Matthias Anton Freiberger

Training Activation Functions In Deep Neural Networks

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme

Telematik

submitted to

Graz University of Technology

Supervisor

Prof. Dr. Horst Bischof

Institute for Computer Graphics and Vision

Dipl.-Ing. Gernot Riegler

Dr. Samuel Schulter

Graz, Austria, Dec. 2015

To my parents, who made it possible.

Abstract

Image recognition is considered one of the most challenging tasks in the field of computer vision. Recently though, convolutional neural networks (CNNs) show excellent results on several recognition data sets. Methods, which train not only the weights between neurons, but also the activation functions of a CNN, show currently the best performance. Nevertheless, these approaches usually train solely steepness parameters for one or several rectifier units, or enforce hard constraints on the shape of their activation functions. In this thesis we present a framework to train a more general family of activation functions that are more expressive and investigate if this way even better performance can be reached. We do so by learning the parameters of a sum of arbitrary base functions. Furthermore, we constrain the optimization process of the parameters in a sensible way to reduce the complexity of our optimization problem as well as keep the number of parameters low. Using our framework, we approach the performance of state-of-the-art methods and outperform rectifier units on three different data sets using two different network architectures. Nevertheless, we find that the range of suitable base functions when training deep structures is confined to functions with largely constant gradients. Therefore, it seems advisable to further pursue the approach of trainable parameters for one or several rectifier-like units in combination with the techniques shown in this thesis, in order to obtain even better performance on state-of-the-art recognition problems.

Kurzfassung

Bilderkennung gilt als eines der schwiergsten Probleme im Bereich des Maschinellen Sehens. Seit kurzem werden jedoch mit Convolutional Neural Networks (CNNs) ausgezeichnete Ergebnisse auf einigen Bilderkennungs-Datensätzen erzielt. Dabei liefern Methoden, die nicht nur die Gewichte zwischen den Neuronen des Netzes, sondern auch die Aktivierungsfunktionen der Neuronen lernen, die besten Ergebnisse. Dennoch trainieren diese Ansätze üblicherweise nur Steigungsparameter von rektifizierten Lineareinheiten (rectified linear units, ReLUs), oder beschränken die Form, die die trainierten Aktivierungsfunktionen annehmen können auf andere Art und Weise. In dieser Arbeit präsentieren wir ein Modell um eine generellere Familie von ausdrucksstärkeren Aktivierungsfunktionen zu trainieren, und untersuchen, ob auf diese Weise noch bessere Ergebnisse erzielt werden können. Dies tun wir, indem wir die Parameter einer Summe von beliebigen Basisfunktionen lernen. Desweiteren setzen wir Rahmenbedingungen für die Optimierung unserer Parameter, um einerseits die Komplexität des Optimierungsproblemes reduzieren, und andererseits die Anzahl der freien Parameter niedrig zu halten. Mit Hilfe unseres Modelles nähern wir uns dem aktuellen Gold-Standard und schlagen ReLUs auf drei unterschiedlichen Datensätzen, wobei wir zwei verschiedene Netzarchitekturen trainieren. Dennoch kommen wir zum Ergebnis, dass die Auswahl an geeigneten Basisfunktionen auf Funktionen, welche konstante Gradienten über große Teile des Definitionsbereiches aufweisen, beschränkt ist. Daher scheint es ratsam, den Ansatz trainierbarer Parameter für eine oder mehrere ReLU-artige Funktionen, in Kombination mit den in dieser Arbeit vorgestellten Techniken, weiter zu verfolgen.

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

The text document uploaded to TUGRAZonline is identical to the presented master's thesis dissertation.

Place	Date	Signature

Acknowledgments

We live in exciting times, generally speaking, and also concerning the most recent breakthroughs and discoveries in the field of computer vision. Convolutional neural networks enable us to reach out towards human-level performance in computer vision for the first time. I would like to thank my supervisor, Prof. Horst Bischof, for granting me the opportunity to do research in this exciting field within my master's thesis, as well as providing the necessary resources. Furthermore, I would like to thank my advisors, Gernot Riegler and Samuel Schulter, who originally proposed the topic of this thesis, for always taking their time, many fruitful discussions and ideas, as well as the quick correction of my drafts. Moreover I'd like to thank Samuel Schulter and Georg Poier for sharing their office with me while this thesis was in the making. During my studies as well as the creation of this thesis, many people accompanied me on my path and I would like to thank all of them: Thank you, your company has meant the world to me. In place for all of them, I would like to mention one special person in particular: My girlfriend Melissa Farasyn, whose smile seems to lighten up even the gloomiest winter nights, and who demonstrated deep knowledge of the english language as well as a very sharp eye when proofreading this thesis. Thank you so much. Finally, I would like to express my deepest, and most sincere gratitude to my parents, Gertraud and Anton Freiberger, for granting me the opportunity to pursue a higher education, and for their neverending love and support.

You will always be in my heart, thank you a thousand times.

Contents

List of Figures

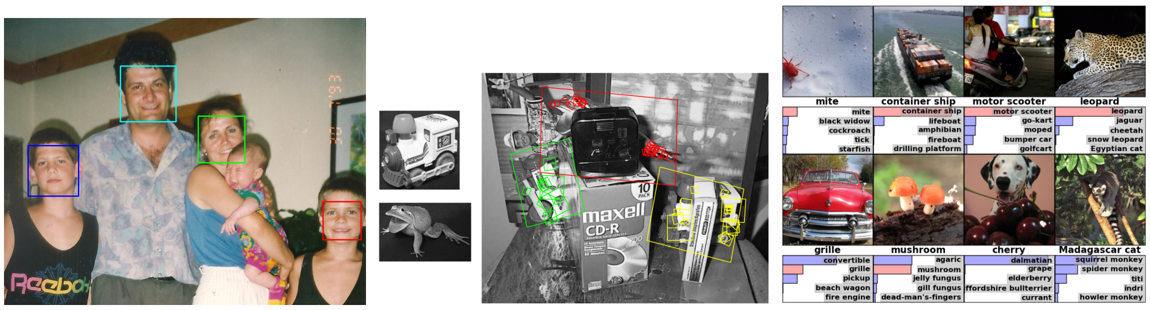
List of Tables

1 Introduction

Implementing image recognition in computer systems, i.e. giving computer systems the ability to identify real-world entities in images and extract information about their state and relations to each other, bears a manifold of real-world applications from pedestrian detection [15, 18, 50, 56, 59], over face detection/recognition [23, 24, 28, 43, 66, 68] and textual image description [20, 21, 35, 73], towards a multitude of consumer applications such as intelligent photo sorting and editing [2, 61, 61, 72] and image search [32, 34, 39, 70]. Therefore, the problem of image recognition has been researched for decades [23], nevertheless only recently He et al. made an outreach towards human performance on a large dataset [27]. As a consequence, recognition is still considered one of the biggest challenges in the field of computer vision [65]. The difficulty of recognition can be attributed to several reasons.

First of all, real world objects usually exhibit a very high intra-class variation. Take for instance a person who is given the task to identify a chair in a given set of images. This person might come up with kitchen chairs, office chairs, and armchairs, which all have a different visual appearance. Some people might even point out sofas and beanbags, which also exhibit different visual appearances. On the other hand it is also possible to sit on a table or cupboard, although many people would not point them out given the above task, because they are not accustomed to sit on them. Even more complicated, some classes are subclasses of one or more classes such as a comfy chair, which might classify as a chair and also as a couch [17]. Thus human class definitions often do not rely on visual appearance alone, but machines lack the prior knowledge humans use to recognize a chair.

Furthermore, the amount of classes humans commonly distinguish is very high, according to a recent publication [16] about 100000. Additionally, the visual appearance of objects in images varies due to conditions in the real world. Objects might be displayed in various scales and viewpoints in an image as well as assume a multitude of different poses. Furthermore, while a single two-dimensional image is static, the three-dimensional world is not. Not all entities we want to detect in images are rigid objects, they move through the world and morph their appearance constantly, such as pedestrians bending limbs, or faces making expressions. Also we do not want to constrain image recognition to images where single objects remain in solitude in front of a uniform background. Entities in the real world interact with each other all the time, so most images we want to extract information from will be full of entities and things that occlude each other, or are merged or somehow morphed in a way we did not see before. While humans easily deal with all these variations, no concrete algorithm has been found yet to implement the task of recognition in a computer system. In fact, all the different tasks which are commonly considered as recognition seem way to inhomogeneous themselves to implement them using a single universal algorithm. Szeliski [65] categorizes recognition into the following subclasses: In case we are looking for a specific class of objects in an image, we call the corresponding problem one of object detection, for instance face detection, or pedestrian detection. If, on the other hand, we are looking for one specific instance of a rigid object shown in various perspectives in a possibly cluttered scene, we talk of instance recognition. Finally, general class or category recognition is considered the most challenging subtask: We want to recognize instances of often highly variant classes such as animals, or fruits and vegetables.



(a) Object detection. **(b)** Instance recognition. **(c)** Category recognition.

Figure 1.1: (a) Face detection using Haar Wavelets [68] as an example for object detection, the algorithm scans the image looking for patterns matching a face looking frontal towards the camera. (b) Instance recognition of train and frog toys by extracting descriptive keypoints and scanning for these keypoints in a valid geometric alignment in the scene. Method and image from [47]. (c) Category recognition of various images of animals, vehicles etc.. The algorithm shows the 5 most likely labels associated to the image along with a confidence value. Method and figure from [38].

Since the task of recognition is too diverse and too complex to implement it through a single algorithm, many approaches proposed in the last decade rely on supervised machine learning methods (e.g. [15, 22, 54, 68]). For this approach, usually patches of interesting points are extracted from the input image and encoded to form a feature vector: Since images are represented in computer memory as a set of brightness values, we are able to apply mathematical transformations to them. Many of these transformations have been proposed (see [3, 4, 11, 12, 15, 44, 47, 60]), which seek to encode the features of the local patch invariant to lightning, scale, two-dimensional and partly three-dimensional rotation on the image, and thus try to make the appearances of objects to recognize invariant to variations given by the image acquisition process. The vectors which result from these transformations are called feature vectors and their corresponding space is called feature space. Now, under the assumption that a feature vector describes the essential appearance of an image patch, the location of a feature vector in feature space is an indication of what is shown on the image patch. Therefore, we expect objects of the same class to be close to each other in feature space, since they look similar.

Using supervised machine learning, we can train a classifier function that learns which subspaces of the feature space belong to certain classes using manually labeled training images. Thus, we can assign a corresponding output class for an input image by retrieving the class of the subspace it lies in. An alternative approach is to interpret the image itself as vector in a high-dimensional space. We can do so, for instance, by writing all brightness values of an image in a single column. If we use an image that has been preprocessed that way, we need to learn the transformation from the original "image vector" to a vector in the feature space jointly with the subspaces of a feature space that belong to the different image classes.

One machine learning method which trains the feature transformation jointly with the image class is a convolutional neural network (CNN) [40]. Inspired by the human brain, CNNs consist of so-called neurons, which are arranged in layers and connected through hidden connections. The output of a single neuron is determined by its non-linearactivation function. By separating the input image vectors through adjustment of the weights between neurons and repeatedly transforming the separated input image vectors into different spaces, where they are further separated, the neural network is able to learn a transformation from the input image vector space towards a feature vector space. This feature vector space then bears subspaces that can be assigned to the different classes of input images shown to the network. Activation functions play an important role in applying repeated non-linear transformations to the image vector space and thus bringing the images into a vector space where their classes can be detected. Due to very high CPU and memory requirements and their need of large amounts of training data though, until recently, different methods were preferred over CNNs apart from a few applications [40].

