)\epsilon_n(x)] \\
&= \frac{1}{M^2}\sum_{m=1}^{M}\sum_{n=1}^{N} \mathbb{E}[\epsilon_m(x)\epsilon_n(x)] \\
&= \frac{1}{M^2}\sum_{m=1}^{M} \mathbb{E}[\epsilon_m(x)^2] \\
&= \frac{1}{M}\mathbb{E}[\epsilon_1(x)^2].
\end{aligned}$$

[9] Although many assumptions are made in the following derivation, and there are more complete ways of decomposing errors (more generally taking bias and variance into account) we do not go into too many details and just show the general principle at work. A more appropriate discussion into MSE and Bias-Variance decomposition can be found in [Zav17].

Hence by using an ensemble of size M, the reducible part of the MSE is reduced by a factor $1/M$. Technically reducing the MSE is not necessarily the same as reducing over-fitting, as there are many reasons why the model might have a high MSE. However, if over-fitting is the cause of some (unbiased) error, then the ensemble method can be seen to "smoothen" the result, thereby reducing the MSE. Intuitively, if the separate models are unbiased estimators, then the different ways in which multiple models are over-fitting cancel each-other out when averaging.

2.4.2 Dropout

In dropout, on every iteration of the training procedure, i.e., in determining every parameter update during a batch gradient descent step, a random selection of nodes is removed from the network, to be added back again in the next step. The random selection is realized by removing every node with a certain probability α. Input neurons are sometimes exempted from this procedure, as this would just remove information. After the removal of some of the nodes, the parameters that remain are altered, while the parameters of nodes that were removed are remembered for when they are placed back in, but do not influence the output of the network.
After restoring the dropout nodes, the process is repeated, selecting a new random subset of nodes to delete. By repeating this processs, the network will learn a set of parameters, say p', that have been learned under conditions in which a fraction of α of the nodes have been dropped out. However, at the end of the training procedure, *all* nodes are used to form the finished model. This means that there will be more nodes active than in the situation for which the parameters were optimized for. To correct for this an extra scaling factor is needed: all weights are multiplied by $1 - \alpha$.
To see why this is appropriate, and to better understand why dropout is successful, we appropriately redefine the input of a certain node of the network during training. Let for $n \in [L]$, $i \in [d_n]$, the input of node i in the n-th layer be given by

$$z_i'^n = \sum_{j=1}^{d_{n-1}} W_{ij}'^n a_j'^{n-1} D_j + b_i'^n, \tag{20}$$

where D_j is a random Bernoulli variable which has the value 0 with probability α, and 1 with probability $1 - \alpha$. We use the notation W' and b' for parameters that have been optimized for a network with node inputs given by equation 20, as opposed to the original equation 13. The different parameters in turn lead to different corresponding node inputs and activations z' and a'.
Let $f_\alpha = f_\alpha(x, p')$ be a network based on equation 20, with a dropout probability of α and corresponding optimized parameters $p' = p'_\alpha$.
The idea now is to find the optimal parameters p to use for the full (i.e., with no nodes removed) network, using the parameters p' that are found during training. More explicitly, it turns out that the final network $f(x, p)$ can be considered as

the expected value of a network of the form $f_\alpha(x, p')$. Considering that taking the expected value is the theoretical equivalent of averaging a large sample of networks, the dropout method can in fact be seen as an ensemble method, revealing why it works well against over-fitting.
To show that this is the case, let the parameters p indeed be chosen such that

$$f(x, p) = \mathbb{E}[f_\alpha(x, p')], \tag{21}$$

where the expectation is taken with respect to the dropout random Bernoulli variables D that are embedded in the network f_α.
Using again the notations as introduced in the back-propagation algorithm, we see that equation 21 is equivalent to

$$a^L = \mathbb{E}[a'^L],$$

omitting the dependency on x for clarity. Choosing p to comply with this, we work by induction. Note first that the input layer activations remain constant: $a^0 = a'^0 = \mathbb{E}[a'^0] = x$. Now assume $a^{n-1} = \mathbb{E}[a'^{n-1}]$. We choose the parameters p_n such that $z_i^n = \mathbb{E}[z_i'^n]$, thus

$$\begin{aligned}
\sum_j W_{ij}^n a_j^{n-1} + b_i^n &= \mathbb{E}[\sum_j W_{ij}'^n a_j'^{n-1} D_j + b_i'^n] \\
&= \sum_j W_{ij}'^n \mathbb{E}[a_j'^{n-1}]\mathbb{E}[D_j] + b_i'^n \\
&= \sum_j W_{ij}'^n a_j^{n-1}(1-\alpha) + b_i'^n
\end{aligned}$$

This holds if we let $b^n = b'^n$ and $W^n = W'^n(1-\alpha)$. If this is sufficient, the only necessary change to the parameters will be to scale the weights found in the dropout procedure by $(1-\alpha)$. However, to finish the induction argument, we still have to arrange that $a^n = \mathbb{E}[a'^n]$, having only reached the conclusion that $z^n = \mathbb{E}[z'^n]$. In case σ is not a linear function (which usually is not the case) the latter does not simply implicate the first statement. This issue is treated extensively in [BS13] for linear, logistic and softmax activation functions σ, with extensions to (among others) ReLU functions in [BS14]. By carefully inspecting the expected value over all possible sub-networks, it is found that the so-called "dropout approximation" $\mathbb{E}[a'^n] \approx \sigma(\mathbb{E}[z'^n])$ is negligible. In that case we see that

$$\begin{aligned}
\mathbb{E}[a'^n] &\approx \sigma(\mathbb{E}[z'^n]) \\
&= \sigma(z^n) \\
&= a^n,
\end{aligned}$$

concluding the proof of the induction step.
In other words, choosing weights p derived from weights p' by $W^n = W'^n(1-\alpha)$ for all layers $n \in [L]$, leads to a network $f(x, p)$ given by

$$f(x, p) = \mathbb{E}[f_\alpha(x, p')|x],$$

showing that dropout is in fact an ensemble method.

2.5 Optimizers

Given M training data batches $B_t \subset S_X$, for $t \in [M]$, and some initial $p_1 \in \mathcal{P}$, the classical training procedure uses equation 12,

$$p_{t+1} = p_t - \lambda \frac{1}{|B_t|} \frac{\partial C(B_t, f(p_t))}{\partial p}, \text{ for } t \in [M],$$

with some learning rate $\lambda > 0$. Note that this procedure is nothing more than a method of finding the minimum of $C(S_X, p)$. There are in fact many more of such iterative optimization techniques for finding the minima of (cost) functions, also called *optimizers* when more specifically referring to the bundled lines of programming code written for implementing these techniques. Although their main purpose in the current context is the training of neural networks, such optimizers will be used more generally, which is why we have dedicated a separate chapter to these techniques.
The main optimizer in use, which we shall also stick to for our implementations, is called *Adam*, and is described in section 2.5.3. We explain it by first going over to other (older) optimizers, namely *AdaGrad* and *RMSProp*, as they form the inspiration and building blocks from which Adam originated.

2.5.1 AdaGrad

Recall that the hyper-parameter λ determines the learning rate, reflecting the step-size we allow parameters to move with in the direction of the (negative) gradient. If λ is set too small, the parameter update will be very slow and it will take a large number of iterations to achieve small cost C. However, if it is set too large, the parameters might move all over the space $\mathcal{P}$ and never lead to acceptable values of C at all. Setting λ just right has a large impact on the success of the optimization procedure. However, the high-dimensional non-convex nature of C can lead to different sensitivities in different dimensions of the parameter space P. The learning rate can be too small in some dimension and too large in another dimension.
One obvious way to mitigate this problem is to explicitly choose different learning rate parameters for each dimension, i.e., different λ_j for each $j \in [P]$. However, this quickly leads to an infeasibly large number of hyper-parameters[10], requiring different solutions to be found.
One of the first optimizers incorporating different learning rates for different dimensions of the parameter space is called *AdaGrad*, short for adaptive gradient algorithm [DHS11]. We briefly describe how the method works, without going into the details of the theory behind it.

[10] Recall that the hyper-parameters are fixed during training and are optimized by measuring their success on validation data sets. Optimizing hyper-parameters is a tedious procedure, requiring large amounts of data and many independent training runs. As the number of hyper-parameters increases, the number of possible combinations of hyper-parameter values increases exponentially.

Let for $t \geq 1$ the gradient of the cost function be given by

$$g_t := \frac{1}{|B_t|} \frac{\partial C(B_t, f(p_t))}{\partial p}, \tag{22}$$

and let the matrix G_t be the sum of the outer products of the gradients up and until time-step t,

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^T.$$

Adagrad is now based on the iteration

$$p_{t+1} = p_t - \frac{\lambda}{\sqrt{G_t}} g_t, \tag{23}$$

where we slightly abuse notation by leaving matrix diagonalization and inversion implicit and directly write square roots and division as if working with scalars. We do this more frequently throughout the coming sections, with vector operations also being implicitly computed component-wise.
In practice a small quantity ϵI, where I is the identity matrix and $\epsilon \in \mathbb{R}_{>0}$, is added to prevent dividing by zero. Also, because computing the square root of a high-dimensional matrix is costly, the square root of $\text{diag}(G_t)$ is used as a reasonable approximation instead[DHS11]:

$$p_{t+1} = p_t - \frac{\lambda}{\sqrt{\epsilon I + \text{diag}(G_t)}} g_t. \tag{24}$$

For one element $p_{t+1}^{(i)}$, $i \in [P]$, the parameter update is now given by

$$p_{t+1}^{(i)} = p_t^{(i)} - \frac{\lambda}{\sqrt{\epsilon + G_t^{(ii)}}} g_t^{(i)}. \tag{25}$$

Note that the effective learning rate is thus different for each parameter i, as it is scaled with respect to the accumulated squared gradient at each iteration,

$$G_t^{(ii)} = \sum_{\tau=1}^{t} (g_\tau^{(i)})^2.$$

If G is not approximated by its diagonal matrix, we are in fact also taking into account interactions between parameters when determining the influence of a change in one parameter. In fact G is often used as an approximation of the Hessian matrix of the cost function [Nil+19].
Either way, this accumulation of squared gradients acts as a decay mechanism, slowing decreasing the effective learning rate over time, settling the learning in a good place avoiding oscillations. The decay is made parameter-dependent by dampening more in the direction of dimensions that have on average larger (squared) gradients throughout the optimization procedure.

2.5.2 RMSProp

Over the course of training steps get smaller and smaller in the Adagrad algorithm, because the sum of squared gradients gets larger. In convex optimization this resulting decay mechanism makes sense, because when minima are approached we want to slow down. However, in non-convex cases it can lead to problems, slowing down too much and getting stuck on saddle points. The *RMSProp* optimizer, standing for root mean square propagation [TH12], therefore uses a moving average of the squared gradients, rather than the full accumulated value.
Because storing and updating a large window of parameter gradients is costly, especially when the number of parameters is in the order of millions, a single decay factor is used. Given g_t as in equation 22 and a decay constant (hyper-parameter) $\gamma \in [0, 1]$, let $G_0 = 0$ and let G_t now be given by

$$G_t = \gamma G_{t-1} + (1 - \gamma)(g_t \odot g_t) \text{ for } t \geq 1. \tag{26}$$

The update is again given by equation 25, noting however that G is now defined differently:

$$p_{t+1}^{(i)} = p_t^{(i)} - \frac{\lambda}{\sqrt{\epsilon + G_t^{(i)}}} g_t^{(i)}. \tag{27}$$

2.5.3 Adam

Note that RMSProp uses an estimate of the (moving) mean of the squared gradients, i.e., G_t as given in equation 26. The method is thus based on the second-order raw moment of the gradients, using estimates for that value to create distinct learning rates. More explicitly, the assumption is that

$$G_t \approx \mathbb{E}[g_t \odot g_t].$$

The most popular optimizer currently used, called *Adam*[KB14], further exploits moment estimates of gradients. Its name is derived from "adaptive moments estimation". In particular, the first-order and second-order moments of gradients are used, representing the mean and the uncentered variance. Let $F^0 = 0$, $\gamma_1 \in [0, 1]$ a hyper-parameter representing a decay factor, and

$$F_t = \gamma_1 F_{t-1} + (1 - \gamma_1)(g_t) \text{ for } t \geq 1. \tag{28}$$

Thus F_t represents the moving average estimator of the gradient mean. Similarly we have G_t defined as in equation 26, i.e.,

$$G_t = \gamma_2 G_{t-1} + (1 - \gamma_2)(g_t \odot g_t) \text{ for } t \geq 1,$$

with $G_0 = 0$ and $\gamma_2 \in [0, 1]$ representing the decay factor for the moving average estimator of the second-order moment.

Because F_0 and G_0 are initialized as 0, the estimators have a bias towards 0 in the initial steps of the iteration, especially noticeable when the decay rates are small. We show how to get rid of this bias in $F_t \approx \mathbb{E}[g_t]$, noting that a similar derivation yields the result for G_t. As $F_0 = 0$, we find (from recursion 28) that

$$F_t = (1-\gamma_1)\sum_{\tau=1}^{t}\gamma_1^{t-\tau}g_\tau.$$

Taking the expected value on both sides yields

$$\begin{aligned}
\mathbb{E}[F_t] &= (1-\gamma_1)\sum_{\tau=1}^{t}\gamma_1^{t-\tau}\mathbb{E}[g_\tau] \\
&= (1-\gamma_1)\sum_{\tau=1}^{t}\gamma_1^{t-\tau}\mathbb{E}[g_\tau - g_t + g_t] \\
&= \mathbb{E}[g_t](1-\gamma_1)\sum_{\tau=1}^{t}\gamma_1^{t-\tau} + (1-\gamma_1)\sum_{\tau=1}^{t}\gamma_1^{t-\tau}\mathbb{E}[g_\tau - g_t] \\
&\approx \mathbb{E}[g_t](1-\gamma_1)\sum_{\tau=1}^{t}\gamma_1^{t-\tau}
\end{aligned}$$

The approximation is caused by ignoring the residual value on the right-hand side of the third equality. This term is small when g_τ is approximately stationary, or when the exponential decay rate is chosen sufficiently large enough that the moving average assigns small weights to gradients far in the past. Continuing from this approximated value, some concluding algebra leads to

$$\begin{aligned}
\mathbb{E}[F_t] &\approx \mathbb{E}[g_t](1-\gamma_1)\frac{1-\gamma_1^t}{1-\gamma_1} \\
&= \mathbb{E}[g_t](1-\gamma_1^t).
\end{aligned}$$

We thus find that a slightly better estimate of $\mathbb{E}[g_t]$ is given by

$$\hat{F}_t := \frac{F_t}{1-\gamma_1^t},$$

and similarly

$$\hat{G}_t := \frac{G_t}{1-\gamma_2^t}.$$

Using these unbiased moments, the parameter updates for the Adam optimizer are given by:

$$p_{t+1}^{(i)} = p_t^{(i)} - \frac{\lambda}{\sqrt{\epsilon + \hat{G}_t^{(i)}}}\hat{F}_t^{(i)}. \tag{29}$$

Note that apart from replacing G by the better estimate $\hat{G}$, the most striking difference between Adam and RMSProp comes from using an estimator $\hat{F}$ for the gradient g_t, rather than its true value. This estimator takes into account previous values of g_t, causing the update direction to be influenced by some memory of prior gradients. This behavior is also called *momentum* in the content of optimizers, alluding to the physical process of letting a ball roll on the slope of some "optimization landscape". Momentum prevents the ball from getting stuck in a small hole on the slope, similar to how the optimization iteration here is prevented from getting stuck in a local minimum of the cost function. Momentum also speeds up the learning process, in the same sense that a ball speeding down a slope does not immediately slow down or change direction when it encounters small bumps.

3 Explainability in Machine Learning

In this chapter we deal with an important drawback of deep learning, namely the lack of explainability. We specifically describe what we mean by this in the following introduction, before treating several different approaches that aim to resolve this issue. We describe and implement a Bayesian neural network, and compare it to a regular neural network to show the improved model behavior. We implement the LIME and RDE explanation methods for this Bayesian neural network and show the computed feature importance underlying the model's decisions for some examples.

3.1 Black Box models

Aided by new implementation techniques and better computers, neural networks have become increasingly successful in a quickly expanding research field of possible implementations. However, the dominant interest in high accuracy has skewed the research towards improving predictive powers of deep learning algorithms, at the cost of the explainability and interpretability of these predictions. The lack of fundamental understanding as to how deep learning exactly works, results in the current general view that a trained neural network is practically a black box algorithm. Our goal is to "open the black box", and find explanations for neural network decisions.

The term "Black Box" is often arguably unjustly used, namely when it refers to a person's lack of understanding of algorithms that are in fact not intrinsically impossible to understand, but just require background knowledge of mathematics and computer science. In fact, most machine learning algorithms can readily produce the reasoning behind their predictions. Decision trees and random forests rely on input feature importance by design, whereas statistical methods like generalised linear models and principal component analysis deliver interpretable general outcomes rather than just a set of labeled data elements.

For deep learning this is fundamentally different. A trained deep neural network is an extremely complicated function by design. State of the art neural networks can have tens of billions of parameters, see for instance [Sha+17; TGR15], using 137 and 160 billion parameters, respectively. In fact, since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years [GBC16], see for instance Figure 7.
With such large numbers of neurons, layered to form increasingly complex behaviour of a high-dimensional function, it becomes nearly impossible to understand how these models work.

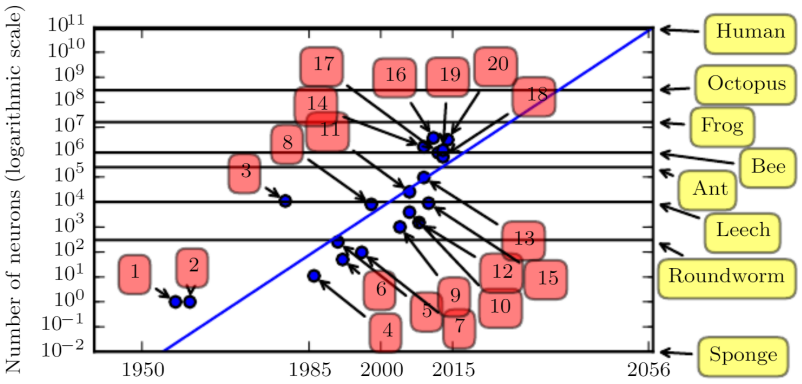


Figure 7: Increasing neural network size over time.[GBC16] Different breakthroughs are shown, numbered chronologically from 1958 until 2014. Note that the number of neurons is not the number of parameters, which form the connections between neurons. The natural brain equivalent of these connections would be synapses, of which there are over 100 trillion.

More explicitly put, given a new data point x and a trained network f, there is no easy way to understand *why* a label $f(x)$ is assigned. The trained function might be very successful on the test dataset, achieving a high accuracy and thus proving to be a good predictor on unseen data points. However, the user of the model is empty-handed when asked for an explanation of the outcome. With the benefit of being able to fit increasingly complicated functions to data, comes the problem of not being able to explain the outcomes of these functions.

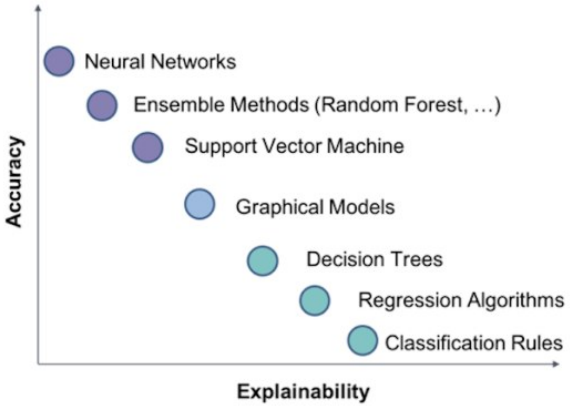


Figure 8: The machine learning methods that achieve a higher accuracy in more difficult tasks are commonly less explainable, and vice versa.

3.1.1 The need for Explainable AI

The most common areas where deep learning is currently used are settings with high-dimensional data containing complex patterns, where explainability is not required, and only the predictive performance counts: as long as the algorithm predicts well, there is in principle no urgency to explain how some sentence has been translated, or a why a face has been recognized. Errors in such applications usually have no dramatic consequences: there are no ethical, accountability, or large financial complications.
However, as neural networks are increasingly stealing the show as the best algorithms in terms of accuracy, new applications are being discovered in a quick pace. In some of these new fields, like medicine and finance, explainability becomes important, as doctors and CEO's are involved with decisions that require clear reasons and justifications.
Also, in some cases simply knowing the outcome of an algorithm is insufficient to act upon it. In the detection of financial crime, like money laundering, flagging data points as suspicious is more efficient if a clear reason for this suspicion accompanies the model's outcome.

When algorithms become more involved in society and people's personal lives, ethical arguments further increase the call for more explainability. If fed with the wrong data, models might discriminate or unknowingly evolve other kind of unwanted behavior. Whereas human decision making may implicitly contain ethical considerations, algorithms only train for accuracy.
Finally, as machine learning methods become more intelligent, there are possibilities for us to learn form them, rather than simply use them blindly. Neural networks can increasingly be used in scientific fields where data is abundant, such as economics, biomedical sciences and astronomy. For enduring scientific progress it is necessary to understand the building blocks leading towards the next discoveries and theories, rather than getting plain results without context.

3.1.2 Holistic view of existing methods

In light of what has just been discussed, specifically noting the use of billions of parameters, one might opt to achieve explainability of neural network decisions by reducing the complexity of the network in the first place.
Although some work has been done on this topic, using specific architecture restrictions like connecting smaller sub networks or enforcing sparse weight matrices, the trend within the deep learning field is similar to the general trend in machine learning, depicted in Figure 8: reducing an algorithm's complexity might lead to some explainability, but at the cost of accuracy.[AB18; YZS19] The networks achieving the highest accuracies are often the least explainable.

An alternative approach to explainability in machine learning is to leave the black box intact, and rather use separate techniques to investigate the model, giving *post-hoc* explanations. Most research on explainable AI belongs to this post-hoc class, and we describe two of these methods in detail in the coming chapters: LIME [RSG16] in Chapter 3.3 and RDE [Mac+19] in Chapter 3.4.
Explainability methods can be *model-agnostic* or *model-specific*, referring to whether they can be applied to any model, or whether they are specifically defined for one machine learning technique. The larger part of explainability research applies to model-agnostic methods.
Yet another possible distinction to be made is between local and global explainability. Global explainability methods aim at understanding the whole logic of a model, following the entire reasoning leading to all the different possible outcomes. On the other hand, local explainability seeks to explain the rules for a single prediction or decision. The distinct attempts at AI explainability are summarized in Figure 9.

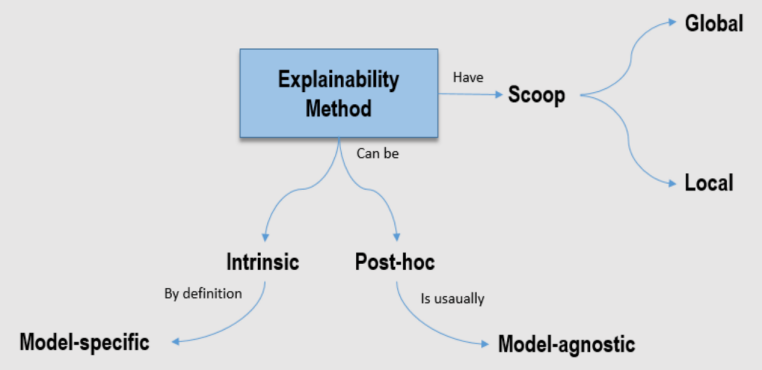


Figure 9: A pseudo ontology of explainable AI methods taxonomy, from [AB18].