Recently though, CNNs have experienced a revival through the increased availability of processing power and memory. In computer vision, CNNs currently form the state-of-the-art on several machine vision data sets [25, 27, 38, 45, 62, 64]. Apart from the availability of large data sets and fast implementations of the corresponding algorithms on graphics processing units, these successes can be attributed to careful net design, methods to reduce overfitting, and newly discovered activation units [71].

We argue that one reason for the success of these newly discovered activation units [1, 25, 27] is, that they all train their own activation functions compared to regular CNNs, which solely train the weight connections between the neurons. Nevertheless, the above methods train activation functions which are constrained to ReLUs with trained varying steepness in one [27] or several [1] points, or are strictly convex piecewise activation functions. Thus, to our knowledge no general method to train continuous non-convex activation functions has been proposed yet.

Such a method might be useful though to obtain better solutions on problems solved with CNNs in computer vision on the one hand, and to gain a deeper understanding on the properties of a good activation function on the other hand. In this thesis we propose a method for training activation functions within neural networks through optimization of the parameters of a sum of arbitrary base functions. This optimization is incorporated into the regular training process. We implement our method within a unit and show that networks utilizing these units perform well when solving the image recognition problems on several standard data sets. Moreover, we briefly discuss the activation functions trained and attempt to understand what qualifies them as a good activation function.

In Chapter 2, we give a short introduction to CNNs and their application in image recognition. Furthermore we will give a brief survey about the recent development of activation functions that adapt as a part of the networks training process. Then, in Chapter 3, we introduce the idea of our proposed method, discuss our design decisions as well as the implementation of our ideas into a neuron. We follow with Chapter 4, where we analyze the behavior of our unit inside a network training process and show that it outperforms the ReLU baseline on our given network for different recognition problems. Furthermore we compare our unit with similar methods [1, 27] which currently form the State of the Art. Finally, we summarize our findings and give a brief outlook in Chapter 5.

2 Image Recognition With Neural Networks

Humans perform image recognition all the time without making a conscious effort, in contrast to, for instance, logical reasoning. Thus, one might be easily fooled to assume that it must be comparably easy to implement recognition into a computer system. In fact, according to a famous anecdote, in the 1960s, the artificial intelligence researcher Marvin Minsky asked one of his first-year undergraduate students to "spend the summer linking a camera to a computer and getting the computer to describe what it saw" [7]. Nevertheless, although image recognition has been researched for more than 40 years, computer vision systems have only recently approached human level performance [27] on a single benchmark, an event whose occurrence was still heavily disputed a few years ago [65].

But why is recognition so challenging? Among other reasons (see Chapter 1 for a more detailed description), geometric variations introduced by the image acquisition process contribute to the hardness of recognition. Objects might be displayed at different scales, in different poses, viewpoints and brightness levels, to mention a few examples. A popular way to deal with these variations is to design vector transformations (see [3, 4, 11, 12, 15, 44, 47, 60]) for images which are able to encode the appearance of objects in images independently from the geometric variations mentioned above. Then, using machine learning, a classifier function can be trained to assign a class to the transformed input images of image patches.

Recently though, an extension of the neural network machine learning algorithm, called a convolutional neural network (CNN), has shown outstanding performance on several machine vision challenges (among several others [1, 27, 37, 45, 64]). Contrary to other machine learning algorithms, convolutional neural networks are able to train a suitable feature transformation for a given set of images jointly with the corresponding classes of the image set. Thus, manually crafting a feature transformation suitable for a given recognition task is not necessary. CNNs can be applied to a wide range of recognition problems and thus generalize well.

In this Chapter, we will discuss the fundamentals of CNNs and how they can be used to perform recognition. In Section 2.1 we give a short introduction to the problem of classification, introduce its terminology and show common ways to perform classification. Thereafter, in Section 2.2, we give an overview of neural networks, their structure and function. We explain how this networks can be trained, as well as what the most common problems are and what the techniques are to avoid these. In Section 2.3, we show how neural networks can be used to perform end-to-end recognition without manually crafted feature transforms by introducing CNNs. Finally, in Section 2.4, we give an overview on work in the field which relates to this thesis.

2.1 The Problem Of Classification

One very popular approach to tackle various sub-problems of image recognition is reducing the problem of image recognition to a problem of data classification. Any image, represented digitally in a computers memory, can be seen as a collection of brightness values, spatially arranged on a regular grid in two dimensions. These collection of brightness values can be rearranged from their usual two-dimensional form into a vector $\mathbf{x}'$, for instance by writing them into a vector column by column, starting with the top most left and proceeding from top to bottom and from left to right. We transform our images $\mathbf{x}'$ in feature vectors $\mathbf{x}$

$$\mathbf{x} = f(\mathbf{x}'), \tag{2.1}$$

such that they are as invariant as possible to variations introduced through the image acquisition process. Such variations might be caused by brightness, viewpoint, pose, position, scale etc. Nevertheless we want these feature vectors to be as descriptive as possible towards the actual appearance of the object. The underlying assumption is, that feature vectors of objects with similar appearance lie close to each other in the feature space. Thus, a possible way to build a classifier is to learn which objects tend to be located in which subspaces of our feature space.

One possibility is to use a set of training images with known class labels and separate them in the feature space such that the resulting subspaces always contain only images of a certain class. We then assign this class to every new, previously unseen image, located in the corresponding subspace. A second possibility is to use unlabeled images right away and separate the feature space by grouping the images based on their location in space (e.g. close images are grouped together) and then separate the space in a way such that each group of images is contained in a single subspace. Then artificial class labels are assigned to each subspace, and new occurring images are assigned the class label of the subspace which contains them.

The former method of learning is called supervised learning, since we know the labels of our training image beforehand, and supervise the learning process by telling a given algorithm which images belong to which class. The latter method on the other hand, where we introduce artificial classes ourselves, is called unsupervised learning. A combination of the two methods is possible as well if we have small amounts of labeled images and larger amounts of unlabeled images. Since we are able to assign a true (natural) class label to a subspace, which has been learned unsupervised, if we have a single labeled image which lies in that subspace, we can assign the class label of an image to its containing subspace which we have learned using unlabeled data. This is called semi-supervised learning. While all three kinds of learning have been applied in computer vision, we will focus on supervised learning in this thesis.

For supervised learning, we need a set of images $\mathcal{X} \subset \mathcal{R}^f$, which have been previously labeled manually to indicate their true class, i.e. the class or content they show. We refer to these set of labels as $\mathcal{Y}$, where $\mathcal{Y} \subset \mathbb{N}$, and to a labeled image as a tuple $(\mathbf{x}, y)$ of the labeled data set (X, Y).

Thus we want to model a set of rules as a function

$$\hat{y} = \mathcal{F}(\mathbf{x}), \tag{2.2}$$

that is able to infer a class label estimation $\hat{y}$ from the position of $\mathbf{x}$ in feature space.

One possible approach is use a parameterized function $\mathcal{F}$, whose output value k is not only dependent on the value of $\mathbf{x}$, but also on the parameter vector $\mathbf{w}$ such that

$$\mathcal{F}(\mathbf{x}) = \mathcal{F}(\mathbf{x}; \mathbf{w}). \tag{2.3}$$

Thus we use our labeled data set (X, Y) as input to $\mathcal{F}$ to attempt to modify the parameter vector $\mathbf{w}$ such that $\mathcal{F}(\mathbf{x}^{(n)}, \mathbf{w}) = y^{(n)}$ for as many $(\mathbf{x}^{(n)}, y^{(n)})$ as possible. In a nutshell, we model a set of rules used to predict $y^{(n)}$ from the image $\mathbf{x}^{(n)}$.

The function $\mathcal{F}$ is then called a *classifier*. Nevertheless, if we build a classifier $\mathcal{F}$ as described above, how can we make sure that our classifier will generalize for previously unseen images? For all we know we might have modeled the rules to derive y from $\mathbf{x}$ too narrow and predict only the y of our known data set correctly. This effect is widely known as overfitting.

To determine whether we have trained a generally valid classifier, we separate our data set (X, Y) in three disjoint parts, namely a training set (X_{train}, Y_{train}), a test set(X_{test}, Y_{test}), as well as a validation set (X_{valid}, Y_{valid}). We use (X_{train}, Y_{train}) to train our classifier, while the test set(X_{test}, Y_{test}) is not incorporated in the training process, but used as a benchmark set, to ensure our classifier performs well on unseen data. Finally, a validation set(X_{valid}, Y_{valid}) can be used to tune parameters which do not directly influence the model, but control the training process. These parameters are called hyperparameters. The tuning of hyperparameters is not done on (X_{test}, Y_{test}) since we want to find parameters that work generally well for our classifier, and are not tailored to the test set specifically.

A question which logically follows is: How can one determine if a classifier function works well in the first place? For that cause, a second function is used on the data set as well as the result of the classifier measure, how well the classifier $\mathcal{F}$ has inferred the labels y from the vectors $\mathbf{x}_i$:

$$\mathcal{L}(X_{train}, Y_{train}) = \sum_{n=1}^{N} l(y^{(n)}, \mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w})), \tag{2.4}$$

where $l(y^{(n)}, \mathcal{F}(\mathbf{x^{(n)}}))$ is some kind of distance measure between the prediction $\mathcal{F}(\mathbf{x_n})$ and the true label y_n. $\mathcal{L}$ is called the loss of the classifier $\mathcal{F}$ on the data set $(\mathbf{x_n}, y_n)$ and $l(y^{(n)}, \mathcal{F}(\mathbf{x}; \mathbf{w}))$ is called the loss function.

2.1.1 Loss

In order to measure the performance of our classifiers, we need a mathematical measure of correctness. This kind of measure is widely called a loss function, and is applied to all tuples $(\mathbf{x}, y) \in (X_{train}, Y_{train})$:

$$\mathcal{L}(X_{train}, Y_{train}) = \sum_{n=1}^{N} l(y^{(n)}, \mathcal{F}(\mathbf{x}; \mathbf{w})) \tag{2.5}$$

where $\mathcal{F}(\mathbf{x}; \mathbf{w})$ is the output of our trained classifier to $\mathbf{x}$ and $l(y, \mathcal{F}(\mathbf{x}; \mathbf{w}))$ is a similarity measure between the estimated class $\hat{y} = \mathcal{F}(\mathbf{x}; \mathbf{w})$ and y.

For classification problems with several classes, the multinomial logistic loss function is often used, which we will use in this thesis as well. It is defined [6] as

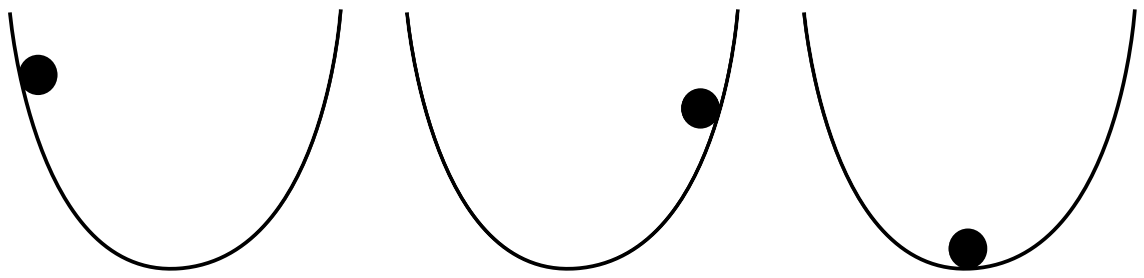
$$\begin{align} l(y^{(n)}, \mathcal{F}(\mathbf{x}; \mathbf{w})) &= -\tfrac{1}{N} \log(\hat{p}(y = l_i|\mathbf{x})) \tag{2.6} \\ &= -\tfrac{1}{N} \log(\mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w})) \tag{2.7} \end{align}$$

where $\hat{p}(y = y_k|\mathbf{x})$ indicates the probability that y has the value of the true class label y_k given the observation of $\mathbf{x}$ according to our classifier.

Note that in Eqn. (2.6) the classifier output is treated as a probability, and thus we train in this case $\hat{p}(y = l_i|\mathbf{x}) = \mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w})$. This loss is often also called cross entropy classifier loss, since $\mathcal{L}(X_{train}, Y_{train})$ is the cross entropy of the random variables $\hat{y}$ and y under the assumption, that our true label y has always the same value for a given image $\mathbf{x}$.

2.1.2 Gradient Descent

To train a classifier we modify the parameter vector $\mathbf{w}$ of a classifier function $\mathcal{F}(\mathbf{x}, \mathbf{w})$ in a way such that we minimize a given loss $L(X_{train}, Y_{train})$ on our data set. To give an intuition on how to minimize loss functions, consider the following analogy. Suppose we wanted to find the deepest point of a bowl illustrated in the sketch in Figure 2.1. A possible way to find the deepest spot, would be to put a ball on one side of our bowl. Gravity would draw our ball downwards, and, after some time, straight towards the deepest spot of the bowl where it came to lie.



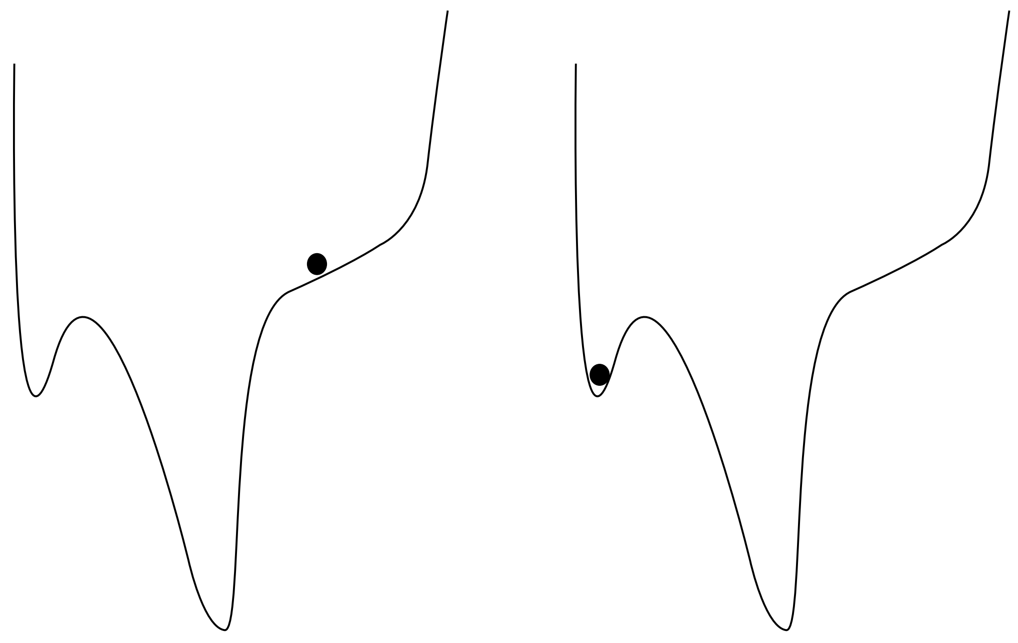
(a) Initial position on left wall. **(b)** Initial position on right wall. **(c)** Final position.

Figure 2.1: Optimization analogy: Ball placed in a bowl. No matter where the ball is set, it is always drawn to the lowest point by gravity.

Similarly we want to find a point $\mathbf{w}_{opt}$ where a loss function $l(y^{(n)}, \mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w}_t))$ has its minimum. A possible solution is to set $\mathbf{w}$ to a more or less arbitrary value on the functions input space, similarly to setting our ball somewhere into the bowl. Now similar to the ball drawn towards the bowls deepest point and therefore changing its location within a time interval Δt, we can make an attempt to take a step towards our loss functions minimum. Under the assumption that our loss function is somehow "bowl-shaped" (in general: convex [8]), we can always move towards the functions minimum by moving into the direction where the loss function decreases. Gradient is a measure of how steep a function rises in positive $\mathbf{w}$ direction. Thus we move towards the direction of the inverse sign of the gradient, which is always the direction where the function value decreases. Therefore, a possible update rule for $\mathbf{w}$ to step towards the minimum of $l(y, \mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w}_t))$ is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\partial l(y, \mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w}_t))}{\partial \mathbf{w}_t}. \tag{2.8}$$

Since our step size is proportional to the loss functions gradient, starting on the "walls" of our bowl, we will make big steps in the beginning, where the gradient is high, and smaller steps as soon as the walls of our bowl function become less steep until we slowly move



(a) Ball thriving towards deepest point. **(b)** Ball caught in a local minimum.

Figure 2.2: Optimization analogy (2): A ball placed on a odd-shaped surface. Depending on where the ball is set, it might be caught in the cavity and not proceed towards the deepest point.

towards the minimum, where we almost stop. Unfortunately, an issue arises with this kind of iterative scheme: Suppose we start on a point of our loss function, where the gradient is very steep, so that we actually overstep the minimum instead of stepping towards it. While we might be able to still find the minimum by oscillating in smaller and smaller steps around it, the opposite might happen when we encounter a gradient even larger than the one in the previous step: in this case we will climb the walls of our bowl, getting further and further from the minimum we seek to find. Therefore we adjust the step equation shown above by introducing an additional parameter to increase or decrease the size of our steps towards the minimum if necessary. The updated step equation is then

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \cdot \frac{\partial l(y^{(n)}, \mathcal{F}(\mathbf{x}^{(n)}; \mathbf{w}_t))}{\partial \mathbf{w}_t}, \tag{2.9}$$

where η is the so-called step size or learning rate, used to control how quickly we approach the minimum of the function. This method is called a gradient descent [6] and is widely used in optimization.

Since we do not want to minimize solely the loss function of a single $\mathbf{x}$, but rather the loss on our whole data set, the corresponding parameter update equation for parameter

$\mathbf{w}$ in $\mathcal{F}(\mathbf{x}^{(n)};\mathbf{w})$ is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \cdot \frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial \mathbf{w}_t} \tag{2.10}$$

$$= \mathbf{w}_t - \eta \sum_{n=1}^{N} \frac{\partial l(y^{(n)}, \mathcal{F}(\mathbf{x}^{(n)};\mathbf{w}_t))}{\partial \mathbf{w}_t}. \tag{2.11}$$

Up till now, we have only considered very simple loss functions which are smoothly bowl shaped. Nevertheless, real loss functions are much more likely to have many different bumps, and are much more oddly shaped. Consider Figure 2.2 for such a loss function transferred to a ball-surface analogy again. Using gradient descent, our "ball" might move only very slowly towards the minimum due to rather shallow region close to the minimum when placed on the right wall. Or, even worse, the ball might get stuck in the cavity shown close to the deepest point, if set on the left wall of our oddly-shaped bowl, thus not locating the deepest point of the bowl. In terms of optimization, we speak of an optimization algorithm locating a local minimum (the cavity) of a loss function instead of the global minimum (the deepest point of the bowl). Optimization methods that do not locate the globally optimal point but only converge to a locally optimal point are called local optimization methods [8]. Gradient descent is such a local optimization method, which is widely used in optimization due to its simplicity and wide applicability, but can be slow for complex loss functions [55]. Despite gradient descent cannot be guaranteed to converge towards the global optimum for a given optimization problem, extensions have been proposed to enhance the likelihood of the algorithm to do so.

For instance, note a that for a real world setup as shown in Figure 2.2, it is unlikely that the ball will get stuck in the cavity next to the ground of the odd-shaped bowl, since in a Newtonian system, the ball gains momentum when rolling down the steep walls of the bowl, and thus is able to pass shallow regions quickly and even overcome smaller bumps by thriving into the direction of the minimum, following the general trend of the bowls steepness, rather than solely the steepness at its actual position. We are also able to incorporate such a momentum term into our optimization model. To do so, we first rewrite the step equation to the form

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_t, \tag{2.12}$$

where

$$\mathbf{v}_t = -\eta \cdot \frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial \mathbf{w}_t}. \tag{2.13}$$

Now, to speed up optimization, we rewrite the update term, now always adding the update term from the previous step w_{n-1} to the current update term of the current step

$$\mathbf{v}_t = -\eta \cdot \frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial \mathbf{w}_t} + a \cdot \mathbf{v}_{t-1}, \tag{2.14}$$

where the weight a is called accordingly the momentum. By recursively adding the previous update term multiplied with a weight, we incorporate a weighted average into our current update term, which allows us to overcome local minima as well as sections with low gradient in our loss function. Indeed it has been proven [55], that the momentum term is actually the equivalent to the momentum of a moving Newtonian body in classic physics.

A second possibility [41] to speed up the optimization process using gradient descent, is the stochastic gradient descent algorithm (SGD). Using SGD, we do not train on the whole training set, but randomly take a batch X_b of b labeled images $\mathbf{x}^{(n)}, y^{(n)}$ out of the data set. Only the samples from the batch training setX_b are incorporated into the computation of loss $\mathcal{L}(X_{train}, Y_{train})_b$ for the current training step t.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \cdot \sum_{\mathbf{x} \in X_b} \frac{\partial l(y, \mathcal{F}(\mathbf{x}; \mathbf{w}_t))}{\partial \mathbf{w}_t}. \tag{2.15}$$

2.2 An Introduction To Neural Networks

Since we have discussed the ideas and principles of a classifier function, we now want to actually build a classifier. Consider the following functional unit, which we call a neuron. A neuron takes two input vectors $\mathbf{x}$ and $\mathbf{w}$ of length N as well as an input scalar b and produces an output value y:

$$y(\mathbf{x}; \mathbf{w}, b) = \sigma(a) = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b), \tag{2.16}$$

where

$$a = \mathbf{w}^T \cdot \mathbf{x} + b = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + ... + w_N \cdot x_N + b \tag{2.17}$$

is called the activation, and σ is called an activation function [6]. If we now have a data set of one-dimensional inputs $(x^{(1)}, x^{(2)}, x^{(3)}, ..., x^{(N)})$, and outputs $(y^{(1)}, y^{(2)}, y^{(3)}, ..., y^{(N)})$, we could fit a line through our data using the concept of a neuron introduced above: We set $\sigma(a) = a$, and learn a parameter vector $\mathbf{w} = [b, w_1]$ by minimizing a suitable loss function. A possible pick for a loss function might be the mean squared error [6],

$$\mathcal{L}(X_{train}, Y_{train}) = \frac{1}{2} \sum_{n=1}^{N} (\sigma([b, w_1] \cdot [1, x^{(n)}]^T) - y^{(n)})^2. \tag{2.18}$$