The methods that we implement in the coming chapters are post-hoc, local explainability methods. Both LIME and the newly developed EVD method are model-agnostic, whereas RDE as well as the material in Chapter 3.2 rely on the unique architecture of the neural network. Note that while a model-agnostic method can be applied to every model, the implementation of the method can often still be largely dependent on the specific model architecture.
In the field of post-hoc model-agnostic local explainability methods, the act of further distinguishing or developing new methods is complicated by a striking issue: what is an explanation?
Most notably missing in explainable machine learning research is a clear definition or consensus of what an explanation is, let alone a measure of success in achieving this explanation. [Lip16]

3.1.3 What is an explanation?

There are many different ways in which the explainability challenge is described. The literature is filled with terminology: opening the black box, interpreting model outcomes, explaining underlying reasons, visualising decisions, and so on. While the scope of this research field can be narrowed down to model-agnostic, local explainability and post-hoc methods, in essence these methods still vary as to what they are trying to achieve. Given an input element $x \in X \subset \mathbb{R}^d$, a neural network $f : X \to [0,1]^N$ and an output $y = f(x)$, we distinguish the following (possibly overlapping) approaches:

1. Feature importance: which features were most important for the decision and which features were least important? How much has each of the d features in x contributed to the final outcome y?
2. Sensitivity analysis: how much does the output y change, or how likely is it to change, for each of the d features, when x is slightly changed along the dimension of that feature? Along which of the d dimensions is this sensitivity largest?
3. Comparing to examples: which similarly labeled elements $x' \in S_X$ does x resemble, hinting at similar reasons for the outcome label y?
4. Inspecting interpretable components: which components of x that are easy to understand do we recognize, and can we valuate their respective importance? Or, taking a more model-specific approach: can we decompose the model into a simple, possibly even linear, combination of several explainable sub-models representing such components?
5. Inspecting model behavior: if we perturb x (in more creative ways than when simply computing input sensitivity) or change intermediate computed activations within the network, how does that affect the information flow through the network and what can we learn from this?

A wide range of methods can roughly be found to belong to one or more of these categories. As regards the methods that we implement, we mention that LIME can be found to belong to the first and fourth category of the above list, whereas the RDE method belongs to the first category. To further substantiate the above defined categorization of methods, we go over some more examples for each category.
For methods applying sensitivity analysis we refer to [WFL15] and [SVZ13], where saliency maps of image data are created to indicate what pixels the model is most sensitive to, also suggesting where it is focusing on. Methods based on example comparisons are for instance [Mik+13] and [Car+99], where nearest neighbor elements from the training data are used. Decomposing a model into several interpretable sub-models is done in [YZS19], whereas more specifically inspecting network behavior is done in [LMJ16] after removing nodes or connections and in [MOT15] after changing the input to enhance specific node activations.

We find that in trying to achieve explainability, most methods only briefly, if at all, treat the notion of what an "explanation" exactly is. More commonly these methods only aim at proxy measures that shed light on the black box, but that are not necessarily meant to be explanations themselves. By focusing foremost on the ultimate goal of *explaining neural network decisions to users*, the intermediate goal of *mathematically formulating an explanation* is arguably undervalued or overlooked.
While the results of all of these methods are striking, in that they successfully make networks and their decisions more insightful in one way or another, we believe that there is still a lot to gain in more formally defining explanations. We propose a two-step approach of first finding a mathematical definition for an explanation, which should be applicable to different learning tasks and models, and secondly trying to interpret this explanation. The interpretation problem can be tuned to the specific learning task under consideration, while the explanation, once properly defined, can be found by a mathematical (optimization) procedure, independent of human understanding and in a model-agnostic way. Note that as such it can be possible that an explanation is found that is difficult to interpret, but is a (or *the*) correct explanation nonetheless.[11] We elaborate more on this topic in Section 4.2.1. Our proposed mathematical definition of an explanation in that section, as well as the discussion heralding it, helps formalizing and intuitively validating the above discussed ideas and perspective.

3.2 Bayesian Neural Networks

A common issue with neural networks is that the output is a point estimate: no uncertainty bounds are given. What's more, because outputs are forced to represent probabilities, due to the soft-max activation function in the last layer (see Section 2.3.1), this point estimate can appear overly confident when confronted with input elements that are unlike anything seen previously in the training dataset. For instance, when a classifier has been trained on recognizing cat and dog images, and an image of a lizard is fed as input, unintended behavior in the network might result in all sorts of values accumulating in the last layer. Unless these values are close to each-other, the soft-max function likely squishes one class output probability score and maximizes the other, leading to an overconfident decision for one class.
To overcome this issue, and to be able to generate a measure of uncertainty in the neural network prediction, the *Bayesian neural network* (BNN) has been developed [Bis97]. While not strictly a pure explainability method, it tries to tackle some of the same issues arising when using standard "black box" neural networks. Also, in some applications a proper confidence bound reduces the need for an explanation.

[11] This partly solves the paradoxical problem arising from the impossibility of explaining AI models that are more "intelligent" than humans, and owe their black box nature simply to the bounds of human understanding.

In this chapter we first discuss the theory behind the Bayesian neural network, as well as the structural differences compared to regular neural networks. We then go over the training procedure for these types of networks and explain the Flipout method that is used to improve results. We end the chapter with implementation results on the MNIST dataset of digits.

3.2.1 Mean-field Variational Bayesian inference

Recall that the standard training procedure for a neural network via optimization is (from a probabilistic perspective) equivalent to maximum likelihood estimation (MLE) for the weights, see also Section 2.3.1. More explicitly for a network $f(p)$ and a dataset S we derived the cross-entropy cost function from

$$C(S, p) = -\log(\mathbb{P}(S|p)).$$

Minimizing this cost function comes down to maximizing the probability of the data being generated from the categorical distribution $f(p)$. From a probabilistic perspective this methods lacks theoretical justification, partly because it ignores uncertainty that we may have in deciding on the proper parameter values. From a practical perspective this method is problematic because it can lead to overfitting.
Of course overfitting can be reduced by adding a regularization term. As we show in appendix A, using l^2 regularization with some regularization parameter $\alpha > 0$, so that the new cost function becomes

$$C'(S, p) = C(S, p) + \alpha \sum_i p_i^2, \tag{30}$$

in fact comes down to inducing a prior distribution on the weights, namely

$$p_i \sim \mathcal{N}\left(0, \frac{1}{2\alpha}\right). \tag{31}$$

This approach is already better in practice, as it helps against overfitting. In theory it comes down to maximum a posteriori estimation, which is similar to maximum likelihood estimation but is enhanced with the use of a prior distribution on the data. Note however, that the theoretical justification is still lacking, and we also still get a point estimate from this training procedure, with no uncertainty bounds on a given output.
Preferably we would have a parameter distribution that is actually based on some analysis of the data, which can then be used to generate uncertainty bounds. More specifically, if the parameters are actually found to follow some distribution $p \sim Q$, we can use this distribution to generate an infinite ensemble of networks by drawing parameters independently from Q. In other words, we can thus use a predictor of the form

$$f_{\text{BNN}}(x) := \mathbb{E}_{p\sim Q}[f(x, p)].$$

Not only does this method have more solid theoretical justification, as the uncertainty in the parameter selection is properly taken into account using the actual training data, we also have access to a distribution for a given network output of $f_{\mathrm{BNN}}(x)$, rather than a point estimate. This distribution follows directly from the output's dependence on the parameters p, which follow a known distribution.
We are free to decide how large an ensemble of networks is used, and note that these networks do not need to be separately trained, as would be the case for a naive ensemble method (see Section 2.4.1). A simple draw $p \sim Q$ is sufficient to generate a whole new network.

But how do we infer a distribution on the parameters from the data? Using Bayes rule for the conditional density $P(p|S)$ we may write that

$$P(p|S) = \frac{P(p,S)}{P(S)} = \frac{P(S|p)P(p)}{P(S)} \propto P(S|p)P(p),$$

where $P(S), P(p)$ are the densities of S and p, and $P(S|p)$, $P(p,S)$ corresponding conditional and joint densities. However, we are now interested in actually finding $P(p|S)$ as a function of p, instead of simply maximizing it with respect to p. Unfortunately, for most neural networks this density function cannot be calculated analytically, neither is there a way to efficiently sample from it.
In what is known as mean-field variational Bayesian inference, a suggested workaround for this problem is to approximate $P(p|S)$ with a more tractable density function $Q_\beta(p)$, choosing function parameters β that minimize the discrepancy. [PBJ12; Gra11]
For this purpose the *Kullback-Leibler divergence* is used, which is a measure of how one probability distribution is different from a second, reference, probability distribution [KL51]. For continuous density functions Q and P the KL-divergence is given by

$$D_{\mathrm{KL}}(Q \parallel P) = \int Q(x) \log\left(\frac{Q(x)}{P(x)}\right) dx,$$

which can be interpreted as the expected difference of the logarithms of (elements drawn from) Q and P, where expectation is taken with respect to Q. Note the asymmetry in this definition: we more specifically refer to $D_{\mathrm{KL}}(Q||P)$ as the *KL divergence of Q from P*.
In the current case, the KL-divergence of $Q = Q_\beta(p)$ from $P = P(p|S)$ is thus given by

$$D_{\mathrm{KL}}(Q_\beta(p) \parallel P(p|S)) \quad = \quad \int Q_\beta(p) \log\left(\frac{Q_\beta(p)}{P(p|S)}\right) dp,$$

and the optimal β^* is to be found by:

$$\begin{aligned}
\beta^* &= \arg\min_{\beta} \int Q_\beta(p) \log\left(\frac{Q_\beta(p)}{P(p|S)}\right) dp \\
&= \arg\min_{\beta} \int Q_\beta(p) \log\left(\frac{Q_\beta(p)P(S)}{P(S|p)P(p)}\right) dp \\
&= \arg\min_{\beta} \int Q_\beta(p) \log\left(\frac{Q_\beta(p)}{P(p)}\right) dp - \int Q_\beta(p) \log(P(S|p)) dp \\
&= \arg\min_{\beta} D_{\mathrm{KL}}(Q_\beta(p) \parallel P(p)) - \mathbb{E}_{p\sim Q_\beta}[\log(P(S|p))]
\end{aligned}$$

As we will further elaborate on in Section 3.2.3, this resulting expression, also called the *variational free energy*, is the cost function we want to minimize when training a Bayesian neural net. We shall from now on denote it as:

$$\mathcal{F}(S, \beta) := D_{\mathrm{KL}}(Q_\beta(p) \parallel P(p)) - \mathbb{E}_{p\sim Q_\beta}[\log(P(S|p))]. \tag{32}$$

This cost function embodies a trade-off between letting the parameters have values as if drawn from a prior $P(p)$, while simultaneously accounting for the extra information received from observing the data S. Indeed, the first term in the above expression aims to reduce the divergence between the estimated distribution of the parameters and its prior, while the second term, when minimized, maximizes the expected log-likelihood of the data.

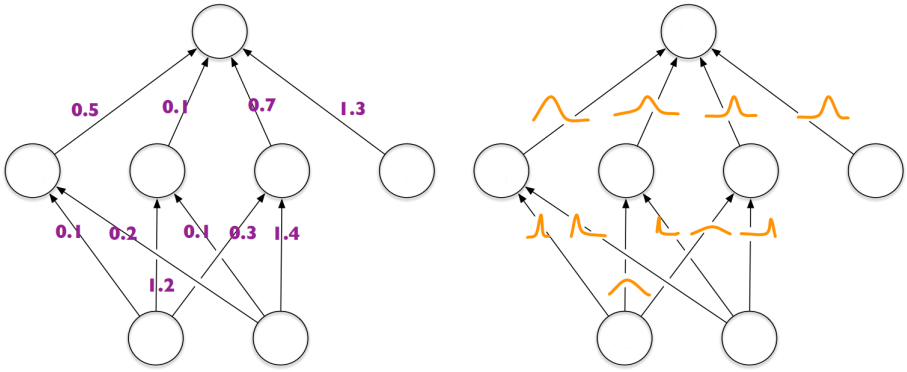
As was the case for the cost function in standard neural network training, this cost function too cannot be exactly minimized. The gradient descent optimization procedure will be addressed in Section 3.2.3.

3.2.2 The Bayesian network format

Before digging deeper into the mathematical derivations and procedures of the training process, let us pause and get an understanding of what a Bayesian neural network actually looks like.

The architecture of the network can be the exact same as that of a standard neural network, although each parameter is now replaced with distribution, see Figure 10.

Figure 10: Visualization of standard neural network (left) and Bayesian neural network (right) architectures. The regular neural network is depicted as having fixed weights and biases (here the bias is represented by an extra node on the right), while the Bayesian neural network has the same connections, but variable parameters represented by some probability distribution. The picture is taken and adapted from [Blu+15].



With the original architecture's parameters replaced by random variables, a Bayesian neural network's parameters are in fact those that define the distributions of these random variables. More explicitly, we let $p \sim Q_\beta$ represent the parameters of the standard network $f(x, p)$. The Bayesian neural network is now the function given by

$$f_{BNN}(x, \beta) = \mathbb{E}_{p \sim Q_\beta}[f(x, p)].$$

In practice the expectation is not computed analytically, and we often implicitly use the Bayesian network in its approximate form

$$f_{BNN}(x, \beta, n) := \frac{1}{n} \sum_{i=1}^{n} f(x, p^{(i)}), \text{ with } p^{(i)} \sim Q_\beta \text{ for all } i \in [n]. \tag{33}$$

The number of parameters of a Bayesian neural network need not be significantly higher than the number of parameters of a regular neural network with the same architecture. If, for example, every single parameter p_j for $j \in [P]$ is independently normally distributed, the actual number of parameters in β is twice the original number P of parameters: a mean and a variance for each parameter p_j is needed.

3.2.3 Training

We now go over the training algorithm for Bayesian neural networks. We eventually show that the back-propagation algorithm in Section 2.3.2 is needed again, but the steps leading up towards it are different. The technique comes from the paper by Kingma and Welling [KW13], further generalized and improved in [Blu+15].

We start by repeating the cost function, equation 32, that is used for training:

$$\mathcal{F}(S,\beta) \quad = \quad D_{\mathrm{KL}}(Q_\beta(p) \parallel P(p)) - \mathbb{E}_{p\sim Q_\beta}[\log(P(S|p))].$$

Now recall from Section 2.3.1 that the original cost function was computed as the negative log-likelihood of the data, i.e.,

$$C(S,p) = -\log P(S|p),$$

so that we can thus write

$$\mathcal{F}(S,\beta) \quad = \quad D_{\mathrm{KL}}(Q_\beta(p) \parallel P(p)) + \mathbb{E}_{p\sim Q_\beta}[C(S,p)].$$

To make things more concrete, we show the derivations for the specific form of the posterior Q_β and prior $\mathbb{P}(p)$ that we use in our implementation (Section 3.2.4), namely diagonal Gaussian distributions.[12] For this case the first term of the cost function can be analytically integrated. Let $Q_\beta(p)$ be a diagonal Gaussian density function with mean μ and diagonal covariance matrix represented by the vector $\sigma \in \mathbb{R}^P$, thus $\beta = (\mu, \sigma)$, and let the prior distribution of p, $P(p)$, be given by $\mathcal{N}(0, I)$. Then we find (in appendix B) that

$$D_{\mathrm{KL}}(Q_\beta(p) \parallel P(p)) = \frac{1}{2}\sum_{j=1}^{P}(\mu_j^2 + \sigma_j^2 - \log(\sigma_j^2) - 1) \tag{34}$$

The remaining part of the cost function, or more importantly its gradient, can often not be found analytically. Note that we are dealing with the cross-entropy cost function defined by equation 11, thus involving the full complexity of the neural network. To find the gradient of this expression a monte-carlo sampling technique is used. A first attempt at this is to write

$$\begin{aligned}
\frac{\partial}{\partial\beta}\mathbb{E}_{p\sim Q_\beta}[C(S,p)] &= \frac{\partial}{\partial\beta}\int Q_\beta(p)C(S,p)dp \\
&= \int Q_\beta(p)\frac{\partial \log Q_\beta(p)}{\partial\beta}C(S,p)dp \\
&= \mathbb{E}_{p\sim Q_\beta}\left[\frac{\partial \log Q_\beta(p)}{\partial\beta}C(S,p)\right],
\end{aligned}$$

[12] Although a mixture of two Gaussians for Q_β is shown in [Blu+15] to achieve the highest test accuracies, this requires more parameters for only a slight improvement in accuracy. However, for our purpose of investigating explainability our priority is not to achieve the highest test set accuracy.

and estimate this expression by sampling $p \sim Q_\beta$. However, it is found that this approach leads to a high estimation variance. Although a sample of size m decreases this variance by $\frac{1}{m}$, still an impractically large sample size is necessary to bring this variance at the desired level for approximation.[PBJ12] To overcome this issue a reparameterization trick is proposed.[KW13] The idea is that p is first reparameterized using a random variable $\epsilon \sim \mathcal{N}(0, I)$ as

$$p = \mu + \sigma \odot \epsilon, \tag{35}$$

so that by sampling ϵ we get samples for p that are used to approximate the expectation. Sampling ϵ rather than sampling p directly now allows computation of the derivatives with respect to μ and σ.
More explicitly, letting $q(\epsilon)$ be the density function of ϵ, we first observe from the respective gaussian density functions and equation 35 that, slightly abusing notation, $Q_\beta(p)dp = q(\epsilon)d\epsilon$, which means we can indeed take expectations with respect to ϵ instead of p. It then follows that

$$\begin{aligned} \mathbb{E}_{p\sim Q_\beta}[C(S,p)] &= \mathbb{E}_\epsilon[C(S, p(\mu,\sigma,\epsilon))] \\ &\approx \frac{1}{m}\sum_{i=1}^{m} C(S, p(\mu,\sigma,\epsilon^{(i)})) \\ &\quad \text{with } \epsilon^{(i)} \sim \mathcal{N}(0,I) \text{ for all } i \in [m]. \end{aligned} \tag{36}$$

Combining equations 34 and 36 we conclude that the full cost function can be approximated by

$$\begin{aligned} \mathcal{F}(S,\beta) &\approx \frac{1}{2}\sum_{j=1}^{P}(\mu_j^2 + \sigma_j^2 - \log(\sigma_j^2) - 1) + \frac{1}{m}\sum_{i=1}^{m} C(S, p(\beta,\epsilon^{(i)})) \\ &\quad \text{with } \epsilon^{(i)} \sim \mathcal{N}(0,I) \text{ for all } i \in [m]. \end{aligned}$$

In practice batch gradient descent is used to find the minimum of this function. However, the first part of the cost-function is independent of the data, and does not scale with it. Repeating it for each batch would have the effect of placing too much weight on this term in the training procedure as a whole. In order to use batch gradient descent we thus first divide the cost function into separate parts. Let $B_t \subset S$ be data batches for $t \in [M]$. Now let

$$\mathcal{F}_t(\beta) := \frac{1}{2M}\sum_{j=1}^{P}(\mu_j^2 + \sigma_j^2 - \log(\sigma_j^2) - 1) + \frac{1}{m}\sum_{i=1}^{m} C(B_t, p(\beta,\epsilon^{(i)})),$$

so as to divide the first part of the cost-function equally among the batches. In [Blu+15] it is noted that different distributions of this part over the batches are possible, if one only takes care that the entire portion is accounted for just once per epoch.

For some learning rate $\lambda > 1$ and a β_1 chosen in accordance with the prior distribution[13] the gradient descent iteration is now given by

$$\beta_{t+1} = \beta_t - \lambda \frac{\partial \mathcal{F}_t(\beta_t)}{\partial \beta} \text{ for } t \in [M].$$

Note that for the derivative of the regular cost function the back-propagation algorithm of Section 2.3.2 will again be used. Of course for the training procedure of Bayesian neural networks the same optimization techniques discussed in Section 2.5 apply, and here too the common choice is the Adam optimizer.

To even further decrease the variance of the estimation technique in (36), Wen et al.[Wen+18] propose a method called *Flipout*. They point out that for each draw of $\epsilon \sim \mathcal{N}(0, I)$, all training examples in (some batch of) the data S share this same random perturbation in the computation of the gradient of $C(S, p(\epsilon))$. This induces correlations between gradients of the cost function C, implying that this variance cannot be eliminated by averaging. They propose, given a random draw of "weight perturbations" ϵ, to multiply each coordinate of ϵ randomly by -1 with probability 0.5, *for every single datum* $x \in S$.
They note that although this does not make the weight perturbations independent, it does decorrelate them while not changing the original (symmetric!) distribution from which they are sampled. Consequently, flipout yields an unbiased estimator for the loss gradients that has a lower variance when averaging over a mini-batch.
Wen et al. further show that just explicitly sampling different ϵ for each datum x would come at a much greater computational cost than their flipout method, which can be applied relatively easily by writing computations on an entire mini-batch in terms of matrix multiplications using random sign matrices (i.e., matrices with entries ± 1).

3.2.4 Implementation of Bayesian and Standard Neural Networks

In this section we implement and compare a regular neural network as described in Chapter 2.2 to a Bayesian neural network. As a dataset we use the MNIST [LCB13] dataset of handwritten digits, consisting of 28×28 pixel black and white images. The training set consists of 60000 elements, roughly equally divided among the digit classes 0 till 9, and we test on 10000 images.
We write the code in Python, mainly aided by the packages TensorFlow and TensorFlow Probability. As our purpose is not achieving the highest accuracy, we simply run on a normal (16G RAM) laptop, training for roughly 15 minutes.

[13] The common choice here is a mean zero and identity variance prior distribution for the parameters, as we described earlier on. However the theory of Section 2.3.3 on weight initialization is applicable here as well. Recently this theory has been further developed for Bayesian neural networks in [RMF18]. We will not go into those details here and assume standard normal priors.

Implemented Architectures

The standard neural network architecture starts (as seen from the input layer) with 2 convolutional layers of 100 and 80 kernels, respectively, each followed by an average pooling layer. The convolutional windows have sizes 5×5. After these layers we add two fully connected layers of 240 and 120 nodes. This **standard network has** 1173850 **parameters**.
For the Bayesian neural network we use a similar architecture, only now with 90 and 50 kernels in the convolutional layers, and 180, 120 nodes in the subsequent fully connected layers. This **Bayesian network has** 1157550 **parameters**.
We deliberately aimed for the number of parameters as well as the training times to be approximately equal between the two networks in order to get a fair comparison. Roughly following this restriction, the hyper-parameters for each network have further been tuned using k-fold cross-validation (see Section 2.1.3) on the training data set, with $k = 12$.
For both networks we furthermore used dropout (see Section 2.4.2) on the second convolutional layer and the second fully connected layer, with a fraction of 0.5 and 0.25 nodes removed, respectively.

Bayesian network implementation details

To create the Bayesian neural network we used *flipout layers* in the implementation, that apply the flipout method, described at the end of the previous section, during training. Each forward pass of input data through an implemented Bayesian neural network layer is intrinsically random, due to the random drawing of parameters from the layer parameter distribution. However, when an entire batch of data elements is passed simultaneously through a flipout layer, only a single draw $\epsilon \sim \mathcal{N}(0, I)$ is used as random perturbation, and the parameters are decorrelated by random multiplication of their perturbations by ± 1. For truly independent parameter draws the elements should thus be passed in different batches.
When inputting a single element to the Bayesian neural network using (33) with n network samples, in practice we first create n identical copies of the element, and divide these copies into batches to be fed as input to the model. Creating one big batch of size n to be passed through the network as a whole has speed advantages, although the weights are not truly independent in this set-up, due to the design of the flip-out layers. On the other hand, for truly independent weights n batches of size 1 would have to be passed through the network, but then the computational time reduction trick of the flipout layers is no longer exploited.