As elaborated in Section 2.1.2, a possible way to do so is to perform gradient descent. A suitable update rule would be

$$w_1 = w_1 - \eta \cdot \frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial w_1}, \tag{2.19}$$

where $\frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial w_1}$ can be computed using the chain rule of derivation:

$$\frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial w_1} = \frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial \sigma(a^{(n)})} \cdot \frac{\partial \sigma(a^{(n)})}{\partial a^{(n)}} \cdot \frac{\partial a^{(n)}}{\partial w_1} \tag{2.20}$$

with

$$\frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial \sigma} = \sum_{n=1}^{N} \frac{\partial l(y, \sigma(a^{(n)}))}{\partial \sigma(a^{(n)})}, \tag{2.21}$$

$$\frac{\partial l(y, \sigma(a^{(n)}))}{\partial \sigma(a^{(n)})} = \sigma(a^{(n)}) - y^{(n)} \tag{2.22}$$

$$\frac{\partial \sigma}{\partial a^{(n)}} = 1, \tag{2.23}$$

$$\frac{\partial a^{(n)}}{w_1} = x^{(n)}, \tag{2.24}$$

$$\tag{2.25}$$

we get

$$w_1 = w_1 - \eta \cdot \sum_{n=1}^{N} (\sigma(a^{(n)}) - y^{(n)}) \cdot x^{(n)}. \tag{2.26}$$

Similarly, our weight update rule for the trainable bias computes as

$$b = b - \eta \cdot \sum_{n=1}^{N} (\sigma(a^{(n)}) - y^{(n)}). \tag{2.27}$$

If we want to do classification on the other hand, and therefore $y \in \{0, 1\}$, we can set σ to a function which will approximately map the input space to $\sigma \in \{0, 1\}$. A common used function for that purpose is the logistic function

$$\sigma(a) = \frac{1}{1 + \exp(-a)}, \tag{2.28}$$

with the derivative

$$\frac{\exp(a)}{(\exp(a) + 1)^2}. \tag{2.29}$$

We get as an update rule for w_1

$$w_1 = w_1 - \eta \cdot \sum_{n=1}^{N} (\sigma(a^{(n)}) - y^{(n)}) \cdot \frac{\exp(w_1 \cdot x^{(n)} + b)}{(\exp(w_1 \cdot x^{(n)} + b) + 1)^2} \cdot x^{(n)}. \tag{2.30}$$

Analog to our regression example, the update equation for the bias now is

$$b = b - \eta \cdot \sum_{n=1}^{N} (\sigma(a^{(n)}) - y^{(n)}) \cdot \frac{\exp(w_1 \cdot x^{(n)} + b)}{\exp(w_1 \cdot x^{(n)} + b) + 1)^2}. \tag{2.31}$$

When we attempt to do linear fitting, as in the first example, we talk of linear regression. If we use a logistic function, as with the neuron in the second example, we talk of logistic regression, which can be considered classification.

Now that we have introduced the concept of neuron, we use them as building blocks to build arbitrarily complex classifier functions. The usual way to do so is to arrange the neurons in layers as shown in as shown in the sketch in Figure 2.3, where every neuron of a given layer takes the outputs of every single neuron of the previous layer as its input. These layers of neurons stacked together, performing sequential nonlinear transformations of the input space, are referred to as a neural network [49]. It is common to stack at least 2 or 3 layers of neurons that way, but recently excellent results have been achieved by training networks from 6 up to 22 (for instance [38, 45, 64]), layers of neurons, which is referred to in the literature as deep learning [5].

In a nutshell, we can see that neural networks transform the input data by performing sequential nonlinear transformations on it by stacking several layers of neurons with a nonlinear activation function.

If we want to evaluate an output value of our neural network as a function, we can take an input vector $\mathbf{x}^{(n)}$ from our data set (X, Y) and set it as the input layer of our neural network, and therefore as input of all neurons in the next layer. The output value of the network is then computed layer by layer: For every layer, its output is multiplied with the weight vectors of the neurons of the next layer, to compute the input to these nodes. For instance, as already mentioned, the input layer holds the input vector $x^{(n)}$:

$$\mathbf{x}_{\text{input}}^{(n)} = \mathbf{x}^{(n)} \tag{2.32}$$

where the n superscript in $\mathbf{x}_{\text{input}}^{(n)}$ refers to the sample with the index n of the data set and the subscript refers to the layer of the network, in this case the input layer. The first dimension (or for deeper layers: neuron) of $\mathbf{x}_{\text{input}}^{(n)}$ would then be $x_{\text{input},1}^{(n)}$. For weights, a similar notation is used, $w_{k,m}^{(l)}$ is the neuron connecting neuron k of layer $l-1$ with neuron m of layer l.

Therefore, using our new notation, the activation for neuron m in layer l for input sample $x^{(n)}$ in the network can be computed as

$$a_{l,m}^{(n)} = w_{1,m}^{(l)} \cdot x_{(l-1),1}^{(n)} + w_{2,m}^{(l)} \cdot x_{(l-1),2}^{(n)} + ... + w_{D^l,m}^{(l)} \cdot x_{(l-1),F}^{(n)} + b_o^{(l)}. \tag{2.33}$$

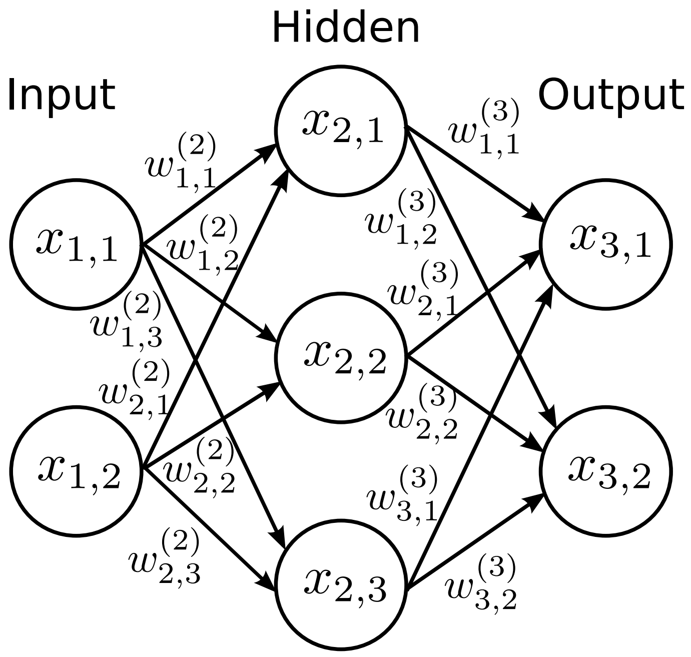


Figure 2.3: Sketch of neural network with input layer, output layer and one hidden layer

The output of layer m is then computed by applying the activation function of the neuron to the activation

$$x_{l,m}^{(n)} = \sigma(a_{l,m}^{(n)}). \tag{2.34}$$

Note that the equation above can also be written in matrix-vector notation

$$x_m^{(l+1)} = \sigma(\mathbf{w}_m^{(l)} \cdot \mathbf{x}_{l-1}^n + b_m^{(l)}), \tag{2.35}$$

where $\mathbf{w}_m^{(l)}$ is the weight vector of neuron m of layer l, b_m^l is the bias of the same neuron, and $\mathbf{x}_{l-1}^{(n)}$ is the output vector of layer $l-1$ as a response of input vector $\mathbf{x}^{(n)}$.

That way, values are propagated towards the output of the network layer for layer. This process is called forward propagation, and accordingly this type of neural network is called a feed forward network.

2.2.1 Training Of A Neural Network

To train neural networks, the stochastic gradient descent algorithm (see Section 2.1.2) is most commonly used [41]. While we already have demonstrated gradient descent for a single neuron, the way in which algorithms can be applied to the parameters in the hidden layers of the network is not obvious. For a given single neuron forming the output stage of a considered N layer neural network with the logistic activation function

$$\sigma(a) = \frac{1}{1+\exp(-a)}, \tag{2.36}$$

we can compute the gradient updates as shown before. This translates furthermore to the parameters of every neuron in the network: We compute the weight $w^{(l)}_{m,o}$, connecting neuron m layer $l-1$ with neuron o in layer l straight forward:

$$w^l_{m,o} = w^l_{m,o} - \eta \cdot \frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial w^l_{m,o}}, \tag{2.37}$$

where

$$\frac{\partial \mathcal{L}(X_{train}, Y_{train})}{\partial w^l_{m,o}} = \sum_{n=1}^{N} \frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial w^l_{m,o}}. \tag{2.38}$$

Thus, we need to compute

$$\frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial w^l_{m,o}} \tag{2.39}$$

for all parameters $w^l_{m,o}$ and every input sample $\mathbf{x}^{(n)}$ update the weights of our output layer.

We could now calculate all loss function gradients for all parameters given all input images individually, but this becomes very tedious in terms of computational as well as implementation effort. For very deep networks with 7 or more layers and 10^6 parameters, it is definitely infeasible. Thus the algorithm of backpropagation has been developed [10, 19, 36, 46, 57, 69], which efficiently makes use of the chain rule of derivation to perform as little computations as possible when "backpropagating" the gradients through the network. For illustration, consider an L-layer network. The gradient of a given weight $w^{(L)}_{m,o}$, from a neuronm of layer $L-1$ to neuron o of the output layer L, in our notation $\frac{\partial l(y^n, \hat{y}^n)}{\partial w^L_{m,o}}$ can be computed as

$$\frac{\partial l(y,\hat{y})}{\partial w^L_{m,o}} = \frac{\partial l(y,\hat{y})}{\partial \sigma(a_{L,o})} \cdot \frac{\partial \sigma(a_{L,o})}{\partial a_{L,o}} \cdot \frac{\partial a_{L,o}}{\partial w^{(L)}_{m,o}}, \tag{2.40}$$

where $\sigma(a_{L,o})$ can be rewritten as $x_{L,o}$:

$$\frac{\partial l(y,\hat{y})}{\partial w^L_{m,o}} = \frac{\partial l(y,\hat{y})}{\partial x_{L,o}} \cdot \frac{\partial x_{L,o}}{\partial a_{L,o}} \cdot \frac{\partial a_{L,o}}{\partial w^{(L)}_{m,o}}. \tag{2.41}$$

If we consider a similar weight $w^{L-1}_{m,o}$ in layer $(L-1)$ we get for the gradient

$$\frac{\partial l(y,\hat{y})}{\partial w^{L-1}_{m,o}} = \frac{\partial l(y,\hat{y})}{\partial x_{(L-1),o}} \cdot \frac{\partial x_{(L-1),o}}{\partial a_{(L-1),o}} \cdot \frac{\partial a_{(L-1),o}}{\partial w^{((L-1))}_{m,o}}, \tag{2.42}$$

due to the identical structure of the neurons in layers L and $L-1$. We can calculate $\frac{\partial l(y,\hat{y})}{\partial x_{(L-1),o}}$ as

$$\frac{\partial l(y,\hat{y})}{\partial x_{L-1,o}} = \sum_{f=1}^{F} \frac{l(y,\hat{y})}{x_{L,f}} \cdot \frac{\partial x_{L,f}}{\partial x_o^{(L-1)}}. \tag{2.43}$$

Thus, by backpropagating the gradients on the network from the last layer on, we can always compute parameter updates according to Eqn. (2.42) when we have previously computed the backpropagated error gradient $\frac{\partial l(y,\hat{y})}{\partial x_{l,o}}$ using Eqn. (2.43).

The algorithm can thus be comprehended to the listing shown in Algorithm 1.