Training results

We used the Adam optimizer with learning rate $\lambda = 0.001$ and decay parameters $\gamma_1 = 0.9$ and $\gamma_2 = 0.999$ (see Section 2.5). In coming sections, when referring to the Adam optimizer, these are the default parameter values we use whenever we do not explicitly mention either the learning rate or the decay parameters. We divide the training data in batches of size 256. The accuracy of the standard neural network on the test data is given by **0.9742**. For the accuracy of the Bayesian neural network we note that this depends on the sample size used, see (33). Roughly taking our preferences for accuracy and low output variance as well as computation time into account, we settle for a sample size of 50 (divided into 10 batches of size 5) for most purposes, unless mentioned otherwise. This leads to an accuracy of **0.9675**.

Comparing results on different test cases

We now show some results for specific image examples. We created bar plots showing the probabilities assigned to each digit class by both regular and Bayesian neural networks on some problematic examples, i.e., images for which both networks fail to predict the correct digit. As the Bayesian neural network is in fact an ensemble, given by (33), we also show the standard deviation computed from the sample of 50 networks used in that case. It is represented as an error bar, where the bottom part is truncated so as not to fall below 0. The length of the error bars give an indication of the (expected) deviation from the sample mean experienced for each single draw of the parameters, it does not represent the standard deviation of the mean itself. Assuming 50 fully independent draws, the latter would be given by dividing the error bar lengths by $\sqrt{50}$.

In Figures 11 and 12 we see the standard neural network being overly confident about a wrong decision. The Bayesian neural network does make the same mistakes, but shows more correctly to be uncertain about these predictions.

These images represent a behavior we recognize more generally. For a sample of 1000 test images we find that on average the Bayesian network assigns a probability of 0.55 to the maximum probability class when incorrectly predicting a label, compared to 0.74 for the regular network. Of course one might object that the Bayesian neural network perhaps generally assigns lower maximum probability in the first place. However, we find the difference on correctly predicted labels to be smaller: when correctly predicting a label, the maximum probability class is assigned 0.93 by the Bayesian network and 0.97 by the regular network, on average.

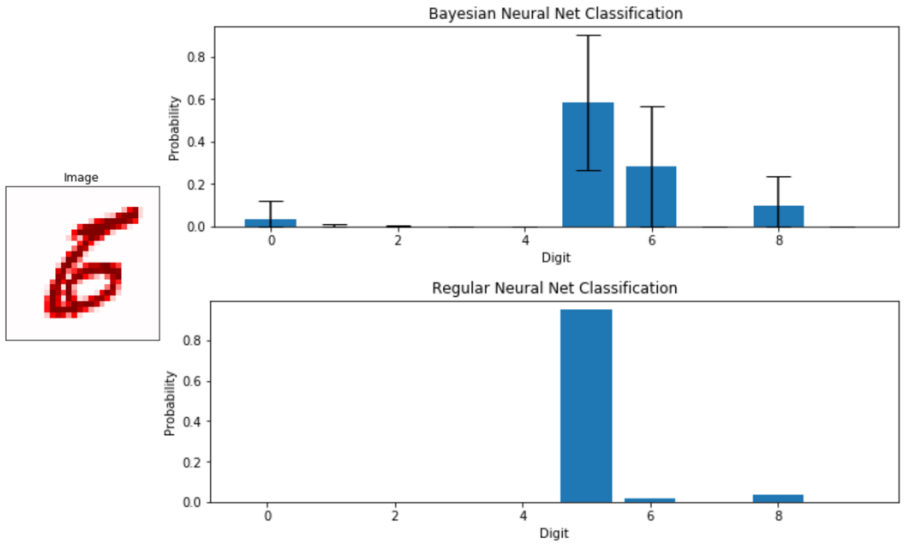


Figure 11: Incorrect prediction of a Bayesian neural network (using 50 samples) and a regular neural network on an element of the test set. The correct digit class is 6 but both networks predict a 5.

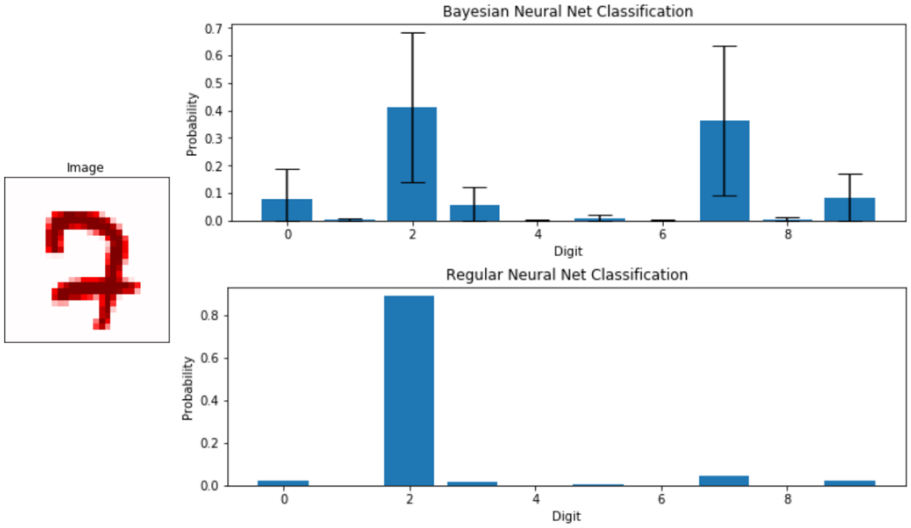


Figure 12: Incorrect prediction of a Bayesian neural network (using 50 samples) and a regular neural network on an element of the test set. The correct digit class is 7 but both networks predict a 2.

Another striking difference is the behavior of these networks on input elements that are very different from the training data. We show the behavior on random noise, the average of the training data, and a randomly created image from a normal distribution with mean and covariance estimated from the training data. In the latter case the values above 1 and below 0 are set at 1 and 0, to stay within the bounds of pixel intensities used for other input images. In each case we find the Bayesian neural network to behave more trustworthy, in the sense that it acknowledges that no single digit class should be assigned a large probability.

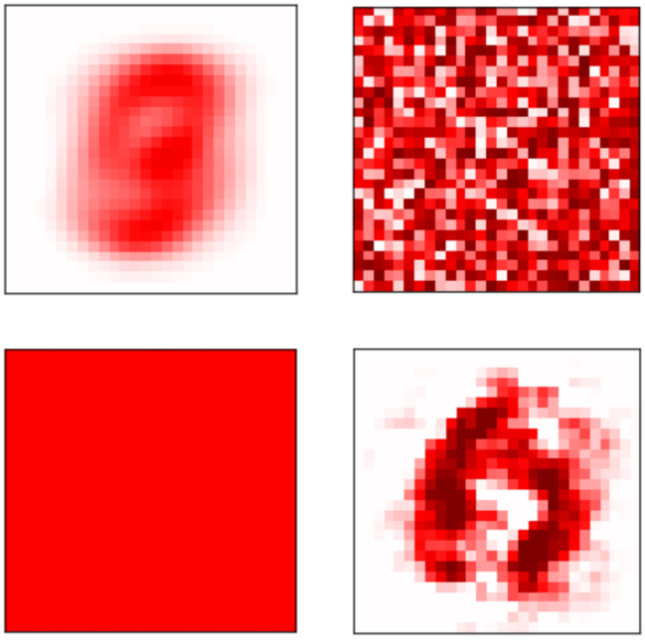


Figure 13: Images to be presented to the networks that is unlike any training data seen before. Top left: **Mean** of the training data. Top right: **Noise** that is independently generated between 0 and 1. Bottom left: **Uniform** value of 0.5. Bottom right: **Random element** generated from a normal distribution using average and covariance estimated from the training data.

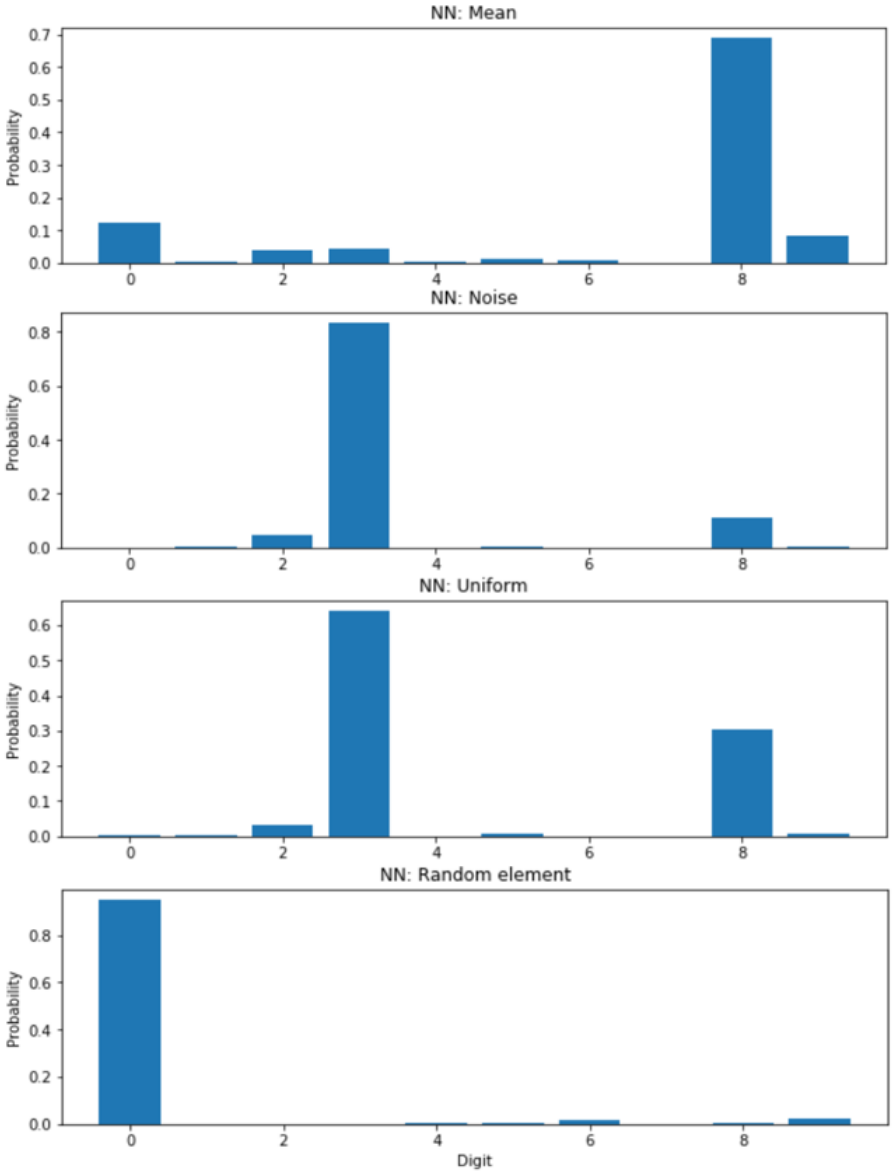


Figure 14: Regular neural network responses to the data inputs of figure 13.

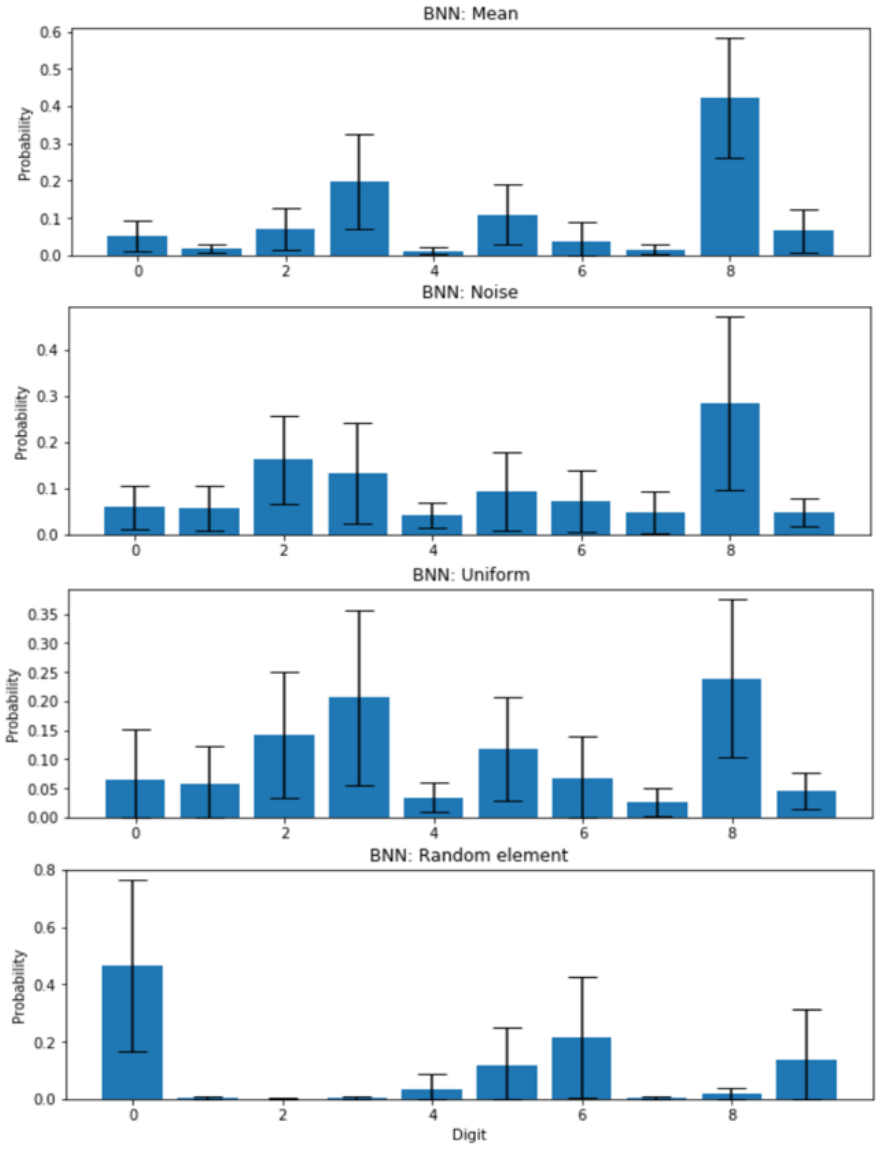


Figure 15: Bayesian neural network responses to the data inputs of Figure 13. Note that in comparison with the regular network outputs, as depicted in Figure 14, we see probability scores that are more spread out. Although the "Noise" and "Uniform" inputs do not result in a uniform distribution among the classes, it is an improvement compared to the regular neural network's behavior. Also on the "Mean" and "Random element" inputs we find a behavior that is more in agreement with our general intuition when seeing these input images.

We find that although the Bayesian neural network does not achieve an equally high accuracy as the regular neural network, it does exhibit more preferred behavior when encountering entirely new data or unclear situations. This suggests it has a better intrinsic understanding of the data in general, making it better suited for explainability tasks. We get back to this point in Section 4.4.3, after having applied a new explainability method on a Bayesian neural network.
For now we note that a proper comparison between different explainability methods requires using the same model, which is why in the coming sections we will apply LIME and RDE to a Bayesian, rather than a regular, neural network. However, the results of this chapter alone may be sufficient reason to stick with the Bayesian neural network anyway, when working on explainability problems.

3.3 LIME

Currently one of the most popular approaches to achieve explainability of black box models is the LIME algorithm, short for Local Interpretable Model-Agnostic Explanations[RSG16]. As the name suggests this method is suitable for any model, and delivers local explainability, i.e., explaining the reason for $f(x)$ *given* an input element x.
The method is based on approximating the true (black box) model f by some interpretable model g in the vicinity of the input x for which an explanation is required. This is done by minimizing the weighted difference between f and g with added complexity constraints on g, where weights are determined by the distance from x: the closer to x, the more important the difference becomes. We give a more detailed description of how the method works in the following introductory section. Although the authors first briefly present their method generally, i.e., they acknowledge that different families of functions can be chosen to supply the approximator g, and that the difference between f and g can be defined in different ways, we present the LIME method by following the same implementation decisions as in the paper. After describing the method we go over the implementation and results, discussing the quality and usefulness of the explanations offered as well as the drawbacks of this method.

3.3.1 Introducing the method

We again let $f : X \to Y$ be a neural network as defined in Section 2.2 and let $x \in X \subset \mathbb{R}^d$ be an element for which we wish to explain its outcome $f(x) \in Y$. An essential characteristic of the LIME method is the reduction of the feature space $\mathbb{R}^d$ to an *interpretable representation* space $\mathbb{R}^{\tilde{d}}$ with $\tilde{d} < d$. The authors justify this by stating that the explanation needs to be understandable for humans, thus a less complex model is necessary to begin with. In other words: explaining a model based on a too large number of actual (sometimes by itself incomprehensible) features is destined to fail from the start. The idea is thus to create a model $g : \mathbb{R}^{\tilde{d}} \to Y$ that outputs values that are close to the true outputs given by f in the neighborhood of x. To properly fit g to f, their difference on a set of sampled elements in the vicinity of x is taken.

As our implementation will again be based on the MNIST data set, we describe the LIME method for image recognition. Here the interpretable representation space is defined by dividing x into predetermined contiguous patches of pixels, i.e., *super-pixels*, see Figure 16. By optionally graying out some of these super-pixels, elements of the representation space are given as binary vectors indicating the presence or absence of a super-pixel. To divide the input image x into super-pixels, the Quickshift algorithm [VS08] is used, but many different image segmentation algorithms exist for this task, and we do not go into the details of these algorithms.

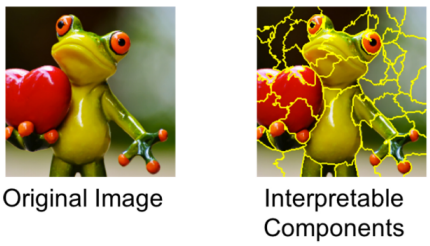


Figure 16: A treefrog image seperated into super-pixels.

With $\tilde{d}$ the chosen number of super-pixels recognized in the image x, we now let $\tilde{x} \in \{0,1\}^{\tilde{d}}$ represent an element of the interpretable representation space. Here 1 indicates the presence of an original super-pixel and 0 indicates a grayed out super-pixel, so that x itself would correspond to a full vector of ones in this interpretable representation space.
Using this representation we generate a dataset of perturbed samples based on x, denoted by $\tilde{S} = \tilde{S}(x)$, that will be used to fit g to f on. Elements $\tilde{z}$ of $\tilde{S}$ are generated by randomly choosing a number of indices $i \in [\tilde{d}]$ for which we set $\tilde{z}_i = 1$, with the rest getting value 0. Here both the random selection of indices as well as the number of indices is chosen uniformly at random. As such, elements of $\tilde{S}$ are derived from the true image x, but have randomly grayed out super-pixels, as can be seen in Figure 17.

Figure 17: A selection of elements of $\tilde{S}(x)$, where x refers to the image of the treefrog in Figure 16.

To replace removed super-pixels any other color than gray can also be used, considering what approach is most reasonable in the context. The super-pixel can also be replaced by the average value among all pixels in it. We use the black background color observed in the MNIST data set to replace super-pixels that are removed. Colors are commonly implemented as 3-dimensional vectors of integer elements between 0 and 255, possibly normalized. Black, being represented as $\mathbf{0}_3$, is then a natural choice to represent the absence of input signal.

We now define a function $g_\beta : \{0,1\}^{\tilde{d}} \to \mathbb{R}$ to represent an interpretable model. With $\beta \in \mathbb{R}^{\tilde{d}}$, and $\beta_0 \in \mathbb{R}$ an intercept we let

$$g_\beta(\tilde{z}) := \beta_0 + \beta \cdot \tilde{z}.$$

The goal is to choose parameters β that make g closely fit the true (inexplicable) model f in the neighborhood of x. The intercept β_0 will also be fitted, but we leave this implicit in this analysis as our interest lies mainly in the coefficients of the super-pixel variables. (One can from now on also consider β to have an extra dimension indexed by 0.) We assume the case of a classifier $f : X \to [0,1]^N$, and focus only on the output of f that represents the discovered class of x. For the example of the treefrog in Figures 16 and 17 this means we only care to approximate g to the output coordinate of f corresponding to the probability of a treefrog. We denote this coordinate by $i_x := \arg\max_i(f_i(x))$.
Let for a given $\tilde{z} \in \{0,1\}^{\tilde{d}}$ its representation in the *original* feature space be denoted by z. To fit g to f we use the locally weighted square loss, with the weights determined by a gaussian with width σ, i.e.,

$$\beta^*(x) = \arg\min_\beta \sum_{\tilde{z} \in \tilde{S}(x)} (f_{i_x}(z) - g_\beta(\tilde{z}))^2 e^{-\frac{D(x,z)^2}{\sigma}}, \tag{37}$$

where we use as distance metric D the cosine distance

$$D(x,z) = 1 - \frac{\langle x, y \rangle}{||x|| \cdot ||z||}.$$

The above expression can then be directly solved using weighted least squares. However, in the source code of the method the parameters β are slightly regularized, and the expression is actually solved using weighted ridge regression, see [Hol73], with regularization constant 1.
The actual explanation can now be derived from β^*. For super-pixel $j \in [\tilde{d}]$ the definition of g shows that removing it from the picture would contribute $-\beta_j^*$ to the (highest) outputted probability of label i_x. In this sense we observe a type of feature importance from β^*, based on features that make up the interpretable representation space $\mathbb{R}^{\tilde{d}}$. Simply put, if a certain super-pixel's presence is important for the model's outcome, it has a high corresponding value in β^*. Negative values may also occur, signaling that the super-pixel actually has a negative effect on the outputted probability of the label under consideration: the label has been outputted *despite* the presence of that super-pixel.

3.3.2 Implementation and results

We implement the Bayesian neural network as described in Section 3.2.4 on the MNIST data set, using a sample size of 50 networks for the actual model. We use the Quickshift algorithm to divide the input images into super-pixels [14] creating approximately $\tilde{d} \approx 13 \pm 2$ super-pixels in a given image, the precise number depending on the image. The regression weights in equation 37 are determined by letting $\sigma = 0.25$, following the default settings of the method. Finally we let $|\tilde{S}| = 500$, in other words we generate 500 samples that are used to fit the linear interpretable model. Using these settings it takes a few minutes to generate an explanation for a given image.
We end this section with a selection of results of the LIME method on MNIST images, and briefly discuss their effectiveness on explainability.

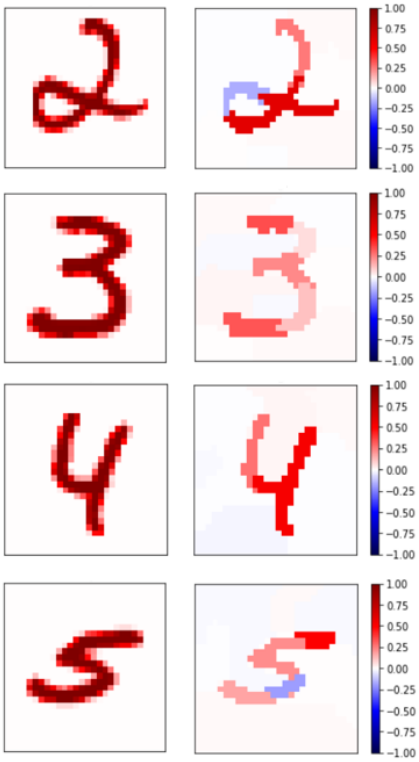


Figure 18: Results of the LIME method on (arbitrarily chosen) digit labels 2-5. Original images are depicted on the left, LIME explanations on the right. The super-pixels are shown as areas of similar color, where the color represents the feature importance according to the mapping shown on the right of the image.