Algorithm 1: Gradient Backpropagation

```
Data: input sample $x^{(n)}$, label $y^{(n)}$, NN with L layers
Result: Update gradients $\frac{\partial l(y,\hat{y})}{\partial w^l_{m,o}}$
Propagate input sample $x^{(n)}$ forward;
Retrieve class estimate $\hat{y}^{(n)}$;
Compute loss gradients for output neurons $\frac{\partial l(y,\hat{y})}{\partial x^{(n)}_{L,o}}$ ;
Set layer index l=L;
while layer l is not input layer do
    Compute update gradients for layer parameters $\frac{\partial l(y^{(n)},\hat{y}^{(n)})}{\partial w^l_{m,o}}$ using Eqn. (2.42);
    Backpropagate loss gradients down to the previous layer using Eqn. (2.43);
    Proceed to previous layer (l = l-1);
```

2.2.2 Different Activation functions

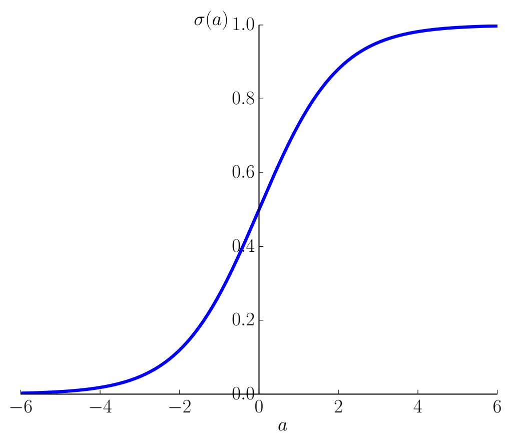
Through repeated nonlinear transformations of the input space, which are controlled through the choice of the networks activation function, very powerful models can be trained [27]. Until recently, one of the most common class of activation functions were sigmoids, from whom the best known are the logistic function

$$\sigma(a) = \frac{1}{1+\exp(-a)} \tag{2.44}$$

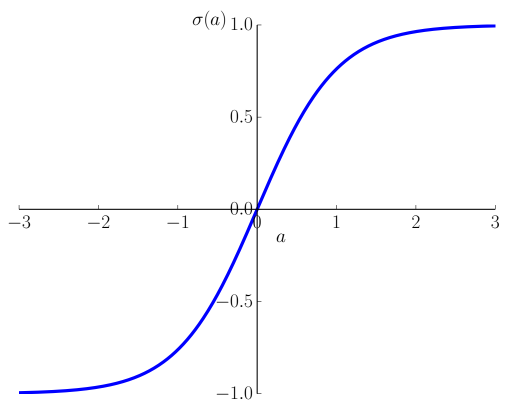
as well as the hyperbolic tangent

$$\sigma(a) = \tanh(a). \tag{2.45}$$

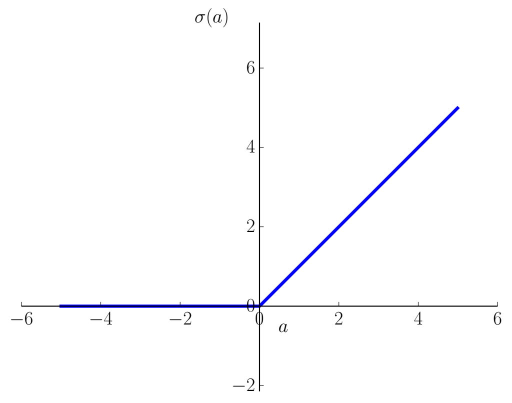
Considering the plots of both activation functions as shown in Figure 2.4, we see that both activation functions have a steep ascent around the origin, and get more flat as they thrive against positive and negative infinity. Obviously, the intention is to simulate an on-off behavior, where the network can steer the input to the neuron in a way such that the neuron is either active (1) or inactive (0/-1). Using logistic or hyperbolic tangent functions for this purpose has the advantage that these functions are continuous and thus differentiable over the whole input space of real numbers. They have a drawback though: for input values > 1 or < -1, these functions exhibit almost no gradients. Suppose now for a given neuron, the majority of the input values is either larger than 1 or smaller than -1: in that case the corresponding update gradients will be pretty small. Thus, the learning process will slow down towards a point where it almost comes to stop for that particular neuron. If this occurs for many neurons in a given network, the network will not train properly anymore. This gradients in the network have "vanished" due to saturation of the output values [30, 41].



(a) Logistic function.



(b) Hyperbole tangent.



(c) Rectified Linear Unit (ReLU).

Figure 2.4: Various common activation functions for neural networks: the logistic function, the hyperbolic tangent, as well as the most recent proposed rectified linear unit (ReLU).

Recently though, Nair and Hinton [52] have proposed a non-linearity with constant gradient in the direction of the positive axis which is defined as

$$\sigma(a) = \max(0, a). \tag{2.46}$$

Due to their constant gradient, the so-called rectified linear units (ReLUs) can be used to train much deeper network architectures than logistic or hyperbolic tangent functions, and are used in some variation in many networks which pose the state of the art in image recognition [1, 27, 37, 45, 64]. Nevertheless, Maas et al. [48] argue that ReLUs still show room for improvement considering their gradient properties, since the gradients for samples $x < 0$ are still 0, and therefore some neurons might be rendered inactive nevertheless. To tackle this problem, the leaky ReLU (LReLU) [48] has been proposed:

$$\sigma(a) = \begin{cases} a & \text{if } (a >= 0) \\ d \cdot a, & \text{else} \end{cases} \tag{2.47}$$

where d is an arbitrary constant which is $\ll 1$. That way, the activation function is still nonlinear as necessary to perform several nonlinear transformations of the input space, but a small constant gradient is preserved also for $(x < 0)$. Other approaches made the steepness parameter d of the LReLU trainable [27], or even learned convex piecewise functions by forming weighted sums of these negative ReLUs and learning their weights [1].

2.2.3 Overfitting In Neural Networks

We have already read about the problem of overfitting in Section 2.1: A classifier trains a model which predicts the training set labels well, but fails to generalize the problem at hand well enough to predict the labels of the test setwith satisfactory accuracy. This phenomenon is called overfitting [6]. The classifier derives for every single, or very small sets, of training vectors individual rules on how to obtain the output label, and thus learns the vector-label pairs of the training set"by heart", rather than deriving rules on how to generally compute the correct output labels from a given training vector.

One possible way to reduce overfitting is to penalize large weights in our network in order to avoid that they stretch overfit the data. Bishop [6] suggests to penalize the squared L2 norm of our current weight vector $\mathbf{w}^l$ of layer l to keep it small. We can do so by adding the following term to our loss function $\mathcal{L}(X_{train}, Y_{train})$

$$\begin{aligned} \hat{\mathcal{L}}(X_{train}, Y_{train}) &= \mathcal{L}(X_{train}, Y_{train}) + \frac{\lambda}{2}\sum(w_{m,o}^{(l)})^2 && (2.48) \\ &= \mathcal{L}(X_{train}, Y_{train}) + \frac{\lambda}{2}\mathbf{w}_o^{(l)T}\mathbf{w}_o^{(l)}, && (2.49) \end{aligned}$$

where $\mathcal{L}(X_{train}, Y_{train})$ is the loss function we seek to minimize and $w_{m,o}^{(l)}$ is the weight connecting neuron m on layer $l-1$ to neuron o of layer l. This process is called regularization. Note that if we now compute our derivatives for the layers of our neural network, for every layer except layer l, the right term $\mathbf{w}_o^{(l)T}\mathbf{w}_o^{(l)}$ of our loss function $\hat{\mathcal{L}}(X_{train}, Y_{train})$ is constant. Therefore, its gradient is zero when derived for a weight $w_i^{(q)}$, where $q \neq l$, and the optimization process of layer q is completely unaffected by the additional term. In layer l, on the other hand, we derive

$$\frac{\partial\hat{\mathcal{L}}(X_{train}, Y_{train})}{\partial w_{m,o}^l} = \frac{\partial\mathcal{L}(X_{train}, Y_{train})}{\partial w_{m,o}^l} + \lambda \cdot w_{m,o}. \tag{2.50}$$

Thus our overall gradient update equation is

$$w_{m,o}^l = w_{m,o}^l - \frac{\partial\mathcal{L}(X_{train}, Y_{train})}{\partial w_{m,o}^l} - \lambda \cdot w_{m,o}^l. \tag{2.51}$$

Eqn. (2.51) also delivers a nice interpretation on how the regularization of the loss affects the weight update. We subtract a fraction of our current weight in the update equation to avoid our weights to become too big. As a consequence, the network is penalized when minimizing the loss solely through the weights magnitude instead of learning meaningful combinations, and therefore avoid a degenerate solution of the problem.

Thus, the parameter λ is called the weight decay, because it encourages the classifier to minimize the weights to 0 in case they do not fit the data [6]. Since all other layers of our neural network are unaffected by the adding of such a term, we can add such a term for every layer of the network where we have parameters to optimize. In implementation terms this usually boils down to modifying the update equation for every layer to Eqn. (2.51).

Similarly we can also apply L1 regularization of our loss function:

$$\hat{\mathcal{L}}(X_{train}, Y_{train}) = \mathcal{L}(X_{train}, Y_{train}) + \lambda \cdot \sum \sqrt{(w_i^{(l)})^2}. \tag{2.52}$$

The according gradient update rules to implement in the layers where weights have to be learned is then:

$$\frac{\partial\hat{\mathcal{L}}(X_{train}, Y_{train})}{\partial w_i^l} = \frac{\partial\mathcal{L}(X_{train}, Y_{train})}{\partial w_i^l} + \lambda \cdot \operatorname{sgn}(w_i). \tag{2.53}$$

An additional strategy to avoid or at least diminish overfitting has been introduced by Hinton et al.[29] among others [63]: For every training iteration of the neural network, Hinton et al. randomly exclude a certain percentage of neurons in every layer from the training process. For evaluation, however, they always train the whole network with reduced weights. Hinton et al. argue, that this way, the network, left with less parameters to train in the current iteration, is forced to train a well generalizing model, while evaluating the whole network corresponds to averaging process of all these individually trained models. Formally, the procedure can be expressed as a modification of the networks activations [14], such that for a layer l with dropout applied, and a input vector

$\mathbf{x}_{(l-1)}$

$$\mathbf{x}_l = \sigma\left(\left(\frac{1}{1-r} \cdot W^{(l)} \cdot \mathbf{m} \odot \mathbf{x}_{(l-1)}\right) + b^{(l)}\right). \tag{2.54}$$

The constant $r \in [0,1]$ is the probability of a neuron to drop out and $\mathbf{m}$ is a binary mask randomly drawn from $Bernoulli(1-r)$, which indicates which neurons of layer $(l-1)$ shall be dropped in the current forward step. $\odot$ denotes Hadamard product of the vectors $\mathbf{m}$ and $\mathbf{x}_{(l-1)}$ and $W^{(l)}$ is the set of weight vectors $w_1^{(l)}...w_F^{(l)}$ of layer l in matrix notation such that

$$\begin{pmatrix} \mathbf{w}_1^{(l)T} \\ \mathbf{w}_2^{(l)T} \\ ... \\ \mathbf{w}_F^{(l)T} \end{pmatrix}. \tag{2.55}$$

At test time, values are propagated through the network with regular activations, which poses the implicit model averaging process mentioned before, since it can be interpreted evaluating an ensemble of networks with partly shared weights and averaging their output (by omitting the factor $\frac{1}{1-r}$ in the networks activations).

In general Dropout, has proven to be more effective for fully connected layers than for convolutional layers, whom we will discuss in Section 2.3, since convolutional layers tend to have less parameters through their implicit weight sharing mechanism. Dropout has shown to deliver excellent results in terms of overfitting reduction, and is currently (2015) part of almost every deep neural network achieving state of the art results [1, 25, 62].