[14] The settings used are a kernel size of 1, maximum distance set to 4 and ratio to 0.1. (See [VS08] for details on the algorithm and these settings.) We decided on these settings manually, by observing results for different images and taking computation time into account.

We notice that the LIME method is excellent in finding the relevant information used in the 28×28 grid, namely the actual digit. This might seem redundant at first, as the explanation looks almost identical to the original input, but it is important to realize that the algorithm receives 784 abstract input dimensions, and LIME is able to filter out a small important subset of those, assigning relevant importance.
Of course the MNIST data-set with its black uninformative background does not reveal all the benefits of the method. In a widely known example of the importance of AI explainability methods, LIME is shown to detect a husky wrongly being labeled as a wolf because the algorithm is focusing on the snowy background.

(a) Husky classified as wolf (b) Explanation

Figure 19: LIME algorithm showing how a false classification of a husky as a wolf is caused by the snowy background, thus detecting a flaw in how the trained model operates.

Looking back at the results of Figure 18, clearly LIME is successful in detecting the important features that contribute most to the output. However, we believe that simply highlighting these features, although useful in practice, does leave part of the actual "explanation" to the imagination.
The proxy measure of feature importance is different from a clearly defined explanation, in the sense that the *reason* for an image to be labeled as it is, does not become directly clear. We can see what pixels are part of a specific digit, or a husky or a wolf (if the model is correctly trained), but have no understanding as to how these pixels lead to that label. The more pronounced super-pixels do indicate that their removal would have a larger local effect, but even if the super-pixels making up the digit would be more detailed and more conveniently placed (more on this later) this does not produce an actual explanation.
A reasonable objection to the above approach is that the background color (black) is identical to the color used for removing super-pixels in the original image. Although these are the standard settings of the algorithm, which is understandable for image data, this does result in a lack of importance measuring of black parts of the input images for the MNIST data set. For this reason we did decide to experiment with coloring removed super-pixels gray instead of

black, which effectively allows the possibility of attributing importance to the *absence* of input signal at certain locations, rather than solely focusing on the importance of non-zero input signal pixels. However, the method turns out be arguably less effective for these settings, as can be seen in Figure 20.

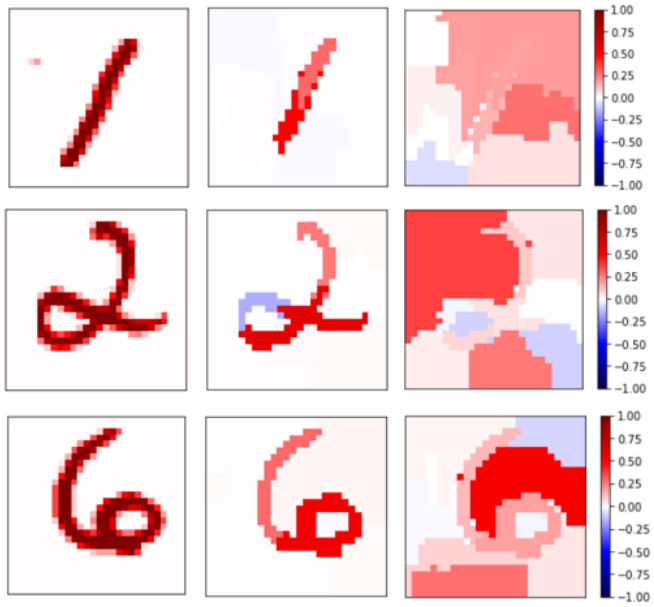


Figure 20: Results of the LIME method on digit labels 1, 2 and 6. From left to right: original images, LIME explanations using erase color black, LIME explanations using erase color gray.

The effectiveness of the method clearly depends on the specific configuration of super-pixels. After some experimentation, we conclude that it is difficult to consistently achieve a good configuration. Of course one can decide on the super-pixels manually, prior to running the LIME method. Although we acknowledge that this could give better results, we have chosen not to follow through with this approach. This is because we believe we must be careful not to use too much prior knowledge on what a good explanation resembles. For instance, we may assume the absence of light pixels in the upper-right part of the digit 6 distinguishes it from an 8, but selecting super-pixels to specifically show this to be true is an approach more related to hypothesis-testing than to truly explaining outputs.
Perhaps more importantly, real situations where model explainability is required might not present themselves with clear strategies on how to fine-tune the method's parameters specifically for each input element. The point is that we do *not* yet fully understand this input element prior to running the explainability method! This further motivates us to be careful and not optimize too much for creating the best possible images.

3.4 RDE

We now turn our attention to a more recent method for explaining neural network decisions, called RDE (Rate Distortion Explanation) [Mac+19]. In contrast with LIME, this method is not model-agnostic, as the algebra is specifically derived and optimized for the neural network case. However, it is also a local method, thus explaining $f(x)$ for a given x.
The main idea of the method is to measure feature relevance based on the predictor's diminished performance when replacing selected features with noise. If, for a given input element x, some features x_i, for $i \in [d]$, can safely be replaced by noisy input (i.e., without affecting the model's ability to predict the correct label) these features can be considered less relevant.
The authors first formulate the binary problem of dividing features into two sets: relevant and irrelevant. They show that the computational complexity of finding such sets of relevant features makes the method infeasible in practice, and thus propose a continuous problem relaxation with a heuristic solution strategy.

In the next section we describe their approach for the continuous problem of assigning relevance scores to features. In the sections after that we treat the implementation and discuss the results.

3.4.1 Introducing the method

Let $f : X \to Y$ be the neural network defined as usual with $X \subset \mathbb{R}^d$ and $Y \subset [0,1]^N$. Let for a given $x \in X$ the relevance score for its d features be encoded by a vector $s \in [0,1]^d$, with higher values representing more relevant features. Let n be some noise vector, randomly distributed according to a gaussian distribution based on mean and covariance estimated from the training data. Now define

$$a(x, n, s) := x \odot s + n \odot (1 - s),$$

where, ideally, the least important features, based on their score s, are replaced most by noise n, whereas important features with coordinates of s close to 1 are barely changed. To measure the correctness of a given choice of s we define the **expected distortion** as

$$D_j(s) := \mathbb{E}_n[\frac{1}{2}(f_j(x) - f_j(a(x, n, s)))^2], \tag{38}$$

where j is the class of interest, which can optionally be given by $\arg\max_i f_i(x)$, i.e., the relevant index based on the maximal probability label of x.
If the feature importance is indeed in line with the elements of s, we would expect a low distortion. Of course there is the trivial case of $D(1) = 0$, where we have assigned every feature an importance score of 1. However, the challenge becomes to simultaneously find a small vector s and a small distortion $D(s)$.

Using a regularization parameter λ for the size of s, the problem is now formulated as finding

$$s^* = \arg\min_{s \in [0,1]^d} D_j(s) + \lambda ||s||_1. \tag{39}$$

To efficiently approximate the above expression we first write the expected distortion's composition into bias and variance as

$$\begin{aligned} D_j(s) &= \mathbb{E}_a \left[\frac{1}{2} (f_j(x) - \mathbb{E}_a[f_j(a)] + \mathbb{E}_a[f_j(a)] - f_j(a))^2 \right] \\ &= \frac{1}{2} (f_j(x) - \mathbb{E}_a[f_j(a)])^2 + \frac{1}{2} \mathbb{V}_a[f_j(a)]. \end{aligned} \tag{40}$$

We now focus on the unknowns of the above expression, namely $\mathbb{E}_a[f(a)]$ and $\mathbb{V}_a[f(a)]$, discarding from now on the dependence on the j-th index. To find these unknowns the layered structure of f is used to propagate the distribution of neuron activations through the network. Note that from here on this method becomes model-specific, whereas the theory until (39) can be more generally applied.

To propagate the distribution of activations through the network we use an approximate method, called assumed density filtering. At each layer we assume a Gaussian distribution for the input, transform it according to the layer's functional form and project the output back to the nearest Gaussian distribution using KL-divergence. It is shown in [Min01] that for a fully connected layer with ReLU activation function, it suffices to match the first two moments of the distribution. In other words, assuming that the layer inputs (i.e., the activations of the previous layer or the inputs of the network) are normally distributed, we only need to compute the mean and variance of the layer outputs. The closest gaussian distribution, based on KL-divergence, can then simply be defined using this mean and variance.

For a given input distribution $\mathcal{N}(\mu_l, v_l)$ of the l-th layer, the task is now to find the output distribution's mean and variance: μ_{l+1} and v_{l+1}. These will be used to define the closest Gaussian to the output distribution, and so on, until the first two moments of $f(a)$ are found, which can be plugged into (40). The first and second moments of the input layer's distribution are readily given:

$$\begin{aligned} \mu_1 &:= \mathbb{E}[a] = x \odot s + \mathbb{E}[n] \odot (1 - s) \\ v_1 &:= \mathbb{V}[a] = \text{diag}(1 - s) \mathbb{V}[n] \text{diag}(1 - s). \end{aligned} \tag{41}$$

We use corollary 2.8.1 to redefine the network as a set of fully connected layers and for simplicity we assume that the network activations within each layer are uncorrelated.[15] This means we can focus on the diagonal terms of covariance matrices only. For some given input a, a weight matrix W, bias b and a ReLU activation function σ, we first note that

$$\begin{aligned} \mathbb{E}[Wa+b] &= W\mathbb{E}[a]+b, \text{ and} \\ \mathbb{V}[Wa+b] &= W\mathbb{V}[a]W^T \end{aligned} \tag{42}$$

Focusing only on the diagonal, using the notation $\mathbb{V}_{\text{diag}}$ for the vector of diagonal terms of a covariance matrix, we further find that

$$\mathbb{V}_{\text{diag}}[Wa+b] = (W \odot W)\mathbb{V}_{\text{diag}}[a]. \tag{43}$$

To deal with the ReLU activation we use the following result, shown in [FH99]:

Lemma 3.1 *Let z be distributed normally with mean μ and variance v. Then for a ReLU activation function σ the mean and variance of $\sigma(z)$ are given by*

$$\begin{aligned} \mu_{relu}(\mu, v) &= \mu\Phi(\mu/\sqrt{v}) + \sqrt{v}\phi(\mu/\sqrt{v}), \\ v_{relu}(\mu, v) &= (\mu+v)\Phi(\mu/v) + \mu\sqrt{v}\phi(\mu/\sqrt{v}) - \mu^2_{relu}(\mu, v), \end{aligned}$$

where Φ and ϕ are the standard normal cumulative distribution function and normal density function, respectively.

We can apply the lemma to the results of equation 42 and 43, noting that the activations are still normally distributed after the affine layer transformation. We slightly abuse notation, firstly by letting v_l refer to the vector of diagonal terms of the variance of the l-th layer input, rather than the full covariance matrix (as it is assumed diagonal anyways), and secondly by implicitly letting square root, division, and normal distribution functions be applied element-wise. For clarity we omit some of the layer dependency and let $W = W_l$ and $b = b_l$ in the following. We find that we can propagate the moments of the distribution by the following iteration:

$$\begin{aligned} \mu_{l+1} &= (W\mu_l + b)\Phi\left(\frac{W\mu_l+b}{\sqrt{W \odot W v_l}}\right) + \sqrt{W \odot W v_l}\,\phi\left(\frac{W\mu_l+b}{\sqrt{W \odot W v_l}}\right), \\ v_{l+1} &= (W\mu_l + b + W \odot W v_l)\Phi\left(\frac{W\mu_l+b}{\sqrt{W \odot W v_l}}\right) \\ &\quad + (W\mu_l+b)\sqrt{W \odot W v_l}\,\phi\left(\frac{W\mu_l+b}{\sqrt{W \odot W v_l}}\right) - \mu^2_{l+1}. \end{aligned}$$

[15] This assumption is not based on the premise that this is indeed approximately the case, as it is highly likely that correlation occurs in the network activations. However, a better approximation technique used by the authors leads to comparable results when it comes down to explainability and successfully determining feature relevance. Therefore we chose the computationally simpler approach of only looking at the diagonal terms of the propagated distributions, which comes down to assuming uncorrelated activations.

Starting with (the diagonal elements of) (41), this iteration eventually gives the unknown last layer moments required in (40).
Having found the proper differentiable functional form of the distortion it is now possible to use gradient descent optimizers to tackle equation (39).

3.4.2 Implementation and results

We implement the RDE method on the MNIST dataset using a Bayesian neural network by replacing the layer parameters of the previous paragraph by stochastic variables. These are drawn from distributions that are in turn based on the Bayesian neural network distribution parameters. For computational efficiency we only take one draw in each step of the gradient descent, in effect thus using a Bayesian network with sample size 1 (as defined in (33)). The actual parameter randomness obtained from using a Bayesian network rather than a normal network is then accounted for by the many steps of the optimization procedure, as each step uses a different network. Note that this approach works similarly to stochastic gradient descent for training neural networks, where every step of the iteration is based on different data elements. Also, optimization using different parameters in each iteration is applied similarly in the training of the Bayesian neural network itself.

For the actual stochastic gradient descent optimization in this case we use the Adam optimizer with learning rate $\lambda = 0.01$, and decay parameters as before. We use projected gradient descent, meaning that the variable s is projected onto $[0, 1]^d$ after each update, as it is restricted by this subset.
We set the regularization parameter to $\lambda = 0.3$ and start the optimization with $s = 0.2 \cdot \mathbf{1}_d$, following the approach of [Mac+19]. We show a single example on each of the 10 digits, depicting the resulting s^* from (39) for each case.

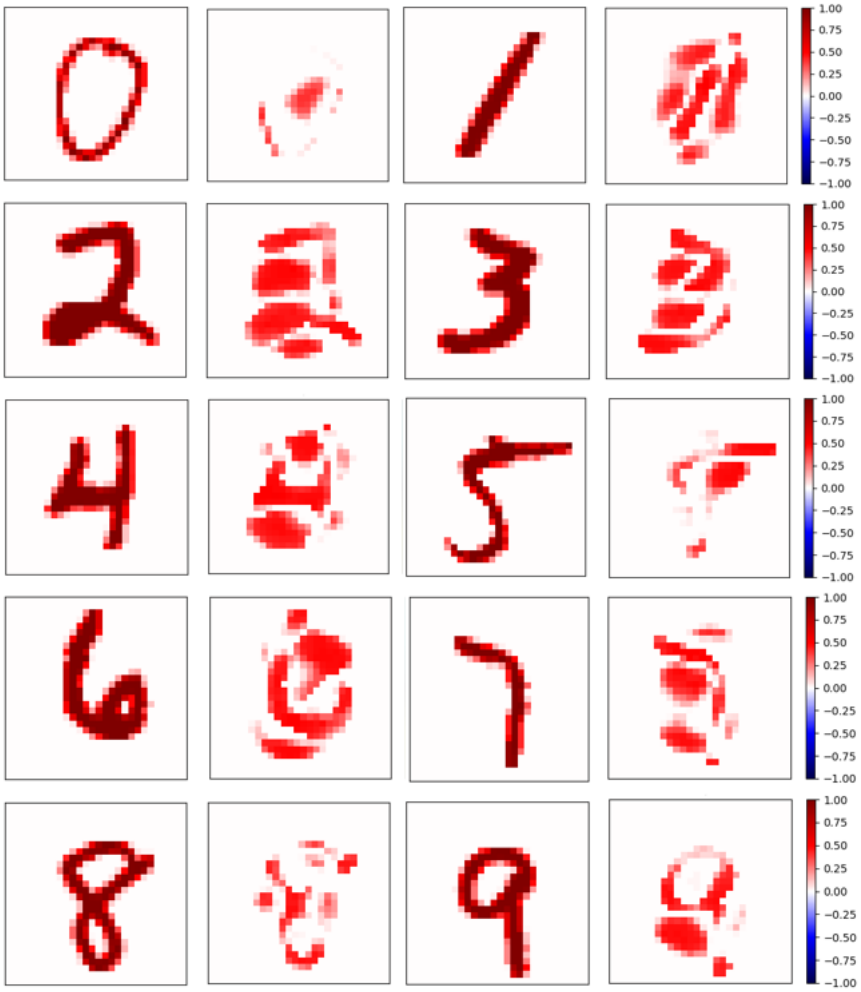


Figure 21: Resulting importance vector s after optimization, shown for a single case of each digit class. The original images are on the left of the pairs.

The images clearly show feature importance in a more intuitive way than in the LIME method results. Rather than simply showing the pixels that are part of the digit, we more generally see the importance attributed to different parts of the input images. In contrast with the LIME method, there is also no need to divide the input into (understandable) super-pixels prior to running.
Although the results are arguably more interesting, we are still only witnessing feature importance. More explicitly the images show which pixels are most sensitive to replacement by noise. The concept of what an actual explanation is will be a point of discussion in the next chapter.

4 Explanatory Vector Decomposition

We propose the *EVD-method*, short for *Explanatory Vector Decomposition*. As briefly hinted in Section 3.1.3, the method originates from the discussion on what an actual explanation is and the ambition to find a simple mathematical definition for this concept. In short we define an explanation as a movement in input space, for which we compute its *explanation strength* as impact (difference in output probability) divided by vector length. After deciding on a "neutral" base point in the input space, we aim to decompose the vector from this base point towards the input element under consideration into an explanatory part and a general part.

4.1 Data Preparation

It is always good practice in machine learning to prepare data before supplying it to the algorithm. The MNIST data set has for instance been arranged to only have features in $[0, 1]$, and does not contain any (relative) outliers. Of course, further preparations like centering and scaling the data are in principle unnecessary, as the first layer of a neural network can also take care of this. Indeed, we have shown that the current form of the MNIST data set is good enough for the training of high-accuracy (Bayesian) neural nets, as well as the application of the LIME and RDE explainability methods.
However, the EVD-method will make such significant use of the input space of the model that we decide to more explicitly propose a proper data preparation procedure to optimize its performance. As we will see, the proposed data preparations in this section substantiate and justify the use of some definitions in the next sections. Given a training data set $S_X \subset \mathbb{R}^d$ we put forward the following step-wise procedure.

1. First, any features that are constant for the full training data set are removed. Although this may be unnecessary with regards to algorithmic behavior, not using these features, and explicitly acknowledging this, does make the explanation method more honest.

2. Although this already has been taken care of in the MNIST data set, for completeness we do mention how we would deal with outliers in the data. For each dimension separately we bring the extreme values back to a high percentile value of the data in that dimension, e.g., the 99.9th percentile, by straightforward cutting off of values, or possibly by applying log-scaling, or both. This ensures that the extreme values are still noticed, but do not disrupt the mean and other statistics computed on the data.

3. The next step is to normalize the data. We compute the mean of the training data S_X and subtract this value from each element. For further reference we denote this mean as x_{mean}.

4. If the variance of some input feature is very low, we generally expect little movement along this dimension. This not only holds for elements $x \in S_X$ themselves, but preferably also for any type of vectors and movements in input space that we encounter in the EVD-method.
   A first option for controlling this is to divide each dimension by its estimated standard deviation. However, the more general approach, also known as principal component analysis, includes covariances between features in the procedure. The full transformation we use, derived in appendix C, is for each $x \in S_X$ to apply

$$x \to T(x - x_{\text{mean}}) = D^{-\frac{1}{2}} P^T (x - x_{\text{mean}}). \tag{44}$$

   where P is the matrix with columns given by orthonormal eigenvectors of the covariance matrix estimated from S_X, and D the diagonal matrix with the corresponding eigenvalues.

5. Although dividing by (estimated) standard deviations is a simple way to normalize a (centered) normally distributed variable, the features of x, or Tx when changing systems, need not be normally distributed. In fact, we are more inclined to believe the "true" distribution has fat tails, assuming that more extreme values than those that have been observed in S_X are likely to occur when experimenting with diverse movements in the input space X.
   If the estimated variance for a dimension i is very small, with $D_{ii} \approx 0$, the transformation becomes extremely sensitive to changes in this i-th dimension. In practice we can for example think of the pixels at the edge of the images, which may have values close to 0 for the entire data set. While we do deliberately choose T to be more sensitive to changes in these pixels, it becomes impractical when the division by values close to 0 for some dimensions suddenly leads to extreme values, overruling all the information in other dimensions.
   However, noting that the data has been normalized, if a dimension i has an impractically small variance D_{ii}, it must be the case that $P^T(x - x_{\text{mean}})$ is always close to 0 in this i-th dimension. Assuming that this value is so close to zero that the model f cannot possibly rely on it for its classification, we can safely remove the dimension altogether, and change D and P accordingly.

In what follows we will use the notation $x \in X_T \subset \mathbb{R}^{d_T}$ when referring to an element of the input space S_X that has been transformed by subtracting x_{mean} and applying (44), possibly with $d_T < d$. We also write $v \in X_T$ for more general vectors v that are represented in the basis formed by the d_T columns of P, and scaled by inverse standard deviations.
When visualizing vectors $v \in X_T$, we must first apply the inverse transformation T^{-1}, thus actually depicting $T^{-1}v$. To similarly invert the procedure when visualizing static points $x \in X_T$, we also add x_{mean} back for interpretability.
Assuming the dimensionality reduction causes negligible information loss, we can safely apply the transformation $T^{-1}Tx = x$ to all data before training.

4.2 Introducing the method

As discussed in Section 3.1.3, many methods for explaining neural network decisions do not produce an explanation directly, but rather return proxy measures as tools that aid the user in coming up with the actual explanation by him or herself. Comparing different methods in this field of research is notoriously difficult, as a method's quality is not directly based on its output, but on the more subjective notion of how effectively this output can be used to form intuitive explanations.
In the LIME and the RDE method a type of feature importance is returned, from which the user can conclude what the reason, the actual explanation, for an output might have likely been.
In some cases, like in the LIME-method for image recognition applied on the MNIST data-set, results may not be useful. Signalling which pixels from the input image form the actual digit that is classified does not give any insight into *why* it was classified as a certain label. Improving results by manually dividing the input into interpretable components also has the disadvantage that prior understanding of the input (image) is necessary, which in practice might not always be available.
In the RDE method the results are considerably more interesting for formulating explanations. Knowledge on which parts of input images can, or can not, be safely replaced by noise without affecting the algorithm output is shown to be a useful measure of feature importance: one can intuitively form explanations based on the method's output. However, this method too does not directly produce an explanation, but relies on the anticipation that the feature importance returned is sufficiently insightful that an explanation can be formed afterwards that is subjectively based on it.
We propose to firstly search for an actual explanation, which we seek to define more precisely in the next section, before achieving practical use by potentially adding interpretation constraints.

4.2.1 Defining an Explanation

Rather than plainly stating the definition first, we start this section by going over the thought process that led to its final form. Specific implementation details are discussed in the sections after this one, making the definition proposed here a general starting point for further ideas and research.
Let $X_T \subset \mathbb{R}^{d_T}$, $Y \subset [0,1]^N$ be the transformed input space and the output space, and let a classifier be given by $f : X_T \to Y$. In the implementation we use a Bayesian neural net for f, but the method is model-agnostic, so in principle any classifier can be used. We discuss the reason for using a Bayesian neural network and more generally the impact of model selection in Section 4.4.3.
Given an input element $x \in X_T$, let $i = \arg\max_j f_j(x)$ be the class with the highest output, i.e., the predicted label. We state that the explainability goal first comes down to explaining why $f_i(x)$ is significantly higher than $1/N$. (Section 4.4.2 treats other approaches and more creative applications.)

To explain $f_i(x)$ we take the network as given, and seek for a reason solely in its input x. For the concept of explainability we require some form a causality.[16] We achieve this by viewing x not as a static point in input space, but rather as the result of a movement through input space, so that we can attribute the cause of $f(x)$ to this movement.
Under this perspective, we need to carefully define where it is that x has moved from. In other words, interpreting x explicitly as a vector requires some intuitively chosen origin, not necessarily equal to the standard origin $\mathbf{0}_{d_T}$ in $\mathbb{R}^{d_T}$. We denote this artificial origin as $\bar{x}$. The interpretation of this point is that it must represent some "neutral" location in X_T. It must firstly be neutral in the sense that it lies relatively centered in the training data set. Preferably, $\bar{x}$ is also neutral in the sense that $f(\bar{x})$ has a low value in each dimension, not clearly favoring any of the output labels. We will for now omit discussions on how to exactly interpret these desired attributes of $\bar{x}$, and how to achieve them, getting back to this topic in Section 4.2.2.

Viewing the model input x now as a displacement rather than as a static point, i.e., $x = \bar{x} + (x - \bar{x})$, we start by proposing the following:

Definition 4.1 *An* explanation *is a movement in input space. We denote the explanation for class i on input x by $e_i[x]$. It can be interpreted after representing it in the original space X by $T^{-1}e_i[x]$.*

In this sense we might initially state that the *reason* for $f_i(x)$ is that the neutral input $\bar{x}$ has been shifted by $(x - \bar{x})$. Although arguably true, this of course is no practical find, as can be seen in Figure 22.
The vector $x - \bar{x}$ is not interesting in its entirety: decomposing it into smaller vectors we find that parts of it might be general, not implying any label in specific, whereas other parts may be highly relevant and unique for the label i. We thus propose to separate $x - \bar{x}$ into three consecutive vectors. First we extract a *general* vector $g_i[x] \in X_T$, contributing little information or conclusive evidence towards $f_i(x)$. From the point $\bar{x} + g_i[x]$ in X_T we now add the *explanatory* part $e_i[x]$, which contains the actual reason behind the high value $f_i(x)$. Finally $d_i[x]$ adds the *details* to arrive at x. We thus have

$$x = \bar{x} + g_i[x] + e_i[x] + d_i[x], \tag{45}$$

which is also depicted in Figure 23.

[16] In other explainability methods that output proxy measures like feature importance, it is exactly this causality that is subtly missing, left for the user to manually define: feature importance only leads to explainability when one postulates on the absence of the most important features. Sensitivity analysis leads to explainability only through imagining changing the most sensitive inputs. Although these steps are sometimes minor, they are manual and possibly subjective. We want the causality to be embedded as much as possible in the method itself, requiring no extra reasoning from the user.

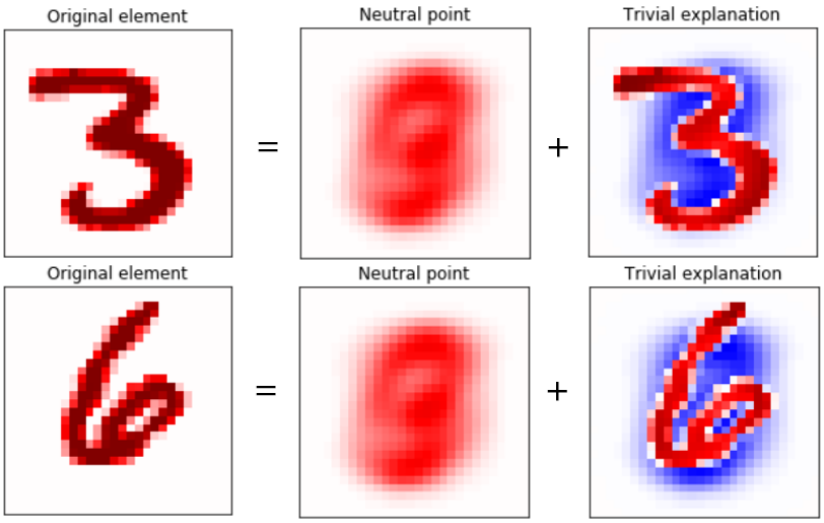


Figure 22: We can for example choose the mean as a neutral point. (This comes down to letting $\bar{x} = \mathbf{0}_{d_T}$, as the training data in X_T is normalized.) We show the trivial explanation vector $e_i[x] = (x - \bar{x})$ for two elements $x \in X_T$ of the training data. We transform all vectors back to the original space, and also add the original data mean back to x and $\bar{x}$ for a clearer image. More explicitly, the Figure represents $[T^{-1}x + x_{\text{mean}}] = [T^{-1}\bar{x} + x_{\text{mean}}] + [T^{-1}e_i[x]]$. The coloring is as before, with blue representing negative values and red positive values, all within $[-1, 1]$.

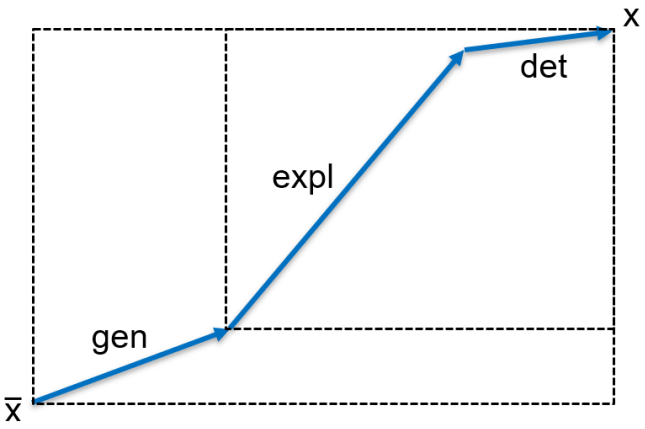


Figure 23: Two-dimensional representation of a decomposition of the vector $x - \bar{x}$ into three components: a general component followed by an explanatory component and lastly the addition of details.

Note that the information in $g_i[x]$ may be general, but is not necessarily unimportant altogether. In fact due to the highly non-linear nature of the model, the remaining information $e_i[x] + d_i[x]$ could be meaningless by itself: neither $f_i(\bar{x} + e_i[x] + d_i[x])$ nor $f_i(e_i[x] + d_i[x])$ is meant to be significantly larger than $1/N$. Practically speaking it is important to distinguish vectors representing specific movements from vectors representing static points in input space. The movement $e_i[x]$ only has meaningful value as an explanatory vector when placed after a *given* general part $\bar{x} + g_i[x]$. We therefore speak of the *conditional explanation* $e_i[x]$, to emphasis the dependency of this vector's usefulness on the prior vectors $\bar{x}$ and $g_i[x]$.

To achieve a sensible decomposition as described above, we need a way to distinguish explanatory and general movements. For this purpose we propose the concept of *explanation strength* of a vector: it is the *vector impact*, i.e., the difference in model output probability before and after adding the vector, divided by its length.

Definition 4.2 *Let $f : X_T \to [0,1]^N$ and $i \in [N]$ a chosen label class of interest. Let $a, b \in X_T$ and let v be the displacement $b - a$. The **explanation strength** of v for the class i is given by*

$$S(v) := S(a,b) := \frac{f_i(b) - f_i(a)}{||b - a||}. \tag{46}$$

We aim for the explanation strength of $e_i[x]$ to be as high as possible, attributing a large amount of information to a small movement in input space. It is this concept of maximizing explanation strength that ensures we do not simply conclude that $x - \bar{x}$ is the explanation for $f_i(x)$: although the impact is high, so too is the vector length, leading to a rather weak explanation.

Of course this reasoning depends on the specific metric that is chosen for computing the distance between a and b, and the use of the standard Euclidean norm might seem arbitrary. However, it is here that the data preparation discussed in Section 4.1 plays an important role. Because of the transformation T in equation 44, we can actually represent the explanation length as

$$\|b - a\| = \|TT^{-1}(b - a))\| = \|D^{-\frac{1}{2}} P^T T^{-1}(b - a)\|. \tag{47}$$

This means that the explanation length in fact takes into account how similar $T^{-1}e_i[x]$ is to elements $x \in X$. Movements along dimensions of X for which a low variance was estimated, will contribute relatively more to the computed vector length, with the opposite being true for high estimated variances. Or if two features are highly correlated in X, nearly always being equal, it will have a relatively large impact on the explanation vector length if $T^{-1}e_i[x]$ attributes very different values to these features.

By aiming for an explanation vector with a small length, this definition then directs an explanation to make intuitive sense: it is peculiar to explain a labeling of x using arguments that require very rare extreme movements in input space. Note that even though extreme movements in x may seem interesting, it is not a given that the model agrees with this. Of course when such movements have large impact on the output we *do* want to include them, but this is taken care of by the numerator in the explanation strength. If equal impact on the model output is achieved by a less extreme movement along some other dimension, we would prefer to include that movement in the explanation. This generates a more intuitive explanation as it prioritizes behavior that we observe and understand from the training data.
We can also reason the other way around: with no data preparation and a regular Euclidean distance metric the explanation strength is more heavily influenced by the input features that have a naturally larger variance, while being relatively unaffected by changes in input features with small variances. Ideally, as no prior information on the input feature's importance is assumed, each input feature is equally relevant when computing vector length and consequently explanation strength. This can be corrected for by using the transformation T before computing (Euclidean) distance.

Before summarizing the ideas discussed in this section in a formal definition, we explain one more important restriction on the vectors $g_i[x]$ and $e_i[x]$. We first present an auxiliary concept.

Definition 4.3 *Let $a, b \in \mathbb{R}^d$. We say that a **is contained** in b if $a = \gamma \odot b$ for some $\gamma \in [0,1]^d$. Put differently, a is confined to the parallelotope defined around the diagonal formed by b and the origin $\mathbf{0}_d$.*

To understand the restrictions we apply on the decomposition of $x - \bar{x}$, we change our perspective and observe the movements defined in (45) again in the original space $X = [0,1]^d$, by applying the inverse operation T^{-1}. The restrictions are now firstly that $T^{-1}g_i[x]$ should be contained in $T^{-1}(x - \bar{x})$, and secondly that $T^{-1}e_i[x]$ should be contained in $T^{-1}(x - (\bar{x} + g_i[x]))$. If we for simplicity interpret Figure 23 as being represented in the original space X, the dashed boxes can be viewed as the restrictions.
The reason for these restrictions, and simultaneously part of the reason for actually having the neutral point $\bar{x}$, is intuitive: the conditional explanation would not make sense if its general part exceeds the actual vector $T^{-1}(x-\bar{x}) \in X$ towards more extreme or unusual values, only to retract back from there and call that retraction its "explanation". More generally, any movement along some dimension in the input space X moving further away from the neutral point than the original element x itself, should not be considered attributable to x in any way. The restriction on $e_i[x]$ follows from the same reasoning, but now viewing $\bar{x} + g_i[x]$ as a starting point.

We now propose the definition for achieving input explainability in its most general form, assuming a properly chosen neutral point $\bar{x}$.

Definition 4.4 *Let $f : X_T \to [0,1]^N$ be a classification function, and let $x \in X_T$ be a given input element. Let $\bar{x}$ be a fixed neutral point in X_T. The set $\{g_i[x], e_i[x], d_i[x]\}$ is an* explanatory decomposition *of x for $i \in [N]$, if*

$$x = \bar{x} + g_i[x] + e_i[x] + d_i[x], \tag{48}$$

and, for some $\alpha, \beta \in [0,1]^d$,

$$\begin{aligned} T^{-1}g_i[x] &= \alpha \odot T^{-1}(x - \bar{x}) \\ T^{-1}e_i[x] &= \beta \odot T^{-1}(x - (\bar{x} + g_i[x])). \end{aligned} \tag{49}$$

The vector $e_i[x]$ is the conditional explanation *for the value of $f_i(x)$. Its* explanation strength*, associated with the above decomposition, is given by*

$$S(e_i[x]; \bar{x}, g[x]) := \frac{f_i(\bar{x} + g_i[x] + e_i[x]) - f_i(\bar{x} + g_i[x])}{||e_i[x]||}. \tag{50}$$

Note that after deciding on a neutral point $\bar{x}$, the entire decomposition, and thus its explanation strength, depends only on the variables α and β. This leads to the following more intuitive perspective.

Corollary 4.4.1 *Selecting an explanatory decomposition that maximizes the explanation strength of $e_i[x]$ is equivalent to selecting two points a and b in the parallelotope defined between the far corners $T^{-1}(x)$ and $T^{-1}\bar{x}$, such that $b - a$ is contained by $T^{-1}(x - \bar{x})$ and the explanation strength $S(a, b)$ is maximal.*

Proof 4.4.1 *The second problem stated in the corollary can be rephrased as maximizing $S(a, b)$ under the restrictions*

$$\begin{aligned} a &= T^{-1}\bar{x} + \alpha \odot T^{-1}(x - \bar{x}) \\ b &= T^{-1}\bar{x} + \gamma \odot T^{-1}(x - \bar{x}) \\ \alpha, \gamma &\in [0,1]^d \\ \alpha &\leq \gamma, \end{aligned}$$

where the inequality symbol is understood component-wise. The original problem of selecting an explanatory decomposition according to Definition 4.4 that maximizes $e_i[x]$, can be rewritten in this format by letting

$$\begin{aligned} a &= T^{-1}\bar{x} + T^{-1}g_i[x] \\ b &= T^{-1}\bar{x} + T^{-1}(g_i[x] + e_i[x]). \end{aligned}$$

This makes α identical in both problem statements, and lets γ follow from

$$\begin{aligned} b - T^{-1}(\bar{x}) &= T^{-1}(g_i[x] + e_i[x]) \\ &= (\alpha + \beta \odot (\mathbf{1} - \alpha)) \odot T^{-1}(x - \bar{x}), \text{ thus} \\ \gamma &= (\alpha + \beta \odot (\mathbf{1} - \alpha)) \end{aligned}$$

Given α, the restrictions on β allow choosing γ freely in the interval $[\alpha, 1]$. But this is equivalent to the constraints $\gamma \leq 1$ and $\alpha \leq \gamma$.

4.2.2 Selecting a neutral point

In the previous paragraph we have skipped details on what neutral point is to be used when defining explanation strength. This is firstly because we acknowledge that this choice can be dependent on the data set of the machine learning problem under consideration. For a decent comparison between different explainability methods, it is only necessary that the same point $\bar{x}$ is used. Acknowledging that in theory very exotic points can be selected and fine-tuned to make almost any explainability method perform well in terms of explanation strength, we do assume that the choices are made fairly, making intuitive sense and that they are substantiated by logical arguments.

Although the selection of $\bar{x}$ is thus partly up to the user of the method, we nonetheless propose a default approach that can be used as a starting point for most machine learning problems. We first go over the features we look for in the neutral point, which will lead in a straight-forward manner to our proposed algorithmic procedure for finding a good $\bar{x}$.
The usefulness of the neutral point $\bar{x}$ is very much dependent on the size of the potential impact $1 - f_j(\bar{x})$ for possible label classes $j \in [N]$. The higher $f_j(\bar{x})$ for some j, the less extra probability can be attributed to an explanation vector in the decomposition. If we are interested in explaining labels from the set J, a first idea would be to compute

$$\bar{x} = \underset{x \in \mathbb{R}^{d_T}}{\arg\min} \max_{j \in J} f_j(x). \tag{51}$$

Although this set-up would potentially increase explanation strengths by enabling larger impacts, the interpretative quality of the explanation is not guaranteed. For example, if one would choose a point $\bar{x}$ that lies far away from all data points $x \in X_T$, the vectors $x - \bar{x}$ will point in reasonably similar directions regardless of the class that x represents. Although uninformative similarities between the vectors should ideally be taken care of by being placed in the general part of the decomposition, there may also be configurations where this is not possible (i.e., when $\bar{x}$ lies on the extension of a line spanned by two data points).[17] We therefore conclude that an extra point of attention in choosing $\bar{x}$ is that it should preferably lie *between* input space regions representing high probability for different classes of interest. (This observation makes the mean of the training data, used in Figure 22 as a first example, indeed an interesting starting point.)

[17] In the original data space X, an example of this problem in practice would be to pick the empty (white) image as a neutral point. Any explanation vector can then only consist of positive values. The *absence* of input signal, which might for example distinguish a label 6 from a label 8 by inspecting the top-right corner of the digit, can not be used in an explanation.

We propose to enforce this by focusing directly on the angle between vectors $x - \bar{x}$ for different data elements $x \in X_T$, suggesting[18]

$$\bar{x} = \underset{x \in X_T}{\arg\max} \sum_{s<t \in S_T} \arccos\left(\frac{\langle x-s, x-t\rangle}{||x-s|| \cdot ||x-t||}\right), \tag{52}$$

for some subset S_T of the transformed training data, possibly filtered to contain only classes that we are interested in explaining.

The two objectives can be combined using a tuning parameter $\lambda \in [0,1]$, suggesting that we can find some kind of optimal $\bar{x}$ by

$$\bar{x} = \underset{x \in X_T}{\arg\min} \left(\lambda \max_{j \in J} f_j(x) - \frac{1-\lambda}{M(|S_T^J|)} \sum_{s<t \in S_T^J} \arccos\left(\frac{\langle x-s, x-t\rangle}{||x-s|| \cdot ||x-t||}\right)\right), \tag{53}$$

where S_T^J represents a data-set possibly filtered to contain only training elements belonging to classes of interest $j \in J$, and $M(|S|)$ is a normalization constant for properly computing the average angle between elements of S. It is given by

$$M(|S|) = \frac{|S|^2 - |S|}{2}.$$

The minimization can be realized using stochastic gradient descent, using a single draw from the Bayesian network on every step for $f_j(x)$. Also, rather than computing the average angle based on all elements in S_T^J, different batches of the data-set can be used in every step of the SGD iteration.

Although this procedure is a convenient way to select $\bar{x}$ in an abstract data-set with little prior information or sense of important directions, it may very well be the case that the user of this method already has a neutral point $\bar{x}$ in mind. For anomaly detection one might select $\bar{x}$ to lie at the centre of elements from the normal class. If there are multiple different normal classes, clustering can be used to find the respective centres of these classes, and $\bar{x}$ can be selected based on the relevant centre for the element x under consideration. We do note that these tricks are in principle taken care of by the method itself.[19] However, if possible, it is nonetheless good to consider using any known information prior to the method, thus potentially selecting or altering $\bar{x}$ manually.
In fact, $\bar{x}$ can also be interpreted more freely, in order to create general problem statements of the form: "Given that an element used to be (/is ordinarily) given by $T^{-1}\bar{x}$, but is now observed as $T^{-1}x$, what specifically caused the model output to now be given by the label $\arg\max_j f_j(x)$?".

[18] With a slight abuse of notation by writing $s < t$, assuming the data elements are ordered.

[19] selecting J to only contain the anomaly class will automatically steer $\bar{x}$ towards the centre of the normal classes to maximize potential impact (due to the second term of equation 53). Also, in the case of multiple clusters of normal regions, the general part of the decomposition can take care of the movement towards a cluster centre, from where the explanation can begin.

4.2.3 The EVD-method

The Explanatory Vector-Decomposition method follows in a straight-forward manner from the definitions in Section 4.2.1. Having mathematically stated what an explanation actually is and how to measure its strength, the obvious objective is to find the explanation with the highest explanation strength for a given input x. This comes down to optimizing (50), where gradient descent is used to update the variables α and β.
In principle this method is model-agnostic, as the actual form of $f(x)$ has not been specified nor used up until equation 50. However, finding the optimal α and β using gradient descent would require f to be differentiable. We propose the method specifically for explaining deep learning models, thus relying on the back-propagation algorithm for efficient computation of gradients.
To apply EVD to a Bayesian neural network, we take a single random draw of the parameters from the posterior distribution in each step of the optimization procedure. In this sense we use a form of stochastic gradient descent, where the stochasticity follows from the parameter selection rather than the data selection. This is similar to how we implemented the RDE method on a Bayesian neural network. Moreover, selecting new random parameters in each iteration is an approach also used in the training of the Bayesian neural network itself.
We summarize the newly defined concepts and subsequent exploration in a step-wise description of the method. For the pseudo-code we refer to Appendix D.

1. Normalize and decorrelate the training data, using T as in (44) as a map to the new data space $X_T \subset \mathbb{R}^{d_T}$
2. Find a neutral point $\bar{x} \in X_T$ using batch SGD to solve (53). Dependent on prior information about the data and relevant preferences for the explanation, $\bar{x}$ can also be chosen or fine-tuned manually.
3. Maximize the explanation strength in (50) using stochastic gradient descent. With a Bayesian neural network a single draw of the parameters is selected in each step of the optimization procedure.
4. Given the general direction $g_i[x]$ in which x is moved relative to $\bar{x}$, propose $e_i[x]$ as an explanation for the (significantly large) value of $f_i(x)$.
5. Translate $e_i[x]$ back to the more interpretable original training data space, thus returning $T^{-1}e_i[x]$ as the actual explanation.

Although the denominator in the explanation strength definition implicitly relies on an elaborate prior data preparation, hence justifying the use of the Euclidean metric as a way of creating more intuitive explanation vectors, the numerator of this equation may seem quite arbitrarily chosen. One might believe more interesting measures for vector impact exist than the difference between two probabilities. Or some normalization could be applied, to correct for the number of classes. However, it is precisely this simple and clear definition of explanation strength that gives the returned vector $e_i[x]$ its useful interpretable properties.

More specifically, using the definition of impact

$$I(e_i[x]) := f_i(\bar{x} + g_i[x] + e_i[x]) - f_i(\bar{x} + g_i[x])$$

for the numerator, we can now say that: *given $\bar{x} + g_i[x]$, we find that $e_i[x]$ is the reason for an increase in probability of $I(e_i[x])$ in the model's assessment of the label i.*
In this sense we accomplished an earlier discussed goal: developing an explainability method that returns an actual explanation, rather than a proxy measure. Another desired quality of the method was that it could be used as a robust measure to quantitatively compare explanations generated in different machine learning problems. For this purpose we re-define explanation strength as

$$S(e_i[x]) := \frac{f_i(\bar{x} + g_i[x] + e_i[x]) - f_i(\bar{x} + g_i[x])}{||e_i[x]||/\mu(\bar{x})}, \tag{54}$$

were we normalize the explanation vector length using

$$\mu(\bar{x}) := \frac{1}{|S_T|} \sum_{x \in S_T} ||x - \bar{x}||. \tag{55}$$

By doing so, we prevent the number of dimensions of having a substantial impact on the explanation strengths, making it a more stable measure.[20]

4.2.4 Interpretability regularization

Up to now we have focused on maximizing explanation strength as the main goal. As such, there is no guarantee that the output of the optimization described in the previous section leads to a useful result in practice. Of course, we have deliberately separated interpretation constraints from the explanation strength definition, in order to make it a more objective measure. However, we will now consider altering the optimization procedure to allow for the implementation of specific output preferences.
From a mathematical perspective, an obvious risk is that the optimization tries to minimize the explanation vector length to near-zero, indeed creating large explanation strengths, assuming the explanation impact is still sufficiently high. This might seem a problem of the method itself, but we in fact do want to allow this behavior more generally: one might still be interested in the (relatively large) impact of extremely small changes. However, we also want to enable restrictions on the explanation length, and thus introduce a parameter $\lambda > 0$ and let the loss function for optimization be given by

$$L(\alpha, \beta) := -\frac{f_i(\bar{x} + g_i[x] + e_i[x]) - f_i(\bar{x} + g_i[x])}{\max(||e_i[x]||/\mu(\bar{x}), \lambda)} \tag{56}$$

[20]Note that because of the data transformation, the restrictions used in equation 49, or corollary 4.4.1, do not necessarily imply $||x - \bar{x}||$ is the *maximal* length of the explanation vector $e_i[x]$. However, we nonetheless assume it is a relevant vector for comparing lengths.