2.3 Neural Networks For Recognition

Usually images are transformed to feature vectors in a preprocessing step to make them invariant towards brightness, perspective, scale, rotation etc.. While this approach has worked well in the past, these feature transformations are very application dependent, and thus, a multitude of feature descriptors has been proposed (see [3, 4, 11, 12, 15, 44, 47, 60]). Thus, performing end-to-end image recognition with a single method that generalizes well seems useful. LeCun et al. [40] found a way to do so by introducing CNNs: They jointly learn low-dimensional invariant feature transforms for input images by applying three concepts. First, they implement locally receptive fields inside a neural network layer to train the network to recognize certain various simple patterns such as lines, blobs, and corners in all possible orientations. Second, they make locally receptive fields share weights, such that certain patterns can be recognized independently of its location in the image. Thereafter, they apply spatial subsampling to the responses these layers to reprocess them with another set of locally receptive fields and thus incrementally recognize more complex structures. Finally, regular fully connected layers are used to learn to infer actual classes for the set of complex structures that have been recognized. Note that this concept is very similar to the concept of simple and complex cells introduced by Hubel and Wiesel [31] as they researched the visual recognition system of a cat. Thus, one might interpret convolutional neural networks as a rough computational model of the mammalian visual system.

To implement locally receptive fields a layer of a neural network, we separate the input image into small, overlapping patches. Instead of receiving input from all input pixels of the image, neurons receive only input from a single image patch. That way, if we have a patch of the image of size $U \times V$ the weight vector $\mathbf{w}$ of a given input vector has $U \times V$ elements. The activation $a_{l,o}^{(n)}$ of the given layer l is then computed as

$$a_{l,q}^{(n)} = b^{(l,q)} + w_{1,1}^{(l)} \cdot x_{(l-1),q,1,1}^{(n)} + w_{1,2}^{(l)} \cdot x_{(l-1),q,1,2}^{(n)} + \cdots + w_{U,V}^{(l)} \cdot x_{(l-1),q,U,V}^{(n)} \tag{2.56}$$

$$= b^{(l,q)} + \sum_{u}^{U} \sum_{v}^{V} w_{u,v}^{(l)} \cdot x_{(l-1),q,u,v}^{(n)} \tag{2.57}$$

where the indices u and v for $w_{u,v}^{(l,q)}$ and $x_{l,q,u,v}^{(n)}$ in this case count the pixel inside the patch q, and n as usually counts the input sample.

While this operation is rather straight forward for gray-scale images $x^{(n)}$, we also need to consider images with multiple color channels. We extend the layers image patches over all channels by adding a third dimension C, the number of input channels to the kernel $\mathbf{w}$ and thus get a kernel of size $U \times V \times C$. Similarly, we introduce $\mathbf{x}^{(n,c)}$ as the channel c of input image $\mathbf{x}^{(n)}$. By stretching our inner product operation over the all channels of a given input patch, we get

$$a_{l,q}^{(n)} = b^{(l,q)} + \sum_{u}^{U} \sum_{v}^{V} \sum_{c}^{C} w_{u,v,c}^{(l)} \cdot x_{(l-1),q,u,v}^{(n,c)}. \tag{2.58}$$

After the activation $a_{l,q}^{(n)}$ has been computed for the image patch $\mathbf{x}_{(l-1),q}^{(n)}$ it is, as usual, transformed using a nonlinear activation function:

$$x_{l,q}^{(n)} = \sigma(a_{l,q}^{(n)}) \tag{2.59}$$

Note that the equation 2.56 resembles the operation of convolution [65] with additional bias. Consequently this kind of layer is called a convolutional layer [40, 42]. The multiplication and summing of the image patch with the weights can be interpreted as a kind of filtering operation, similar to linear filters, which are widely used in computer vision. We can interpret this as training filter coefficients to detect patterns that are relevant for recognition in our images.

Now, if we learn individual parameters for every single image patch we extracted, it might occur that we are able to recognize certain patterns only on certain patches of the image. Since objects and therefore patterns might move inside the image, this might result in weak classifier performance. For this reason we train every patch extracted from an image with the same weight parameters, by sharing them over the patches of the whole image plane and in consequence train the network to detect the same pattern in every image patch. Such a set of neurons with shared weights, who all detect the same pattern, are called a feature map. Since we need to detect more than a single pattern, we train many feature maps for a convolutional layer.

To share the weights of all neurons belonging to a single feature map, LeCun et al. add the computed gradients of the weights for all Q patches extracted from the image [40]. The overall update gradient of weight $w_{s,t}^{(l,q)}$ for image patch q in our layer l is consequently computed as

$$w_{u,v}^{(l,q)} = w_{u,v}^{(l,q)} - \eta \cdot \sum_{q=1}^{Q} \frac{\partial l(y, \hat{y}^{(n)})}{\partial w_{u,v}^{(l,q)}}, \tag{2.60}$$

where Q is the number of patches in our feature map.

Finally, to recognize relevant patterns on different scales, [40] perform subsampling after every convolutional layer, using a so-called pooling layer and thus reduce the dimensionality of the filter responses before processing them with further convolutional layers. This subsampling is arranged as well in feature maps where there is a feature map for every feature map in the previous convolutional layer. Contrary to the convolutional layer, the feature maps of the subsampling layer do not train any weights, but simply perform a pooling operation on the input data. Said pooling operation is usually selecting the maximum value or computing the average value from all input values of the patch.

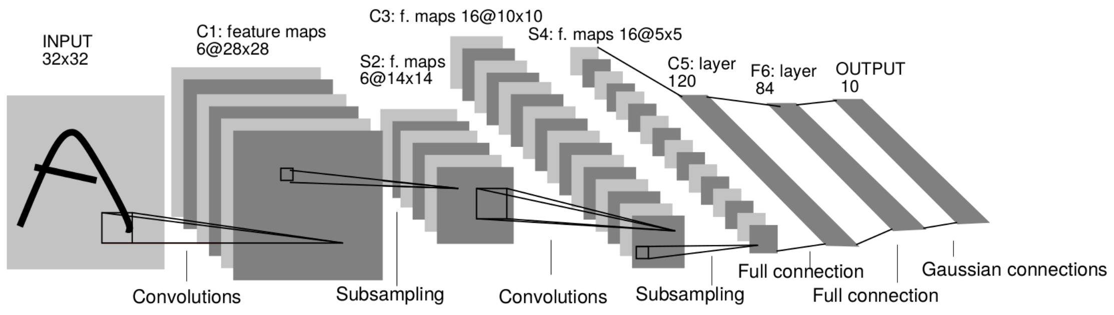


Figure 2.5: A sketch of LeNet-5 illustrating the architecture of a convolutional neural network. Figure taken from [40].

Figure 2.5 shows a sketch from [40] illustrating the architecture of a convolutional neural network. As we can see, the amount of feature maps increases for layers closer to the output. The intention is to train many complex representations in higher layers that are composed of simpler, more general patterns in the lower convolutional layers. state-of-the-art convolutional neural networks are still designed following this principle (e.g. [1, 38]).

2.4 Related Work

Several attempts of training at least parts of activation functions have been performed recently. Lin et al. proposed an approach called Network in Network [45], which replaces the linear model in feature maps as introduced in Section 2.3, Eqn. (2.56), with a small neural network within the network for each feature map of a convolutional layer.

Another approach, which has been developed specifically for use with Dropout is called Maxout [25]. For Maxout, not a single set of weights is trained for a network layer, but several sets of weights for every single neuron of the layer. The neuron then chooses the set of input weights for a given input vector $\mathbf{x}$ that yields the maximal response of the neuron:

$$\begin{aligned} \sigma(\mathbf{x}) &= \max(a^{(l)}_{1,o}, a^{(l)}_{2,o}, ..., a^{(l)}_{G,o}) && (2.61) \\ &= \max(\mathbf{w}^{(l)}_{1,o} \cdot \mathbf{x}, \mathbf{w}^{(l)}_{2,o} \cdot \mathbf{x}, \cdots, \mathbf{w}^{(l)}_{G,o} \cdot \mathbf{x}). && (2.62) \end{aligned}$$

Note that, for simplicity, we make a single exception of our usual notation here: $\mathbf{w}_{1,o}$ is *not* the single weight from neuron 1 of the previous layer to neuron o of layer l, but rather, *the first of G trained weight vectors*, that connect every neuron of the layer $(l-1)$ to the neuron o of the layer l. $a_{1,o}$ is thus the activation of neuron o obtained by multiplication of the input vector $\mathbf{x}$ with the weight vector $\mathbf{w}_{1,o}$.

Goodfellow et al. argue, that introducing this kind of layer corresponds to training a piecewise-linear convex activation function [25].

The remaining recent attempts to train activation functions, constrain themselves to piece-wise linear functions as well: He et al. [27] propose an approach where they train the steepness constant d of a leaky ReLU and have achieved excellent results, approaching human level performance, on the ImageNet Large Scale Visual Recognition Challenge data set [58]. Like for the LReLU, their activation function can be defined as

$$\sigma(a) = \begin{cases} a & \text{if } (a >= 0) \\ d \cdot a, & \text{else} \end{cases} \tag{2.63}$$

except for the fact that He et al. [27] train d as an adaptive parameter as part of the networks training process, contrary to the LReLU where d is a constant. Eqn. (2.63) can also be rewritten as

$$\sigma(a) = \max(0, a) - d \cdot \max(0, -a). \tag{2.64}$$

In a similar way, Agostinelli et al. [1] have attempted to train not only one, but several negative components of the leaky ReLU, where they also optimize their starting positions. They add a number of terms $\max(0, -x + c^{(l)}_n)$, to a ReLU bias and optimize their weights and starting positions in the form

$$\sigma(a) = \max(0, a) + \sum_{i=1}^{I} k_i \cdot \max(0, -a + c_i), \tag{2.65}$$

where k_i is the weight with index i in a given neuron, I is the number of weights in the same neuron, and c_i indicates the position of the negative ReLU term.

Finally, our work has been inspired by the work of Chen et al. [13], who train filter parameters and influence functions for image restoration processes by fitting Gaussians Radial Basis Function model into their diffusion models. In this thesis, we introduce a general framework to train activation functions, which uses a similar model as the one applied by Chen et al. in their diffusion models. While Chen et al. constrain their models to use solely Radial Basis Functions though, we are able to apply every function with computeable gradients as a base function. We start out with a general activation function defined as

$$\sigma(a; \mathbf{k}, \mathbf{c}, u) = \sum_{i=1}^{I} k_i \cdot \phi(a; c_i, u), \tag{2.66}$$

where a is the input to the unit, $\mathbf{k}$ is a vector of amplitudes, $\mathbf{c}$ is a vector of positions on the a-axis and u is a shape parameter, which we use to control the shape of our arbitrary base function $\phi(a, c_i, u)$. Note that when setting

$$\phi(a; c_i, u) = \max(0, -a + c_i) \tag{2.67}$$

we get

$$\sigma(a) = \sum_{i=1}^{I} k_i \cdot \max(0, -a + c_i), \tag{2.68}$$

which is in fact Eqn. (2.65) minus the ReLU bias. If we further set $I = 1$, and $\mathbf{c} = \mathbf{0}$ we get

$$\sigma(a) = k_1 \cdot \max(0, -a), \tag{2.69}$$

which is identical to Eqn. (2.64), again without its ReLU bias. Thus the trainable components in the units of Agostinelli et al. [1] as well as He et al. [27] can be considered as special cases of Eqn. (2.66).

Nevertheless, since the pick of the base functions is arbitrary for Eqn. (2.66), contrary to Maxout [25], APL units [1], and PReLUs [27], we are able train non-convex, continuous activation functions. Consequently, our activation functions are expected to be more expressive in general, since they are less constrained than the ones of the methods mentioned above. Furthermore, contrary to the methods mentioned above, we share the parameters of our base functions in order to keep the number of parameters in our layers low. Moreover, to ensure a stable training process, we to constrain the positioning as well as the shape factors of our base function to meaningful combinations.