The parameter λ can be interpreted as the minimal explanation vector length, normalized by $\mu(\bar{x})$ from (55). The idea is that when the length falls below λ, the optimization procedure will no longer take length into account and switch to only maximizing explanation impact. Under the assumption that this will increase the explanation vector length, we eventually end up with a length that cannot be far below λ.[21]

A second option to add interpretability and flexibility to the method is to restrain the impact from becoming too low. We add two other parameters, $\delta > 0$, representing the desired minimal impact, and $\gamma > 0$ representing how much weight this restriction will receive in the cost function. Writing $I[x]$ for the impact, and $l_\lambda[x]$ for the denominator of (56), we now define the loss function as

$$L_{\lambda,\gamma,\delta}(\alpha, \beta) := -\frac{I[x]}{l_\lambda[x]} + \gamma \cdot (\text{ReLU}(\delta - I[x]))^2 \tag{57}$$

The ReLU function assures that the constraint kicks in only when the impact falls below δ, and we square the difference to allow for a smoother behavior, restricting only a little bit when $\delta \approx I[x]$. Of course this is a matter of taste, if it is extremely important that $I[x] > \delta$, one may want to remove the square on top of making γ very large.

4.3 Implementation and Results

In this section we implement the EVD method using a Bayesian neural network on the MNIST data set of digit images. As we will also use the network implemented here in the numerical experiments of Chapter 5, and we currently only have a laptop with 8GB RAM, we change the architecture of the BNN slightly. The implementation largely follows the description of Section 3.2.4; the only difference is that we now use 64 and 32 kernels (instead of 90, 50) in the first two convolutional layers. This brings the parameter total to 716246. Training this Bayesian network takes approximately 5 minutes, achieving a test accuracy of 0.952.

Before going into the specifics of how the EVD method's optimization itself is implemented, we first go over the data preparation and describe how we picked a neutral point. We then explain how the optimization procedure is implemented and show the results by mapping the vectors in X_T back to their original representations. In this sense we follow the step-wise description of the method as described in the previous section.

[21] If it is really essential that the explanation length is larger than λ, we can always proportionally increase $||e_i[x]||$ in X_T (and appropriately decrease $g_i[x]$ and $d_i[x]$) to meet this requirement. We expect that only a very minor change is needed.

4.3.1 Preparing the data

The method has been described in the most general way, and can be applied to many different models. However, the *convolutional* (Bayesian) neural network is a special case when it comes to the data preparation, as the use of convolutional (as well as pooling) layers is heavily dependent on (local) input correlation.
Still requiring a proper transformed space X_T to work with explanation vectors, their lengths, and the neutral point $\bar{x}$, we solve this issue by simply redefining the model. We first train f on normalized elements $T^{-1}T(x - x_{\text{mean}})$ in X, applying $T^{-1}T$ to assure we really are working with a transformed data set[22], but then change the architecture of the network by placing T^{-1} before the first layer. This ensures that indeed $f : X_T \to [0,1]^N$, as required, while still being able to use convolutional and pooling layers to benefit from data correlations.

It now only remains to decide how many dimensions are removed from the image based on PCA. Recall that any removal of dimensions is entirely reversible *if* the information loss has been negligible.
Of course one could observe for each d_T how the performance of the model improves or deteriorates, using k-fold cross validation. Depending on the preference, weighing model performance against potential stability on new inputs that differ from regular data, an optimal d_T would be selected. Also, going a step further, one could argue that we are in fact not interested in model accuracy, but rather in the EVD method's performance. This suggests computing the average explanation strength that the EVD method achieves when testing it on a large sample of input elements, for set-ups based on different d_T.

Although this may be interesting research, it is very time-consuming, requiring retraining the model, as well as running the EVD method, many times. We also note that the original purpose of removing dimensions never was to try to optimize the model or the EVD method: it was simply to remove the smallest (near-zero) standard deviations to prevent them from disturbing the method's basic functionality. We therefore skip this time-consuming investigation, deciding on $d_T = 600$ based on two quick observations:

- We shortly take note of all the standard deviations (Figure 25), to see how many dimensions in specific have relatively small standard deviations.
- We inspect the visible information loss by applying the $T^{-1}Tx$ "identity" transformation[23], when lowering d_T, see Figure 24.
  Even though the exact impact of small d_T on the model and the EVD method might be uncertain, as our research mainly concerns explainability we do not want to sacrifice clarity of the images in X.

[22] By doing so we take the possible information loss of the map $T : X \to X_T$ into account.
[23] Originally this transformation is given by $T^{-1}T(x - x_{\text{mean}}) + x_{\text{mean}}$. However, to assure we really are working with a transformed data set in every respect, we also redefine x_{mean} by setting it equal to $T^{-1}Tx_{\text{mean}}$ making these transformations equivalent.

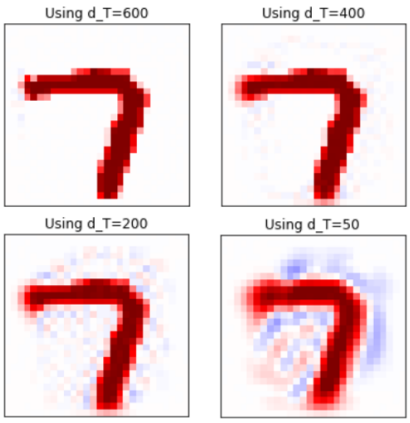


Figure 24: An element from the train data set is transformed using $T^{-1}Tx$ for different dimensions d_T. (An interesting observation: the model still achieves an accuracy of 0.945 for $d_T = 50$.)

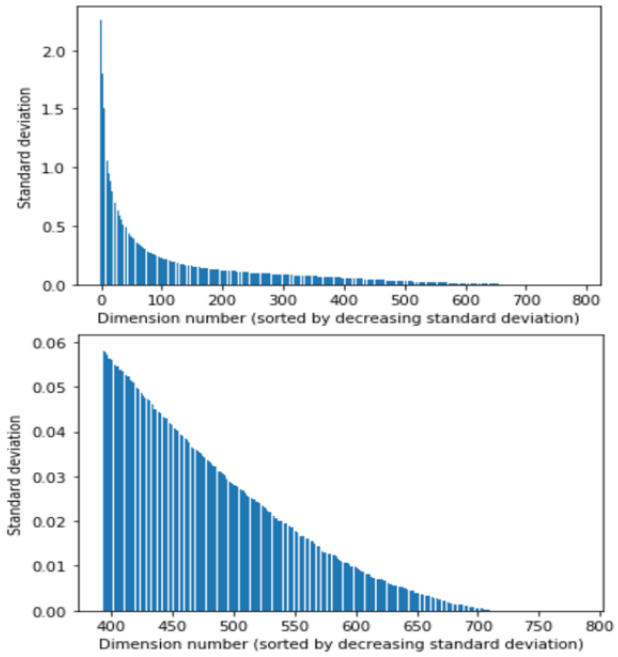


Figure 25: Standard deviations of dimensions in the decorrelated space. The dimensions have been labeled from 0 to 783, sorted by decreasing standard deviations. On the bottom image we selected only the second half of the dimensions, to more clearly show the flattening tail of the barplot.

4.3.2 Finding the neutral point

We first show a reasonable attempt at picking a neutral point $\bar{x}$ manually, and investigate the choice of using the data mean, continuing the example of Figure 22. As the data was normalized, this point is represented by $\mathbf{0}_{d_T}$ in X_T.

With regards to the desired property of $\bar{x}$ being centered in the data, this is a good choice. Using $M = 300$ independent draws of train data elements $x_m \in X_T$, taking care to create a perfectly balanced data-set over the 10 digit classes, we compute the average angle

$$A(\bar{x}) = \frac{2}{M^2 + M} \sum_{s,t \in [M], s<t} \arccos\left(\frac{\langle \bar{x} - x_s, \bar{x} - x_t \rangle}{||\bar{x} - x_s|| \cdot ||\bar{x} - x_t||} \right),$$

finding $A(\mathbf{0}_{d_T}) = 0.499\pi$. This comes as no surprise, pointing out how the expression within the inverse cosine resembles correlation between x_s and x_t. As the data set is decorrelated we expect this value to be close to 0, and the inverse cosine is linear in this region.

However, the mean of the data is a less ideal candidate when it comes to the potential explanation vector impact, which is bounded by $1 - f_j(\bar{x})$ for each class of interest j. We observe this by plotting the outputs of $f(\bar{x})$, which can be seen in Figure 26.

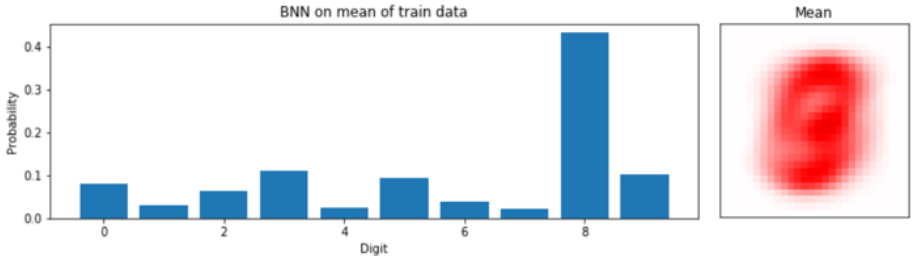


Figure 26: Bayesian neural network output on mean of the data. Most notably the digit 8 will be more difficult to explain using this neutral point, due to a lower possible impact.

Not being completely satisfied by this first attempt, we decide to compute the neutral point $\bar{x}$ using (53) with batch stochastic gradient descent. Starting with $\mathbf{0}_{d_T} \in X_T$, using $\lambda = 0.7$ and balanced batches of size 30 each, the optimization process[24] returns the $\bar{x}$ pictured in Figure 27. We compute an average angle of $A(\bar{x}) = 0.498\pi$, and its output $f(\bar{x})$ shows high potential impacts for each digit. The optimization takes less than a minute to converge.

[24] We use Adam with the standard parameters as before.

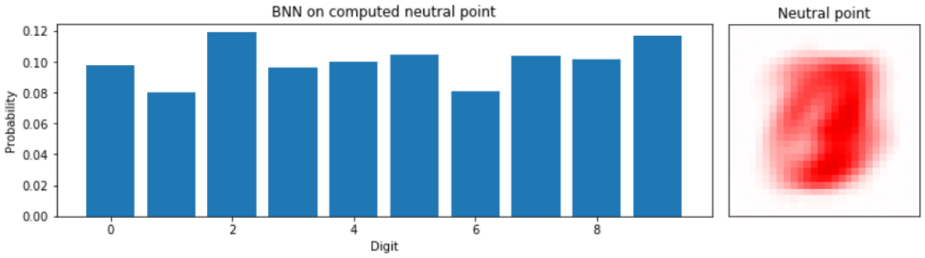


Figure 27: Resulting neutral point $\bar{x}$ from running the optimization procedure in (53). We also depict the outputs of $f(\bar{x})$, showing a more equally divided probability over the labels.

4.3.3 Results

Having eventually finalized our set-up, consisting of a data preparation approach, a Bayesian neural network and a neutral point $\bar{x}$, we can now start the actual optimization procedure. We again use an Adam optimizer with the default settings to solve the loss function, as given in Section 4.2.4, with stochastic gradient descent. We initialize using $\alpha = \mathbf{0}$ and $\beta = \mathbf{1}$, so that $e_i[x] = x - \bar{x}$ at the start of the iteration.
We first show some results for the set-up where $\gamma = \lambda = 0$, thus applying no interpretability constraints and purely focusing on explanation strength. Each run, using 1500 optimization steps, takes less than two minutes. An arbitrary selection of images is shown in Figure 28.

We notice that the EVD method has, at least in the case of applying this BNN on the MNIST dataset, a natural preference for selecting smaller explanation vectors. Running on a balanced set of 50 images we indeed find an average explanation length of 0.019 ± 0.006. However, the average impact of 0.31 ± 0.05 arguably makes the results interesting nonetheless, as this shows that these very small movements in input space have been discovered with relatively large impacts on output probability. Indeed, we find an average explanation strength of 17 ± 4. This can be interpreted by stating that moving a distance of only 1% of the average vector length measured in the data set (i.e., $\mu(\bar{x})$, (55)) along these explanation vectors, causes an increase in output probability of approximately 17%.
We acknowledge that depending on the problem at hand, one may not be interested in finding such small explanation vectors with relatively high impacts. We therefore turn our attention to the interpretability constraints. Noting how we specifically had very short explanation vectors, we now experiment with $\lambda = 0.1$. Recall from Section 4.2.4 that this makes the loss function in the optimization indifferent to lengths smaller than $0.1\mu(\bar{x})$. Results using these setting are shown in Figure 29.

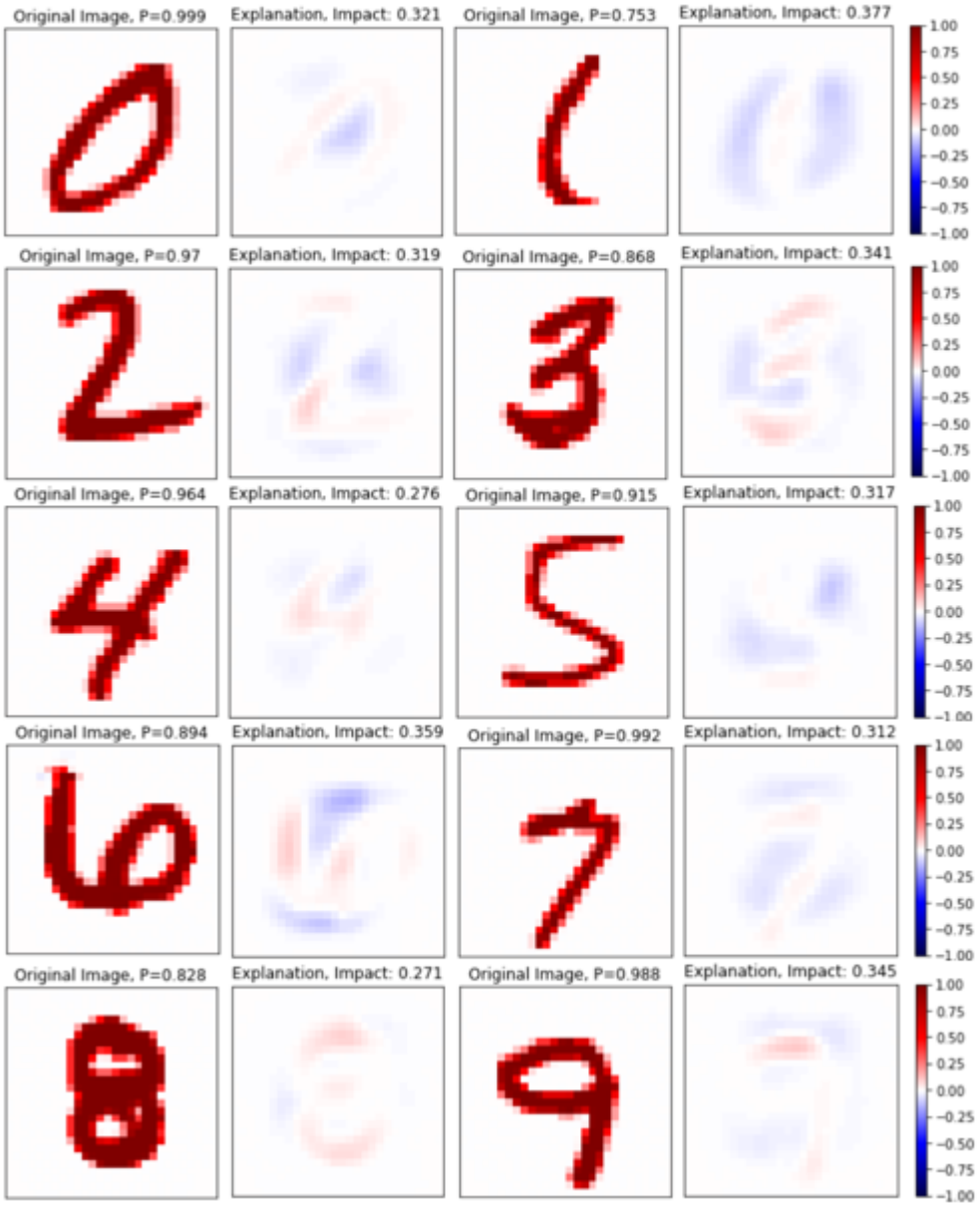


Figure 28: Results of the EVD method on each of the digit classes, using no interpretability constraints. For clarity we depict only the explanation vectors with their impact (i.e., the difference in classification probability that they caused) in the image titles.

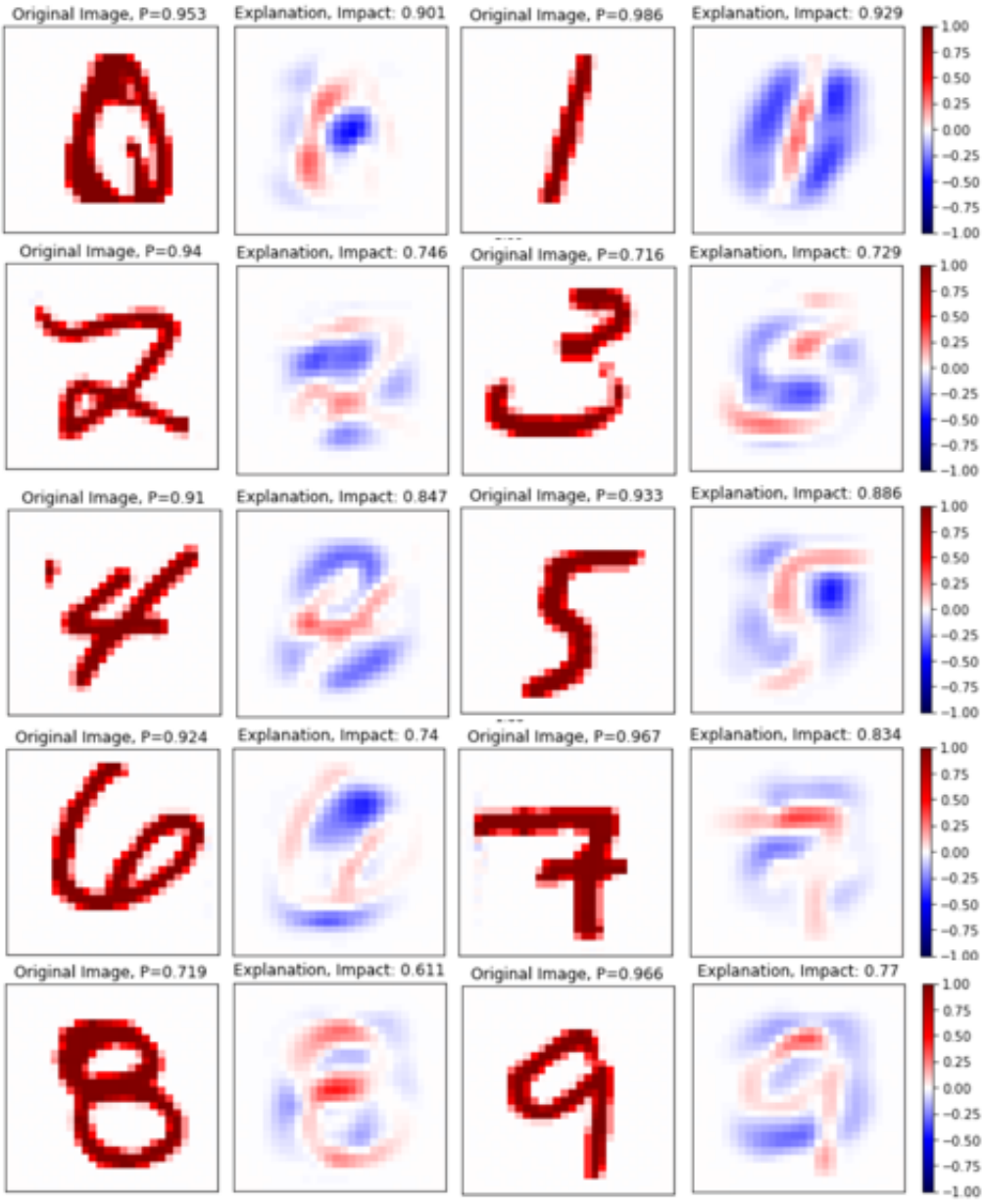


Figure 29: Results of the EVD method on each of the digit classes, using $\lambda = 0.1$. Taking the averages on a balanced set of 50 images, we find the length restriction to be very effective, leading to an average explanation length of 0.100 ± 0.003. These settings lead to an average impact of 0.83 ± 0.08 and an explanation strength of 8.5 ± 1.0.

Of course there are more results to show than just the explanation vector, and we can in fact depict the full explanatory decomposition. However, as it consists of a large set of pictures for each image, we just treat one example in Figure 30, only aimed to improve the general understanding of the method and its potential use.

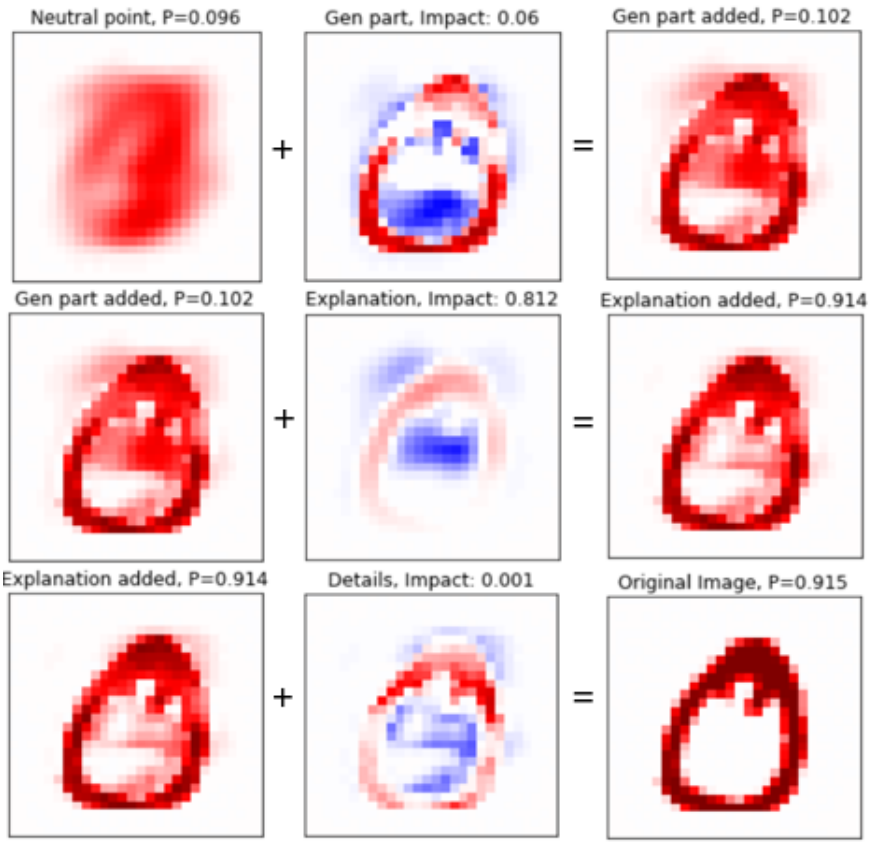


Figure 30: The full explanatory decomposition returned by the EVD method on an arbitrary digit, using $\lambda = 0.1$. Inspecting first the general part, we see that the model does not attribute much importance to the lower part of this digit 0. Indeed, we can imagine any of the digits 3, 5, 6, 8 or 9 being formed using the same lower curve and empty region. However, as the explanation vector shows, the fact that the top part of the centre is *also* empty creates the decisive case for the digit 0. Observe also how the small irregularity in the top right of the original figure is placed under the detailed part, having minimal influence on the decision: in the middle column its presence is visible only in the details vector, while notably missing from the general and explanation vectors.