3

Learning Activation Functions for Recognition Tasks

Deep learning has received a recent boost due to increased memory and processing power as well as novel activation functions and overfitting reduction methods [71]. In the process, several proposed methods which incorporate learning parts of the activation function have shown state-of-the-art performance [1, 25, 27]. Yet, these methods learn solely steepness parameters for ReLUs [1, 27], or the activation functions [25] are strictly convex.

Therefore, we propose a method to train non-convex activation functions, which we hope will be more expressive, by learning the weight, position and shape of a set base functions:

$$\sigma(a; \mathbf{k}, \mathbf{c}, u) = \sum_{i=1}^{I} k_i \cdot \phi(a; c_i, u), \tag{3.1}$$

where a is the input to the unit, $\mathbf{k}$ is a vector of amplitudes, $\mathbf{c}$ is a vector of positions on the a-axis and u is a shape parameter, which we use to control the shape of our exchangeable base function $\phi(a, c_i, u)$. $\mathbf{k}$, $\mathbf{c}$, and u are identical for a given layer and are updated using the regular backpropagation algorithm with gradient descent as introduced in Section 2.2.1. Thus we will attempt to learn an individual activation function for every single layer of our neural network.

Note that while this model is definitely inspired by Radial Basis Functions, which have been used previously in neural networks [9, 40], to our knowledge, these approaches were not directed to learn an activation function the way we propose. Furthermore, we do not constrain our base functions to be Radial Basis Functions and consequently, they don't rely solely on the distance of the input value a to the location parameter c_i as an input, but also may be defined in different ways, depending on how it might fit given the problem at hand.

In this Chapter, we show how the model from Eqn. (3.1) can be implemented in a neuron layer to create neural networks with trainable, non-convex activation functions in its most general form: In Section 3.1, we derive the general gradient update rules needed to train a parameterized activation function as the one proposed above. Thereafter, since the proposed model poses challenges in terms of the initialization of position and shape parameters, as well as computational feasibility, we discuss the measures we take to keep the amounts of parameters low, while learning meaningful models in Section 3.2. Section 3.3 treats enforced regularization as a measure to tackle overfitting in our units, after which we give a short overview on the various base functions we used for the learning process in our networks in Section 3.4. Finally, we shortly discuss the amount of parameters trained in our unit and compare it to the amount of parameters of current state-of-the-art approaches [1, 27] in Section 3.5.

3.1 A General Derivative Framework For Function Approximation Units

To learn the parameters of our activation function shown in Eqn. (3.1), we need to incorporate it into the backpropagation process of the network. As we discussed at length in Section 2.2.1, our preferred way of training neural networks is via (stochastic) gradient descent using gradient backpropagation. Thus, for a given neuron implementing the activation function from Eqn. (3.1), the gradient descent update rule (neglecting momentum and weight decay terms) for amplitude k_i of basefunction i would be

$$k_i = k_i - \eta \cdot \sum_{n=1}^{N} \frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial k_i}, \tag{3.2}$$

where $l(y^{(n)}, \hat{y}^{(n)})$ is a loss measure, $y^{(n)}$ is the label of the input vector with index n, $\mathbf{x}^{(n)}$, and $\hat{y}^{(n)}$ is the networks estimate for the class label, given $\mathbf{x}^{(n)}$. Note that we omitted the layer and neuron indices of k_i in this case to keep the notation simple. According to the backpropagation algorithm (see Section 2.2.1 for details), we need to compute

$$\frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial k_i} = \frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)} \cdot \frac{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}{\partial k_i}. \tag{3.3}$$

Luckily, we get $\frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}$ delivered as a result of a previous step of the backpropagation algorithm and only need to compute

$$\frac{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}{\partial k_i} = \phi(a; c_i, u). \tag{3.4}$$

Similarly, we can compute the gradient updates for $\mathbf{c}$ and u as

$$\frac{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}{\partial c_i} = k_i \cdot \frac{\partial \phi(a; c_i, u)}{\partial c_i}, \tag{3.5}$$

$$\frac{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}{\partial u} = \sum_{i=1}^{I} k_i \cdot \frac{\partial \phi(a; c_i, u)}{\partial u}. \tag{3.6}$$

Furthermore, to backpropagate the gradients down to the next layer, we also need to compute

$$\frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial a} = \frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)} \cdot \frac{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}{\partial a}. \tag{3.7}$$

We compute

$$\frac{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}{\partial a} = \sum_{i=1}^{I} k_i \cdot \frac{\partial \phi(a; c_i, u)}{\partial a}. \tag{3.8}$$

and multiply it with the already known $\frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial \sigma(a; \mathbf{k}, \mathbf{c}, u)}$.

This way, we can take any function $\phi(a, c_i, u)$ for which we are able to compute its (sub)gradients for all parameters, and use it as a base function in our unit. For instance we could set $\phi(a; c_i, u) = e^{-\frac{(a-c_i)^2}{2u_i^2}}$ and accordingly

$$\frac{\partial \phi(a; c_i, u)}{\partial a} = e^{-\frac{(a-c_i)^2}{2u_i^2}} \cdot -\frac{(a - c_i)}{u_i^2} \tag{3.9}$$

$$\frac{\partial \phi(a; c_i, u)}{\partial c_i} = e^{-\frac{(a-c_i)^2}{2u_i^2}} \cdot \frac{(a - c_i)}{u_i^2}, \tag{3.10}$$

$$\frac{\partial \phi(a; c_i, u)}{\partial u} = \sum_{i=1}^{I} e^{-\frac{(a-c_i)^2}{2u_i^2}} \cdot \frac{(a - c_i)^2}{u_i^3}. \tag{3.11}$$

Until now, we have discussed the updates for a single neuron, thus implying that a set of parameters $\mathbf{k}$, $\mathbf{c}$ and u is separate for every neuron, which would imply that we train an activation function for every single neuron. Since we want to keep the amount of parameters of our model low though, we want to train only a single set of parameters $\mathbf{k}$, s and u for a given neuron layer.

For convolutional neural networks the parameters of the filter weights for a given feature maps are combined through addition of the gradients [40]. This method has proven to work well for CNNs (for instance [38]), thus we follow the same direction. We denote the update gradient of parameter k_i for neuron f in our layer as $\left(\sum_{n=1}^{N} \frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial k_i}\right)_f$, and get

$$k_i = k_i - \eta \cdot \sum_{f=1}^{F} \left(\sum_{n=1}^{N} \frac{\partial l(y^{(n)}, \hat{y}^{(n)})}{\partial k_i} \right)_f, \tag{3.12}$$

where F is the number of neurons in our layer. The parameter update equations for c_i and u are updated in the same way.

While this model allows us to learn activation functions using all kinds of different base functions, it still suffers from the drawback of a rather high amount of parameters. While the amount of parameters is arguable neglectable compared to the amount of parameters used in fully connected layers, such a parameterized network would be pretty hard to compare to a standard network with ReLU activation functions, and is furthermore expected to be take a lot longer to train. Furthermore, the initialization is still an open question, though critical for the optimization process, especially for position and shape parameters, since small changes might have large impacts for these parameters. Therefore we propose some constraints on our activation functions to reduce the amount of free parameters in our activation function on one hand, and attempt to enforce meaningful and expressive values for sigma and shape parameters on the other hand.

3.2 Reduction Of Parameters

The model proposed in Section 3.1 bears the drawback of many parameters which are optimized jointly and is therefore expected to exhibit higher training times and memory consumption than a regular ReLU network. Furthermore, the question remains how to set the positions for our base functions, since we usually do not know the exact distribution of the input space and can only make guesses.

To reduce the amount of free parameters while maintaining the overall expressive power of the model, we constrain the centers $\mathbf{c}$ to be spaced equidistant over the input domain. For this purpose, we introduce an initialization which assigns every element c_i of the vector $\mathbf{c}$ a value in the in interval $[-1, 1]$ in a way that the distance between two neighboring centers, c_i and c_{i+1} is always the same. Since we have I base functions, we want to set I centers equidistant on the interval [-1,1], while positioning c_1 at -1 and c_I at 1. Therefore, we divide the interval $[-1, 1]$ into $I - 1$ parts. It follows that the distance between c_i and c_{i+1} is

$$d = \frac{1 - (-1)}{I - 1} = \frac{2}{I - 1}. \tag{3.13}$$

That way, we can calculate the position of every center c_i by adding $(i - 1)$ distances d to the start point c_1 which lies at -1:

$$c_i = c_1 + (i - 1) * d = -1 + (i - 1) * d = -1 + (i - 1) * \frac{2}{I - 1}, \tag{3.14}$$

where I is the number of used base function and $i \in 1, .., I$.

Now, to position our base functions equidistant not only on the interval $[-1, 1]$, but on the whole definition space, we we introduce the input range factor s. By multiplying $\mathbf{c}$ with s, we make our base functions spread equidistantly on the interval $[-s, s]$. To optimize the position of our base functions, we now optimize the input range factor s to control on which interval our base functions spread.

To introduce this constraints into our model, we take the following measures: Since we want to have $\mathbf{c}$ fixed and optimize s instead, we treat $\mathbf{c}$ as a constant vector, defined as shown above, and use s as an parameter in our activation function instead:

$$\sigma(a; \mathbf{k}, s, u) = \sum_{i=1}^{N} k_i \cdot \phi(a; s \cdot c_i, u), \tag{3.15}$$

as well as the according gradient update rule

$$\frac{\partial \sigma(a; \mathbf{k}, s, u)}{\partial s} = \sum_{i=1}^{N} k_i \cdot \frac{\partial \phi(a; s \cdot c_i, u)}{\partial (s \cdot c_i)} \cdot \frac{\partial (s \cdot c_i)}{\partial s} \tag{3.16}$$

$$= \sum_{i=1}^{N} k_i \cdot \frac{\partial \phi(a; s \cdot c_i, u)}{\partial (s \cdot c_i)} \cdot c_i. \tag{3.17}$$

An additional problem, similar to the correct initialization of base function positions, poses also the initialization of base function shape, where applicable: For base functions which make use of the shape parameter u, an approximation of base functions will likely be more useful when the corresponding base functions are overlapping to approximate a target function. Models which attempt to approximate with too small u will create functions that form a saw tooth patterns rather than any continuous function. Therefore they might deliver a satisfactory approximation for single data points, but fail to deliver a reasonable approximation for large portions of the input space.

A possibility to enforce a sufficiently large sigma parameter would be to add a bias to ensure that all base functions are always well-overlapping.

We designed the following initialization scheme for Gaussian base functions $\phi(a; c_i, u) = e^{-\frac{(a-c_i)^2}{2u_i^2}}$, nevertheless it might be useful for other base functions with a similar shape parameter as well.