4.4 Discussion and concluding remarks

Having shown what the EVD-method is capable of for a Bayesian neural network trained on the MNIST data-set, we briefly discuss its success, making a qualitative comparison to earlier methods. We then go a step further and contemplate a wider variety of uses, that have not been investigated here. We also consider the impact what model is used on the EVD method's performance, and investigate the specific case of using a regular (non-probabilistic) neural network.
We conclude this chapter after hypothesizing about possible disadvantages and improvements, concerning ourselves with both the underlying theory as well as the implementation of the method.

4.4.1 Discussion on the EVD method performance

We first note that we are not explaining the digits themselves, but rather how *the model* perceives those digits. One may look at some pictures of the previous section and have comments on the outputted images. Although not all of these results look equally intuitive, and some may not follow the expectations of what *we* believe are the appropriate classifications, the model in fact does attribute a large impact to the shown explanation vectors.

We now briefly point out some qualitative observations. First, note that the method is equally successful as LIME and RDE in generally detecting what the relevant part of the image is, and similarly shows a distinction between more and less important regions of the input here.
We also see the effect of minimizing the explanation vector length in X_T. Recall from Section 4.2.1, specifically the discussion after (47), that we deliberately choose to minimize in this transformed coordinate system to aim for an intuitive vector $e_i[x]$, that would resemble vectors in X. The effect is most clear when comparing to the more grainy images of the general or details vectors, which have not been optimized for this purpose.
Although these latter two have no smooth appearance, they can still contain useful extra information, showing which parts of the image are considered too generic or too detailed to contribute significantly to the output.

As a more general remark, we note that most explainability methods, especially those concerning feature importance, assess each feature (or a selected sub-set of features, as in LIME) as an individual input for which some quantity of interest is computed, whereas we deliberately switch to a different system X_T were the combined effects of input features can be taken into account more truthfully. This is especially appropriate when dealing with highly non-linear models such as neural networks. Although tempting to keep the features separated for interpretability purposes, we have structurally separated explainability and interpretability, allowing a more directed search for the "true" explanation, which

eventually is translated back to an interpretative representation.

We end this discussion of the method by noting that the full advantage of this method might not be completely grasped by looking at images alone. For image data, knowing the precise difference in the shade of a pixel that influenced the model's eventual decision may not be very interesting. Being able to (literally) view the bigger picture is often more valuable. However, for many other data sets, the decomposition of feature values into three distinct categories could be a more noticeable strength of the EVD method.

We speculate, for example, about a model trained for detecting financial crime. If we were to ask why the model classifies a certain party as suspicious, basic feature importance explainability methods would only point out the features that contributed to the decision. More sophisticated methods would have the benefit of quantifying how important those features have exactly been compared to other features, but the EVD method goes a step further. Rather than showing relative importances among features, it dives more into the details of the features itself, by extracting the truly explanatory part.
Say some person has been labeled as suspicious based on feature n, representing a certain transaction value of $10,000$. The EVD method would not only point out this feature, but also separate it into 3 parts, using an intuitive reasoning like:

1. Normally the model does not expect this transaction at all, i.e., $\bar{x}_n = 0$.
2. Given that the transaction is present, then in this case there generally would be no reason for concern if it stays below approximately $g_i[x]_n = 300$.
3. The biggest impact for the model's decision is due to the fact that the transaction is not just 300, but in fact an extra $e[x]_n = 700$ higher.
4. Having already been labeled as suspicious, the remaining $d[x]_n = 9000$ might make the situation even more extreme, but has not contributed to the decision.

Not only do we expect the method to recognize the feature as important, we also expect that it would point out that the threshold that most influenced the decision lies in the area $300 - 700$. One can understand that such an elaborate breakdown of (important) features could be arguably more valuable for other data sets than the MNIST digits data set, where it is not that compelling to locate the most influential shade-threshold of pixels.

4.4.2 Variety of explainability options

We have implemented and shown the results of the EVD method for only a very specific goal: given x, correctly predicted to belong to class i, why do we observe $f_i(x)$? However, the framework allows a much larger variety of possible questions and investigations. We can in one question simultaneously ask for the reason why x has relatively high probability on a set of classes I, and a relatively low probability on a set of classes J. This can simply be achieved by altering the numerator in (50), using instead

$$\sum_{i\in I} f_i(\bar{x} + g_i[x] + e_i[x]) - f_i(\bar{x} + g_i[x]) + \sum_{j\in J}(f_j(\bar{x} + g_j[x]) - f_j(\bar{x} + g_j[x] + e_j[x]))$$

as the numerator when defining explanation strength. Note that the more classes are added to I or J, the more questions $e[x]$ tries to answer simultaneously. For a large number $|I| + |J|$ this may likely cause the result to be less insightful.

As briefly mentioned in Section 4.2.2, preferences on which classes we wish to explain can also be taken into account when computing the neutral point, as well as from what perspective we wish to explain these classes. By having the freedom to place $\bar{x}$ in any high probability region or cluster centre corresponding to a certain label i, we can ask the most general question: "If x would have originally been more like $\bar{x}$, why is it that now, taken that it no longer resembles $\bar{x}$, the output $f_i(x)$ for $i \in I$ is high while $f_j(x)$ for $j \in J$ is low?

To conclude this discussion we give a few more conceptual examples of the advantages of the suggested flexibility.

- If $f(x)$ returns a wrong label, i.e., j instead of the correct i, we can ask: "Why not i?", or even "Why j and not i?".
- If $f(x)$ does not return a clear answer, but shows doubt between two classes i and j, we can ask the combined question "Why i and j?". If we are in need of making a choice between i and j nonetheless, we can investigate further by asking "Why j and not i?", and vice versa, to see if we are swayed by the explanations.
- If we are interested in the relative similarities or differences between classes, we can take a large number of elements from class j, and ask "Why (not) i?". Repeating this for different i and j may give general (global) insights of the model's behavior across the entire input space. The averages of the explanation strengths computed allow giving quantitative conclusions rather than having to settle for subjective comparisons.

- In anomaly detection we may place $\bar{x}$ in the centre of the normal class, or in any cluster centre deemed relevant if multiple clusters of normal elements can first be discovered in the data set. We can then ask specifically: given that x is relatively similar to the normal cluster centre $\bar{x}$, why does f believe x to be an anomality?
- Note that in the above examples we may have casually wrote "*i and j*" while in the truest sense the optimization interprets the altered numerator as "*i or j*". However, we can use both these logical options, by defining the numerator using max or min functions.

As a final remark we note that although the decomposition has been defined by splitting $x - \bar{x}$ into three parts, it is possible to experiment with a 2-part decomposition by setting $\alpha = 0$ in (49). Returning briefly to the discussion at the end of the previous section, the unnecessarily elaborate investigation into precise pixel-shades caused by a three-part decomposition may even have a negative influence on the interpretability of the outcome in the current case. For image recognition a two-part decomposition can actually be more insightful.

Using the two-part decomposition, aiming to simply extract explanatory movement directly from the starting point, we can also take a more general approach: allowing x to be a variable itself, we can search for the most explanatory movement starting from any chosen $\bar{x}$ towards (or away from) any desired goal(s). One can even move through the input space in steps by iteratively applying the method using the newly discovered point x as a starting location, fixing the step size using the regulatory parameter λ.

Clearly the method's flexibility allows it to be fine-tuned to many possible problems encountered in practice. Investigating these, and potential other uses, can be an interesting topic for further research.

4.4.3 Model selection and the use of a Bayesian neural network

As the role of the neutral point $\bar{x}$ is significant, a valid consideration is whether such a point can actually be found for any model f. The results in 3.2.4, most notably Figure 14, suggest this may be problematic for some models. More generally, if we recall the comparison between Bayesian and regular neural networks in that section, an important conclusion was that the regular neural network was more likely to show illogical behavior on data points that are unlike the training elements $x \in S_X$. This causes a concern: perhaps some models are too erratic to properly apply the EVD method in the first place, as it heavily relies on the model's behavior between data points.

We put this idea to the test for a regular neural network, by setting all parameter variances equal to zero in our Bayesian neural network. This still gives a decent regular neural network, achieving a test accuracy of 0.951. More importantly, we can still find a neutral point, relatively similar to the one used in the implementation of EVD for the Bayesian neural network. We also discover that the method still produces similar images on a couple of test runs.
When running on the same inputs used as in Section 4.3.3, setting $\lambda = 0.1$, we even find a slightly higher average explanation strength of 9.2 ± 0.3. This may be due to the optimization being able to target the minimum (or minima) of the cost function more precisely, now that the deterministic nature of the model fixes the location of this minimum in the input space.

Although this deterministic neural network on the MNIST data set may thus still allow use of the EVD method, we acknowledge that this does not remove all concern. In practice regular convolutional neural networks have often been shown to behave erratically on their input spaces[Sze+13], see for instance Figure 31. This illogical and unpredictable behavior might cause the optimization for explanation strength to get stuck in local minima, finding a very strong explanation vector that may not represent an intuitive explanation. This may even be more likely the case when one is experimenting with the alternative application ideas suggested in the previous section.

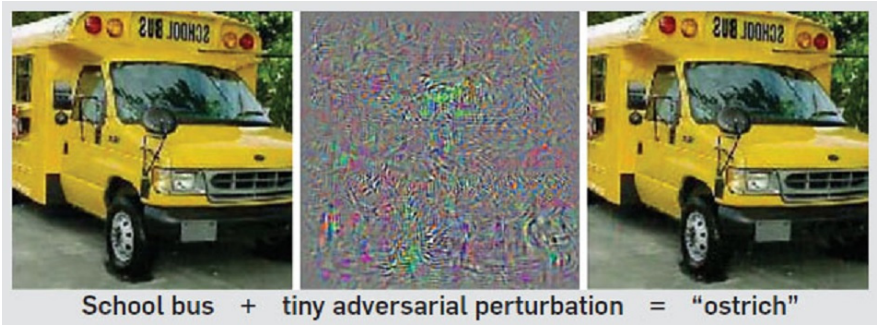


Figure 31: A convolutional neural network is shown to be easily influenced by a specific perturbation, figure from [Sze+13].

However, we suggest two reasons for why the EVD method is still appropriate. Firstly, if the method indeed gets stuck in a local minimum, it may be the case that it does not deliver the explanation that the user was hoping for, but it gives valuable information nonetheless. Note that the resulting explanation vector, no matter how strange and illogical it is, *always shows true behavior of the model*, which cannot be said of all other explainability methods.

This brings us to the second point. If a model can indeed be easily influenced, making a school bus look like an ostrich as in Figure 31, and if this behavior takes such an extreme form that the EVD method does not show *any* insightful results, perhaps we should not even trust an explanation method at all for this model: imagine a method being able to explain flawlessly and intuitively why a certain black box model predicts the image on the left of Figure 31 to be a school bus. What would you make of this explanation when learning that the model classifies the image on the right as an ostrich?
We can go a step further and argue that it is a fallacy of that explainability method to *not* discover and point out how easily the image could be perturbed into something else, giving a false sense of trust in the model's understanding of the data.
Of course, the above discussion is hypothetical, as we have not further experimented with other, possibly more erratic, models and different data sets. Although it has originally been intended as an explainability method for (Bayesian) deep learning, applying the EVD method using different models can be an interesting topic for further research.

4.4.4 Possible Improvements

We divide this section into two perspectives, discussing both ideas on the method's underlying theory as well as our implementation of it.
With respect to the theory, we take a moment to acknowledge the deeper thought process underlying the final choice of the numerator in the explanation strength definition. Firstly, when explaining $f_i(x)$ one may want to include the probabilities of other classes $j \neq i$ in the definition. Intuitively an explanatory output change of $f_i(x) = 0.3$ towards $f_i(x) = 0.5$ may seem less revealing for class i, if during that change the distribution of outputs for other classes changed from being a uniformly divided $0.7/(N-1)$ towards having one other class j also receiving a probability of $f_j(x) = 0.5$.
However, we chose to stick to focusing on $f_i(x)$ alone, noting that the just described intuition may be a little treacherous: mathematically speaking the actual model's certainty about label i strictly depends on $f_i(x)$ alone, and not on how the remaining probability $1 - f_i(x)$ has been distributed.

Given that we thus embrace a more binary intuition of a probability $f_i(x)$ for i and a probability for "the rest" given by $1 - f_i(x)$, we could still value particular changes in the interval $[0, 1]$ differently. The change in probability from 0.4 to 0.6 might be far more decisive than the change from 0.8 to 1.0. On the other hand, exactly *how* to value such changes, even deciding whether to use a convex, concave or sigmoid shaped function for the valuation of probability on $[0, 1]$, is unclear, because it can vary depending on the application at hand. In some cases probabilities of 0.4 or 0.6 are both relatively meaningless, if the problem deals with such delicate or important decisions that it generally requires a minimal accuracy of, say, 0.975.

The EVD method as currently presented is flexible enough to (be adapted to) deal with such situations, but loses part of this flexibility if we would impose prior ideas on how to valuate probability changes. And of course, a more general reason for keeping the numerator simple is that it leads to a more interpretative method, which is arguably more important in the current context of explainability research. However, we do not exclude the possibility that more practical and equally insightful approaches can be proposed.

We now turn our attention to the restrictions posed on the specific parts of the explanatory decomposition. Specifically that they should be contained in the "box" defined by $T^{-1}\bar{x}$ and $T^{-1}x$. Our main concern with these restrictions is that although they make intuitive sense, we also care about our original intention of separating explainability and interpretability, with the first encapsulating a truly mathematical definition and only the latter taking care of its interpretation. The restrictions may have crossed this line, influencing the definition of explanation strength based on reasons from an interpretation perspective.
A first small idea is to simply be less restrictive. For instance allowing the details part $d_i[x]$ to retract back to $T^{-1}x$, or to more generally allow the restrictions to be less strictly enforced, using instead regularization in the loss function. These ideas open up a whole new world of possibilities, which we have not further investigated.
A more fundamental point of interest is that we defined these restrictions in the original data space X, as we argued that explaining by "retracting back" towards $T^{-1}x$ is unsensible, but it may in fact be more accurate to use the space X_T when defining these restrictions. This may lead to an explanation exiting the restrictions box (Figure 23) in X, but any negative consequences on the interpretation of this outcome would, and should, then be handled by interpretation regularization.

Concerning the implementation, a general point of attention is that we have proposed many new definitions and optimization techniques simultaneously, and a proper experimental investigation, taking into account possible interactions between choices made, is still needed to fine-tune parameters for optimal use.
More generally speaking, the results of the method have been far from optimized in this research, and the main focus has been on developing the underlying theory.
As discussed we have tried to separate explainability and interpretability in this theory. We acknowledge that the emphasis in our research, both in theory and implementation, has mainly been towards defining and obtaining explanations, leaving many ideas for interpretability, and better implementations of current ideas, for further research.

5 Numerical Experiments: Comparing explainability methods

Having described and implemented three different explainability methods, namely LIME, RDE and EVD, we now compare their performances on the MNIST dataset. We use the newly developed definitions of the EVD method to make this comparison, focusing on the average explanation strength achieved when explaining elements from a validation data set.

As the RDE and LIME methods do not return explicit explanations, we propose FITEV, short for Feature Importance To Explanation Vector. This is a method to change the outputted feature importances of (for example) RDE and LIME into similar explanation vector decompositions as those that the EVD method is based on. As we will see, FITEV not only allows comparisons between different methods, but it can also be applied to add new insights to explainability methods that would usually only return feature importances.

5.1 Introducing the method

The FITEV method is applied in the same framework as the EVD method, i.e., the same data preparation steps are required, see Section 4.1. In this chapter we assume this has been properly taken care of. In other words, we have trained a Bayesian neural network $f : X_T \to [0,1]^N$, where $X_T \subset \mathbb{R}^{d_T}$.

We first go over the implementation of LIME and RDE, explaining how we can obtain feature importances in X in this new setting. We then introduce the FITEV method.

The essence of the FITEV method comes from first assuming that the more important a feature is, the bigger the part of it that belongs to the explanation vector in the vector decomposition. Although this may be a strong assumption, it is almost inevitable as a first step, considering the information available. Note also that for every pixel we only have 1 piece of information, namely its feature importance, whereas an explanatory vector decomposition in X requires two variables for each pixel (α and β in definition 4.4). We thus need to add variables to the problem, and shall allow optimization of these variables with respect to explanation strength.

5.1.1 Transforming the problem to apply LIME and RDE

To apply LIME and RDE we cannot trivially use the data in X_T as inputs to return a feature importance in X.
For LIME this is because it is specifically designed to use super-pixels, or more generally interpretable components, of $x \in X$. In the specific case of image classification, the method does not allow negative inputs, with the "lowest" color in the image being black.
The RDE method can be changed to use inputs in X_T, but will however also return a feature importance in X_T when doing so. Changing the feature importance by itself from X_T to X is a nontrivial task, as the feature importance does not simply represent a regular vector in X_T that can be transformed.
Realizing the above complications, one approach would be to try and change the methods themselves, finding a way to efficiently use transformed data and transform feature importances back from X_T to X. However, it is not our intention to further investigate and experiment with LIME and RDE, we simply care about their original results.

We thus choose a different approach, and rather work with the untransformed data X when applying LIME and RDE. However, for generality we assume that the model f in fact *does* use transformed data.[25] This means we need to make a few small model adjustments before being able to apply LIME and RDE.
For RDE we redefine the model by

$$f_{\text{RDE}}(x) := f(Tx) \text{ for } x \in X, \tag{58}$$

thus allowing us to effectively explain $f_{\text{RDE}} : X \to [0,1]^N$, rather than the original model $f : X_T \to [0,1]^N$ itself.
Similarly, for LIME we redefine a separate model by

$$f_{\text{LIME}}(x) := f(T(x - x_{\text{mean}})) \text{ for } x \in X, \tag{59}$$

also taking care of inverting the data normalization in this case.
Because both these models have inputs in X, the resulting feature transformation is necessarily an element of X as well. Having shown that, for an initial model $f : X_T \to [0,1]^N$, feature importances can eventually be outputted in X, we will from now on write $F(x) \in X$ to represent a feature importance result, regardless of whether $x \in X_T$ or $x \in X$.

[25] Recall that the Bayesian neural network has been trained on normalized data (see Section 4.3.1) after which its architecture has been changed by placing T^{-1} before the first layer.

5.1.2 From feature importance to explanation vector

Apart from the data preparation being similar to that of the EVD method, and the model preparation discussed for LIME and RDE above, to apply the FITEV method we also again need a neutral point $\bar{x} \in X_T$. This point can be found following the same procedure as described in Section 4.2.2.
Having finalized our set-up, the first insight towards creating explanation vectors comes from using $F(x)$ to divide each coordinate of $(T^{-1}(x - \bar{x})) \in [0, 1]^d$ into two quantities, one representing the fraction of this direction that is important, and the other representing the rest. We assume the explanation to be the important part, and let

$$T^{-1}e_i[x] = F(x) \odot (T^{-1}(x - \bar{x})). \tag{60}$$

This defines the orientation of the explanation vector that is properly contained in $T^{-1}(x - \bar{x})$. However, we still need to find the general part from which the explanation vector originates.
Of course we could select the remaining quantity $\mathbf{1}_d - F(x)$ to fill this gap, but the most general approach would be to also take the details $d_i[x]$ into account. We therefore divide the remainder (of $T^{-1}(x - \bar{x})$) over $d_i[x]$ and $g_i[x]$, using a variable $\zeta \in [0, 1]^d$, by setting

$$T^{-1}g_i[x] = \zeta \odot (1 - F(x)) \odot T^{-1}(x - \bar{x}), \tag{61}$$

or, equivalently, as the totals must sum up,

$$T^{-1}d_i[x] = (1 - \zeta) \odot (1 - F(x)) \odot T^{-1}(x - \bar{x}). \tag{62}$$

The idea is now to find ζ by optimizing for explanation strength of $e_i[x]$. Of course allowing optimization of ζ for explanation strength is only fair, as the notion of a general and detailed part in the decomposition is not incorporated in the (more binary) concept of feature importance.
However, although ζ allows vector translation of $e_i[x]$ towards a different support point, it does not allow rotations or scaling. This causes the explanation length to be fixed. We thus propose to take the idea of making FITEV fairer (and more flexible) a step further, by adding another degree of freedom. We define a variable $\omega \in [-w, w]$, for some parameter $w \in [0, 1]$ that the user is free to decide, by which the original feature importance $F(x)$ can be changed, i.e., letting the *effective feature importance* be given by

$$F_\omega(x) = \max(\min(F(x) + \omega, 1), 0). \tag{63}$$

The parameter w can be interpreted as the allowed error margin in the original discovered $F(x)$. Of course we do not mean to imply the feature importances returned by RDE and LIME are intrinsically wrong, but if we hypothesise that the intent was to find the elements of the explanation $T^{-1}e_i[x]$ we do allow for some discrepancy, which is simply caused by the methods not being optimally suited for that task.

Assuming a neutral point $\bar{x}$ has been found, we can now create an explanatory decomposition according to definition 4.4, with reasonably allowed flexibility. Summarizing all the above steps, we now define the FITEV method as follows.

1. Normalize and transform the data using T, to create a new data space X_T, following the approach of Section 4.1. Train a model $f : X_T \to [0,1]^N$ on this transformed data.

2. If a feature importance has not yet been obtained, compute the feature importance $F(x)$ of an input element $x \in X_T$, possibly using an altered model that is based on f but has its domain in X.

3. Compute a neutral point $\bar{x}$ in X_T, using batch stochastic gradient descent to solve (53). Compute also the average distance of training elements to this point, $\mu(\bar{x})$.

4. Define an explanatory decomposition as follows. Choose a $w \in [0,1]$ and let $\omega \in [-w, w]^d$. Let the effective feature importance be given by

$$F_\omega(x) = \max(\min(F(x) + \omega, 1), 0), \tag{64}$$

   and let $\zeta \in [0,1]^d$. Following definition 4.4, let

$$\begin{aligned} T^{-1}e_i[x] &= F_\omega(x) \odot T^{-1}(x - \bar{x}), & (65)\\ T^{-1}g_i[x] &= \zeta \odot (1 - F_\omega(x)) \odot T^{-1}(x - \bar{x}), & (66) \end{aligned}$$

   from which we see that consequently

$$T^{-1}d_i[x] = (1 - \zeta) \odot (1 - F_\omega(x)) \odot T^{-1}(x - \bar{x}). \tag{67}$$

5. Optimize the decomposition by maximizing explanation strength, as defined in (50), only now with respect to the parameters ω and ζ.
   Note that selecting a small w allows very little freedom, while selecting $w = 1$ makes FITEV identical to EVD.

5.2 Implementation and results

Using a data set of 30 images we run the LIME, RDE and EVD methods. For the LIME and RDE methods we apply FITEV with a window size of $w = 0.1$ to generate explanations. The optimization takes less than half a minute to converge. Both in the FITEV and the EVD optimization for explanation strength we select $\lambda = 0.1$, although it is unlikely that the constraint is necessary when applying FITEV, where the explanation vector length is more likely restricted by the variable $\omega \in [-0.1, 0.1]$. A selection of the resulting images, i.e., feature importances as well as the corresponding explanation vectors obtained using FITEV, is shown in Figures 32 and 33.

By computing averages on the data set we obtain the following results.

- The LIME method achieves the lowest explanation strength of 2.3 ± 0.9. In agreement with our intuition, this is likely caused by the large and widely varying explanation lengths of 0.3 ± 0.3. Impacts are relatively more stable at 0.5 ± 0.2.
  However, the method runs fast, taking less than 2 minutes per input element, which includes the computation of both the feature importance and the explanation vector.
- The RDE method achieves a slightly higher explanation strength of 3.2 ± 1.3. It is caused by both a lower explanation length of 0.21 ± 0.11 and a higher impact of 0.6 ± 0.2. However, computing the feature importances takes about 4 minutes per element.
- For proper reference, we also mention that the EVD method achieves an average explanation strength of 8.1 ± 0.7 on this data set. Admittedly this high value is for the main part caused by the low explanation length of 0.100 ± 0.001, although the impact is also higher: 0.81 ± 0.07.

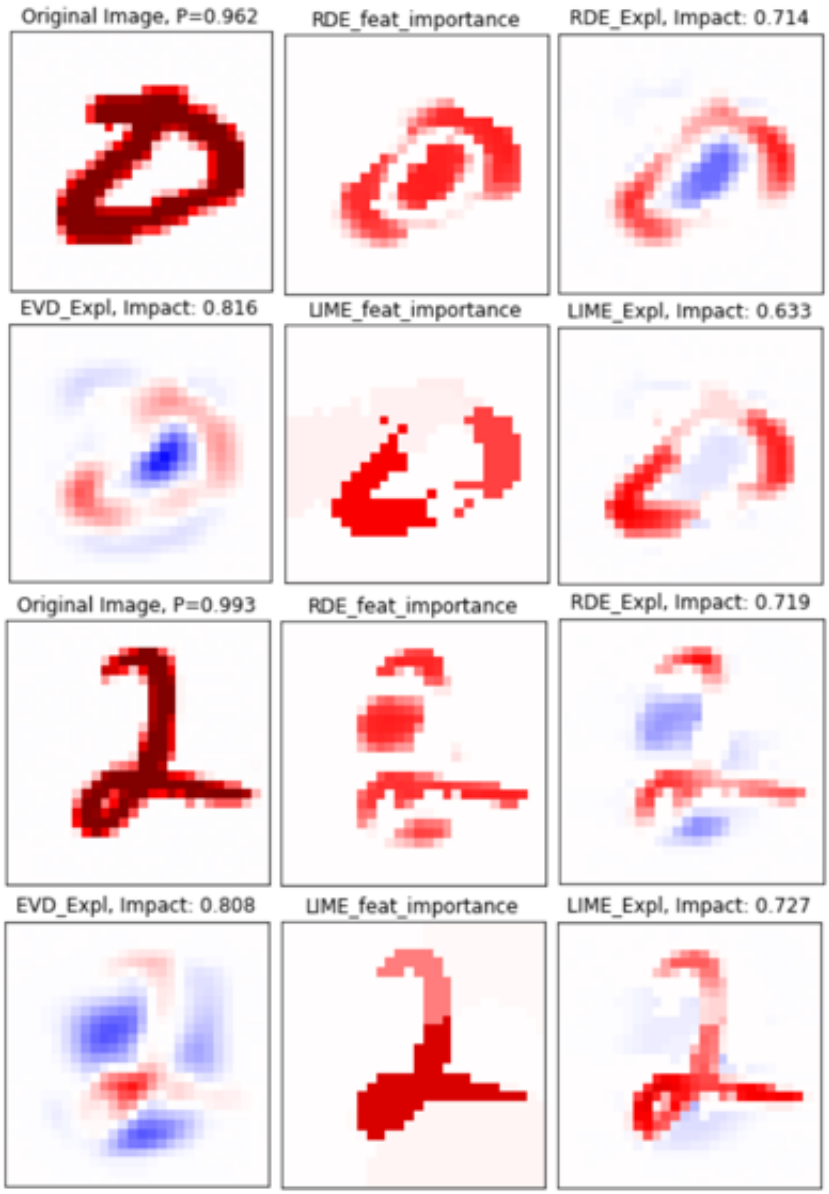


Figure 32: Feature importance results of RDE and LIME, their corresponding explanation vectors computed using FITEV, and the directly computed explanation vector of EVD are depicted.

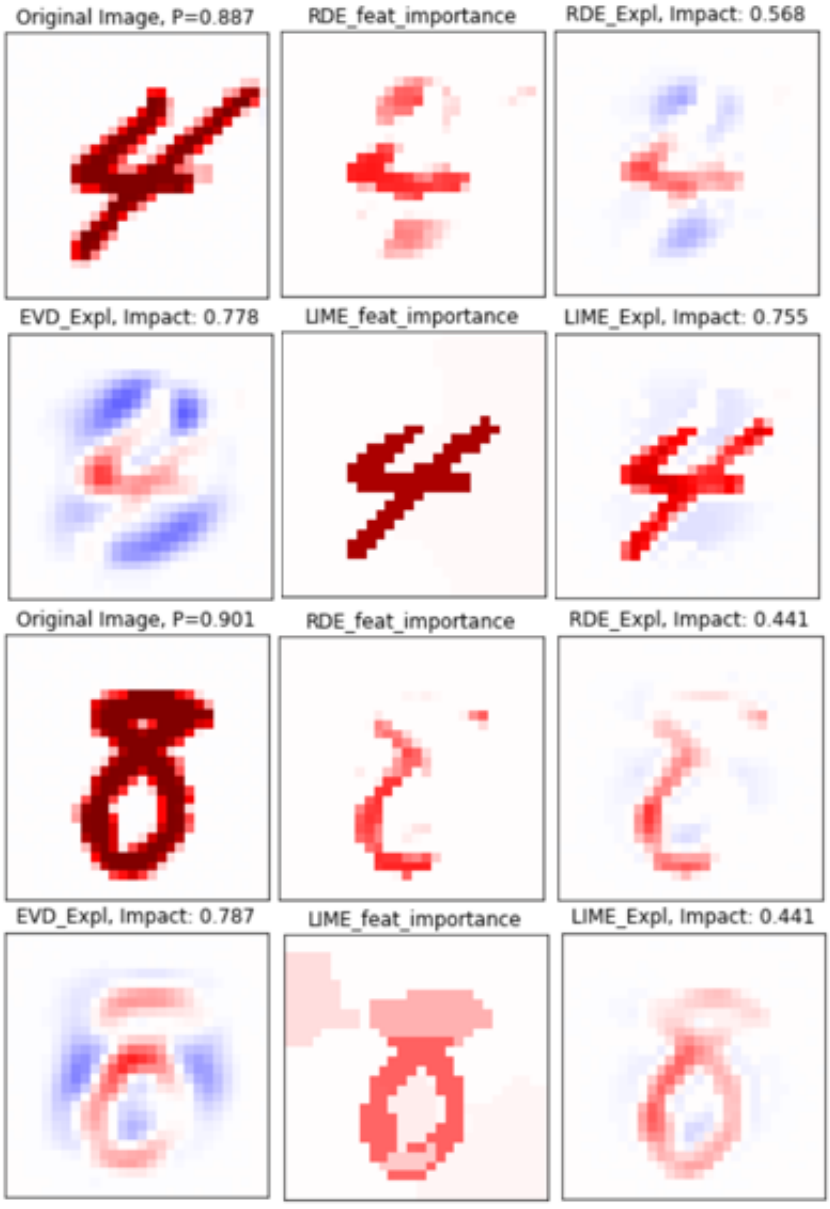


Figure 33: Feature importance results of RDE and LIME, their corresponding explanation vectors computed using FITEV, and the directly computed explanation vector of EVD are depicted.

5.3 Discussion and concluding remarks

Having already compared RDE and LIME qualitatively, we can now point out that the results discovered by the FITEV comparison support our earlier findings. The LIME method indeed seems less effective than the RDE method in explaining images of the MNIST data set. What's more, our intuition as to *why* that was the case, namely because LIME is too much inclined to simply present the entire digit as an explanation, is *also* supported by FITEV, through the definition of explanation strength: the images of the LIME method had larger explanation vector lengths, causing relatively low average explanation strength. The EVD method is shown to outperform both LIME and RDE, although the comparison is arguably unfair, because it has been specifically designed to maximize explanation strength, and has the most freedom to do so.
Going over our precise implementation of FITEV, we do acknowledge possible improvements as well as reasons to restrain ourselves from drawing too direct conclusions from these results.

Firstly we may not have compared the best possible implementations of LIME and RDE, and implementation changes could lead to very different results. We therefore mainly want to point out that the results hold for *these specific* implementations, and that FITEV is indeed successful in comparing two different methods. Continuing this line of thought, a method like FITEV, when properly adapted and perfected, could even be used to train and optimize hyperparameters of other explainability methods.
The definition of explanation strength need not only be used to compare, but can also be used to improve other explainability methods, that may not have explanation strength as their prime goal, but would nonetheless aspire higher strengths.

On a more technical note on the method itself, we could opt for making the window size w different in each dimension, an idea we have not further investigated to keep the method relatively straight-forward.
We also realize that the choice of $w = 0.1$ has been rather arbitrary. Indeed, the best comparison would follow from computing the average explanation strengths for a wide range of choices of w, and comparing the resulting graphs. Perhaps one method can been aided a lot (in maximizing explanation strength) by a relatively small window $w = 0.1$, while another method's results only significantly improve in a higher regime of $w \in [0.2, 0.3]$. However, given that FITEV relatively quickly starts to resemble EVD as w gets higher, the region of interest can be assumed to be small and close to 0.

6 Conclusion

In this thesis we have thoroughly explained the neural network and the Bayesian neural network. We have implemented both, and demonstrated the latter's ability to more intuitively cope with uncertainties.
Diving deeper into the research field of explainability methods in machine learning, describing, implementing and comparing two such methods (LIME and RDE), we noticed the lack of a proper defininition for an explanation in this context.

This has led us to come up with such a definition of our own, as well as the concept of explanation strength. To compute explanations and optimize their strength, we put forward the Explanatory Vector Decomposition method.
Not only does the method produce explanations for neural network decisions in a flexible way, with the added benefit of giving quantitative results, the newly defined concepts also enable objective comparisons between other explainability methods.

For this purpose we have proposed the FITEV (Feature Importance To Explanation Vector) method. We have compared the three implemented methods, i.e., LIME, RDE and EVD, based on achieved explanation strengths. We find that the objective results of FITEV in this comparison support our earlier subjective insights obtained when qualitatively comparing RDE and LIME.
Naturally, as it has been designed specifically for the optimization of explanation strength, the EVD method outperforms both RDE and LIME according to this measure.

Appendices

A l^2 regularization

We show that using $L2$ regularization with a regularization parameter $\alpha > 0$ is in fact equivalent to inducing a prior distribution on the model parameters. Specifically, with the new cost function given by

$$C'(S,p) = C(S,p) + \alpha \sum_i p_i^2, \tag{68}$$

the prior distribution on the parameters is implicitly given by

$$p_i \sim \mathcal{N}\left(0, \frac{1}{2\alpha}\right). \tag{69}$$

To realize this, note that from this distribution we find that

$$\mathbb{P}(p_i) \propto e^{-\alpha p_i^2},$$

and thus for the full set of (by assumption independent) parameters we have

$$\begin{aligned} \mathbb{P}(p) &\propto e^{-\alpha \sum_i p_i^2}, \\ \log(\mathbb{P}(p)) &\propto -\alpha \sum_i p_i^2. \end{aligned}$$

We conclude that we get the cost function stated in (68) from

$$\begin{aligned} \arg\max_p \mathbb{P}(S|p)\mathbb{P}(p) &= \arg\max_p e^{-C(S,p)} e^{-\alpha \sum_i p_i^2} \\ &= \arg\max_p -C(S,p) - \alpha \sum_i p_i^2 \\ &= \arg\min_p C(S,p) + \alpha \sum_i p_i^2 \\ &= \arg\min_p C'(S,p). \end{aligned}$$

B KL divergence between gaussians

Let $Q_\beta(p)$ be a Gaussian distribution with mean $\mu \in \mathbb{R}^P$ and variance $\sigma \odot \sigma I$ for $\sigma \in \mathbb{R}^P$, and $\mathbb{P}(p)$ be given by $\mathcal{N}(0, I)$. Then

$$\begin{aligned} \int Q_\beta(p) \log(\mathbb{P}(p)) dp &= \int \mathcal{N}(p; \mu, \sigma \odot \sigma I) \log \mathcal{N}(p; \mathbf{0}_P, I) dp \\ &= -\frac{P}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{P}(\mu_j^2 + \sigma_j^2). \end{aligned}$$

Also we have

$$\begin{aligned}\int Q_\beta(p)\log(Q_\beta(p))dp &= \int \mathcal{N}(p;\mu,\sigma\odot\sigma I)\log(\mathcal{N}(p;\mu,\sigma\odot\sigma I))dp \\ &= -\frac{P}{2}\log(2\pi)-\frac{1}{2}\sum_{j=1}^{P}(1+\log(\sigma_j^2).\end{aligned}$$

Combining the two, we find that

$$\begin{aligned}D_{\mathrm{KL}}(Q_\beta(p)\parallel\mathbb{P}(p)) &= \int Q_\beta(p)(\log(Q_\beta(p))-\log(\mathbb{P}(p)))dp \\ &= \frac{1}{2}\sum_{j=1}^{P}(\mu_j^2+\sigma_j^2-\log(\sigma_j^2)-1).\end{aligned}$$

C Decorrelation of the training data

Let the (normalized) training set S_X have an estimated covariance of $\mathrm{Cov}(x) = C$. We seek a coordinate transformation matrix T such that $\mathrm{Cov}(Tx) = I$. Let C be diagonalized as $C = PDP^{-1}$ with P a matrix with columns equal to orthonormal eigenvectors[26] of C, and D the corresponding eigenvalues. Then

$$\begin{aligned}\mathrm{Cov}(D^{-\frac{1}{2}}P^{-1}x) &= D^{-\frac{1}{2}}P^{-1}\mathrm{Cov}(x)(P^{-1})^T(D^{-\frac{1}{2}})^T \\ &= D^{-\frac{1}{2}}P^{-1}C(P^{-1})^T(D^{-\frac{1}{2}})^T \\ &= D^{-\frac{1}{2}}P^{-1}PDP^{-1}(P^T)^{-1}(D^{-\frac{1}{2}})^T \\ &= D^{-\frac{1}{2}}D(P^TP)^{-1}(D^{-\frac{1}{2}})^T \\ &= I,\end{aligned}$$

using in the last step that $P^TP = I$, as the eigenvectors are orthonormal. We thus conlude that $T = D^{-\frac{1}{2}}P^{-1}$ does the job. Noting that the columns of P are orthonormal eigenvectors of C, and P is thus an orthogonal matrix, we can also write the transformation as

$$T = D^{-\frac{1}{2}}P^T, \tag{70}$$

which is more intuitive, as P^T now is the matrix with row vectors represented by the orthonormal eigenvectors of C. This shows that applying Tx comes down to first projecting x to each dimension of the new space, before dividing that dimension by its corresponding standard deviation.
Reducing the dimensionality of the new space X_T, e.g. from d to d_T, comes

[26]The eigenvalues need to be different for arbitrarily computed eigenvectors to be truly orthogonal, but it is unlikely for a large estimated covariance matrix that this is not the case. Either way the same steps as in principle component analysis (without reducing dimensions) can be used to find orthogonal eigenvectors.

down to removing $d - d_T$ selected columns from P as well as removing the corresponding rows and columns from D. Note that the inverse,

$$T^{-1} = PD^{\frac{1}{2}}, \tag{71}$$

still maps $\mathbb{R}^{d_T}$ to $\mathbb{R}^d$, as it should.

D Pseudocode of the EVD optimization

Input: data transformation matrix T, transformed data point $x \in X_T$, neutral point $\bar{x} \in X_T$, label $i \in [N]$ and number of optimization steps n.
Set $\alpha = 0 \cdot \mathbf{1}_{d_T}$
Set $\beta = \mathbf{1}_{d_T}$
Set $g_i[x](\alpha) = T\alpha \odot T^{-1}(x - \bar{x})$
Set $e_i[x](\alpha, \beta) = T\beta \odot T^{-1}(x - (\bar{x} - g_i[x](\alpha)))$
Set initial loss $L(\alpha, \beta) = -\frac{f_i(\bar{x}+g_i[x]+e_i[x]) - f_i(\bar{x}+g_i[x])}{||e_i[x]||}$
for step $= 1, \ldots, n$ **do**
    Update α, β simultaneously using the Adam optimization step and the derivative $\nabla_{\alpha,\beta} L(\alpha, \beta)$.
    Project both α and β back to the interval $[0, 1]^{d_T}$.
end for
Return $g_i[x](\alpha)$ and $e_i[x](\alpha, \beta)$.

References

[KL51] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. ISSN: 00034851. URL: http://www.jstor.org/stable/2236703.

[FC54] B. Farley and W. Clark. "Simulation of self-organizing systems by digital computer". In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (Sept. 1954), pp. 76–84. ISSN: 2168-2704. DOI: 10.1109/TIT.1954.1057468.

[Iva70] AG Ivakhnenko. "Heuristic self-organization in problems of engineering cybernetics". In: *Automatica* 6.2 (1970), pp. 207–219.

[Hol73] Paul W Holland. "Weighted Ridge Regression: Combining Ridge and Robust Regression Methods". In: Working Paper Series 11 (Sept. 1973). DOI: 10.3386/w0011. URL: http://www.nber.org/papers/w0011.

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (1989), pp. 303–314. URL: http://dx.doi.org/10.1007/BF02551274.

[Hor91] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

[Bis97] Christopher M. Bishop. "Bayesian Neural Networks". en. In: *Journal of the Brazilian Computer Society* 4 (July 1997). ISSN: 0104-6500. URL: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65001997000200006&nrm=iso.

[Car+99] Rich Caruana et al. "Case-based explanation of non-case-based learning methods". In: *Proceedings / AMIA Annual Symposium. AMIA Symposium* (Feb. 1999), pp. 212–5.

[FH99] Brendan Frey and Geoffrey Hinton. "Variational Learning in Nonlinear Gaussian Belief Networks". In: *Neural computation* 11 (Feb. 1999), pp. 193–213. DOI: 10.1162/089976699300016872.

[Min01] Thomas P. Minka. "A Family of Algorithms for Approximate Bayesian Inference". In: (2001). AAI0803033.

[VS08] Andrea Vedaldi and Stefano Soatto. *Quick Shift and Kernel Methods for Mode Seeking*. 2008.

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12 (July 2011), pp. 2121–2159. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2021068.

[Gra11] Alex Graves. "Practical Variational Inference for Neural Networks". In: (2011). Ed. by J. Shawe-Taylor et al., pp. 2348–2356. URL: http://papers.nips.cc/paper/4329-practical-variational-inference-for-neural-networks.pdf.

[PBJ12] John Paisley, David Blei, and Michael Jordan. "Variational Bayesian Inference with Stochastic Search". In: (2012). arXiv: 1206.6430 [cs.LG].

[TH12] Tijmen Tieleman and Geoffrey Hinton. "Divide the gradient by a running average of its recent magnitude". In: *Coursera: Neural Networks for Machine Learning* (2012).

[BS13] Pierre Baldi and Peter J Sadowski. "Understanding dropout". In: (2013), pp. 2814–2822.

[KW13] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: (2013). arXiv: 1312.6114 [stat.ML].

[LCB13] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. "MNIST handwritten digit database". In: (2013). URL: http://yann.lecun.com/exdb/mnist/.

[Mik+13] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and Their Compositionality". In: NIPS'13 (2013), pp. 3111–3119. URL: http://dl.acm.org/citation.cfm?id=2999792.2999959.

[PMB13] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. *On the number of response regions of deep feed forward networks with piece-wise linear activations*. 2013. arXiv: 1312.6098 [cs.LG].

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: (2013). arXiv: 1312.6034 [cs.CV].

[Sze+13] Christian Szegedy et al. "Intriguing properties of neural networks". In: (2013). arXiv: 1312.6199 [cs.CV].

[BS14] Pierre Baldi and Peter J. Sadowski. "The dropout learning algorithm". In: *Artificial intelligence* 210 (2014), pp. 78–122.

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). arXiv: 1412.6980 [cs.LG].

[Mon+14] Guido F Montufar et al. "On the Number of Linear Regions of Deep Neural Networks". In: *Advances in Neural Information Processing Systems 27* (2014). Ed. by Z. Ghahramani et al., pp. 2924–2932. URL: http://papers.nips.cc/paper/5422-on-the-number-of-linear-regions-of-deep-neural-networks.pdf.

[Sch14] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *CoRR* abs/1404.7828 (2014). arXiv: 1404.7828. URL: http://arxiv.org/abs/1404.7828.

[Blu+15] Charles Blundell et al. "Weight Uncertainty in Neural Networks". In: (2015). arXiv: 1505.05424 [stat.ML].

[He+15a] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[He+15b] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[MOT15] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. "Inceptionism: Going Deeper into Neural Networks". In: (2015). URL: https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

[TGR15] Andrew Trask, David Gilmore, and Matthew Russell. "Modeling Order in Neural Word Embeddings at Scale". In: *CoRR* abs/1506.02338 (2015). arXiv: 1506.02338. URL: http://arxiv.org/abs/1506.02338.

[WFL15] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[LMJ16] Jiwei Li, Will Monroe, and Dan Jurafsky. "Understanding Neural Networks through Representation Erasure". In: *CoRR* abs/1612.08220

(2016). arXiv: 1612.08220. URL: http://arxiv.org/abs/1612.08220.

[Lip16] Zachary Chase Lipton. "The Mythos of Model Interpretability". In: *CoRR* abs/1606.03490 (2016). arXiv: 1606.03490. URL: http://arxiv.org/abs/1606.03490.

[RSG16] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *CoRR* abs/1602.04938 (2016). arXiv: 1602.04938. URL: http://arxiv.org/abs/1602.04938.

[Tel16] Matus Telgarsky. "Benefits of depth in neural networks". In: *CoRR* abs/1602.04485 (2016). arXiv: 1602.04485. URL: http://arxiv.org/abs/1602.04485.

[Kai+17] Lukasz Kaiser et al. "One Model To Learn Them All". In: *CoRR* abs/1706.05137 (2017). arXiv: 1706.05137. URL: http://arxiv.org/abs/1706.05137.

[Lu+17] Zhou Lu et al. "The Expressive Power of Neural Networks: A View from the Width". In: *Advances in Neural Information Processing Systems 30* (2017). Ed. by I. Guyon et al., pp. 6231–6239. URL: http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf.

[STR17] Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. *Bounding and Counting Linear Regions of Deep Neural Networks*. 2017. arXiv: 1711.02114 [cs.LG].

[Sha+17] Noam Shazeer et al. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer". In: *CoRR* abs/1701.06538 (2017). arXiv: 1701.06538. URL: http://arxiv.org/abs/1701.06538.

[Zav17] Maksym Zavershynskyi. "MSE and Bias-Variance Decomposition". In: (2017). URL: https://towardsdatascience.com/mse-and-bias-variance-decomposition-77449dd2ff55.

[AB18] A. Adadi and M. Berrada. "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)". In: *IEEE Access* 6 (2018), pp. 52138–52160. DOI: 10.1109/ACCESS.2018.2870052.

[RMF18] Simone Rossi, Pietro Michiardi, and Maurizio Filippone. *Good Initializations of Variational Bayes for Deep Models*. 2018. arXiv: 1810.08083 [stat.ML].

[Wen+18] Yeming Wen et al. "Flipout: Efficient Pseudo-Independent Weight Perturbations on Mini-Batches". In: *CoRR* abs/1803.04386 (2018). arXiv: 1803.04386. URL: http://arxiv.org/abs/1803.04386.

[Mac+19] Jan MacDonald et al. "A Rate-Distortion Framework for Explaining Neural Network Decisions". In: *CoRR* abs/1905.11092 (2019). arXiv: 1905.11092. URL: http://arxiv.org/abs/1905.11092.

[Nil+19] Geir K. Nilsen et al. "Efficient Computation of Hessian Matrices in TensorFlow". In: (2019). arXiv: 1905.05559 [cs.LG].

[YZS19] Zebin Yang, Aijun Zhang, and Agus Sudjianto. “Enhancing Explainability of Neural Networks through Architecture Constraints”. In: (2019). arXiv: 1901.03838 [stat.ML].