If we space the centers of our base functions such that their distance is $2u$, we can expect around 22% overlap for Gaussian base functions due to the 68-96-99-rule, which states that 68% of all values for a Gaussian distribution lie within $[-\sigma, \sigma] = [-u, u]$. Considering that the distance between the neighboring centers is $2u$, we set I base functions in a predefined interval of our input space. The centers of these base functions then divide the interval into (I-1) sub intervals of length $2u$. Therefore we get the relation

$$2u = \frac{l}{I-1}, \tag{3.18}$$

where I is the number of Gaussians and l is the length of our interval. Consequently,

$$u(l, I) = \frac{l}{2(I-1)}. \tag{3.19}$$

Considering that we already set a input range parameter for the model, which always describes the maximum value in an interval symmetric around zero, it is reasonable to replace l with $2s$ which leads to

$$u(s, I) = \frac{s}{(I-1)}. \tag{3.20}$$

For Gaussian base functions, this parameterized u can be used as a bias $u_{bias}(s, N)$ to ensure a basic reasonable spacing of Gaussians while adding an additional optimizeable component u_{opt} for fine-tuning:

$$u = u_{bias}(s, I) + u_{opt}. \tag{3.21}$$

If we now replace u with $u' = u_{bias}(s, I) + u_{opt} = \frac{s}{(I-1)} + u$ we obtain

$$\sigma(a; \mathbf{k}, s, u) = \sum_{i=1}^{I} k_i \cdot \phi\left(a; s \cdot c_i, \frac{s}{(I-1)} + u\right). \tag{3.22}$$

for our updated activation function. Since u is a function of s, this implies that we need to adjust the gradient update rules for u:

$$\begin{aligned}\frac{\partial\sigma(a; \mathbf{k}, s, u)}{\partial u} &= \sum_{i=1}^{N} k_i \cdot \frac{\partial\phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial(\frac{s}{(N-1)} + u)} \cdot \frac{\partial(\frac{s}{(N-1)} + u)}{\partial u} \qquad (3.23)\\ &= \sum_{i=1}^{N} k_i \cdot \frac{\partial\phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial(\frac{s}{(N-1)} + u)}, \qquad (3.24)\end{aligned}$$

as well as for s:

$$\begin{aligned}\frac{\partial\sigma(a; \mathbf{k}, s, u)}{\partial s} &= \sum_{i=1}^{N} k_i \cdot \frac{\partial\phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial(s \cdot c_i)} \cdot \frac{\partial(s \cdot c_i)}{\partial s} \qquad (3.25)\\ &= \sum_{i=1}^{N} k_i \cdot \frac{\partial\phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial(s \cdot c_i)} \cdot c_i \qquad (3.26)\end{aligned}$$

So we finally get the following set of gradient update rules

$$\frac{\partial \sigma(a; \mathbf{k}, s, u)}{\partial a} = \sum_{i=1}^{N} k_i \cdot \frac{\partial \phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial a} \tag{3.27}$$

$$\frac{\partial \sigma(a; \mathbf{k}, s, u)}{\partial k_i} = \phi(a; s \cdot c_i, \frac{s}{(N-1)} + u), \tag{3.28}$$

$$\frac{\partial \sigma(a; \mathbf{k}, s, u)}{\partial s} = \sum_{i=1}^{N} k_i \cdot \frac{\partial \phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial (s \cdot c_i)} \cdot c_i, \tag{3.29}$$

$$\frac{\partial \sigma(a; \mathbf{k}, s, u)}{\partial u} = \sum_{i=1}^{N} k_i \cdot \frac{\partial \phi(a; s \cdot c_i, \frac{s}{(N-1)} + u)}{\partial (\frac{s}{(N-1)} + u)}, \tag{3.30}$$

$$\tag{3.31}$$

where we can plug in any base function $\phi(a; c; u)$ for which we can compute the (sub)gradients after $\mathbf{x}$,$\mathbf{c}$ and u.

We refer to the units implementing this model as Approximating Learnable Units (ALUs), since they learn the optimal activation function through approximation.

3.3 Enforced Regularization

To stabilize the optimization process and reduce overfitting, for some of our models, we enforce the L1 or L2 norm of our weights k_i to be constant, such that

$$\sum_{i=1}^{I} |k_i| = 1, \tag{3.32}$$

for a constant L1 norm, and

$$\sqrt{\sum_{i=1}^{I} k_i^2} = 1, \tag{3.33}$$

for a constant L2 norm respectively. By renormalizing the amplitudes $\mathbf{k}$ after each gradient update, we seek to force our layers to learn meaningful combinations of weights instead of minimizing the loss by changing the overall magnitude of the activation function which is undesirable.

3.4 Proposed Approximation Base Functions

We expect the choice of base function to have high influence on the approximation performance. Consequently, we intend to preselect base functions for evaluation in our experiments, which promise the best results. In our preselection of base functions we intend to to incorporate the classical choices for approximation such as the Gaussian function which has been used successfully in Radial Basis Functions [9] as well as pursue novel approaches by attempting to approximate a function on the whole input domain by optimizing a sum of cosine functions. A short description of the base functions combined to approximate activation functions within neural networks follows.

3.4.1 Cosine

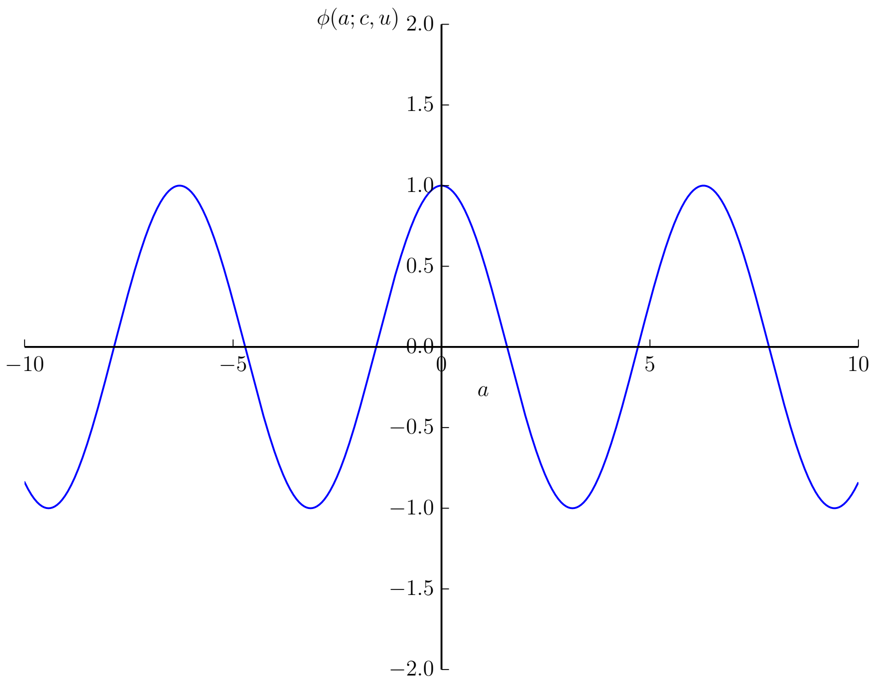


Figure 3.1: Cosine base function as it has been used in our network layers.

Cosine functions promise to be suitable base functions (see Figure 3.2 for illustration), since they are defined over the whole input space. That way, the question of choosing the correct input range based on the distribution of the input samples does not rise. Furthermore, cosine base functions exhibit varying gradient over the whole input domain. Therefore, we deem vanishing gradient problems when training with cosine base functions as unlikely. We define our cosine base function as

$$\phi(a; c, u) = \cos(c \cdot a). \tag{3.34}$$

The partial derivatives for a, c_i and u of the functions are compute then as

$$\frac{\partial \phi(a;c,u)}{\partial a} = -\sin(c \cdot a) \cdot c, \tag{3.35}$$

$$\frac{\partial \phi(a;c,u)}{\partial u} = 0. \tag{3.36}$$

and

$$\frac{\partial \phi(a;c,u)}{\partial c_i} = -\sin(c \cdot a) \cdot a. \tag{3.37}$$

Since it is not possible to define locally confined cosines, the cosine base functions pose an exception here concerning the interpretation of their parameters: We use $\mathbf{c}$ here to describe the angular frequencies of our set of base functions. This has proven more useful than taking u, which is the shape parameter of our base function and thus would be the natural choice. Unfortunately though, we have defined u to be fixed for all base functions to reduce parameter space, which lead to poor performance in preliminary experiments for cosine base functions. This was due to all cosines oscillating with the same base frequency u, which prevented the training of expressive models. Therefore, we choose this variant, since our initialization scheme of $\mathbf{c}' = s \cdot \mathbf{c}$ comes in handy as well here, and has a nice interpretation: c'_i has the interpretation of the angular frequency of the corresponding cosine, thus $\mathbf{c}'$ is a set of angular frequencies symmetrically spread over the spectrum. Consequently, s is then the maximal angular frequency used in the ensemble of base functions defining the breadth of the spectrum.

3.4.2 Gaussian

We use Gaussians base functions (see Figure 3.2 for illustration) because they are the classical choice when approximating functions with a Radial Basis Function network. Furthermore, Chen et al. [13], who originally inspired this work, obtained excellent results using parameterized sums of Gaussians to train influence functions for reaction diffusion models applied to image restoration. Our Gaussian base function is defined as

$$\phi(a;c,u) = e^{-\frac{(a-c)^2}{2u^2}}. \tag{3.38}$$

Consequently, the derivatives of $\phi(a;c,u)$ for a, c and u compute as

$$\frac{\partial \phi(a;c,u)}{\partial a} = e^{-\frac{(a-c)^2}{2u^2}} \cdot -\frac{(a-c)}{u^2}, \tag{3.39}$$

$$\frac{\partial \phi(a;c,u)}{\partial c_i} = e^{-\frac{(a-c)^2}{2u^2}} \cdot \frac{(a-c)}{u^2}, \tag{3.40}$$

and

$$\frac{\partial \phi(a;c,u)}{\partial u} = \sum_{i=1}^{N} e^{-\frac{(a-c)^2}{2u^2}} \cdot \frac{(a-c)^2}{u^3}. \tag{3.41}$$

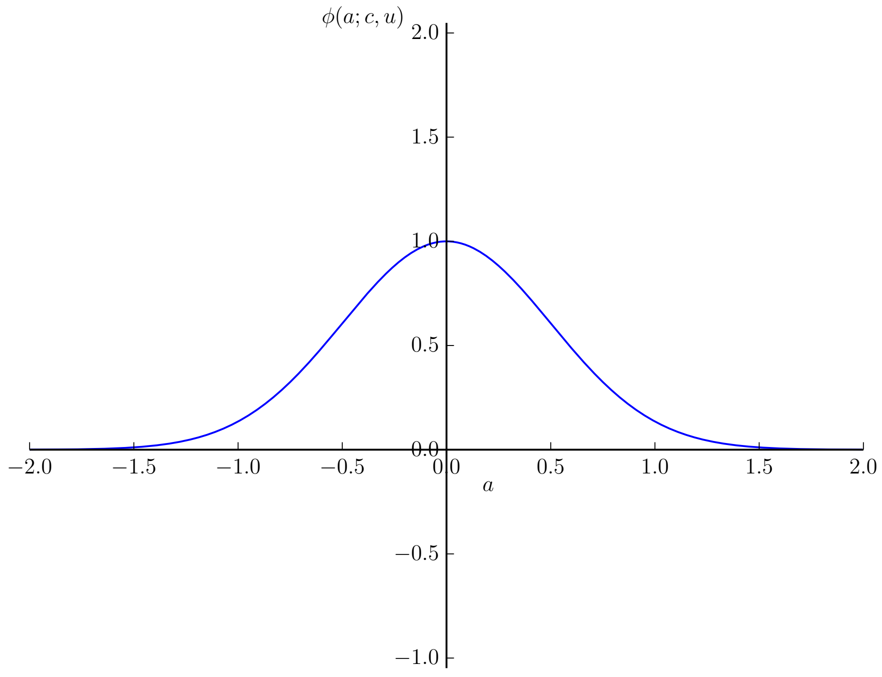


Figure 3.2: Gaussian base function as it has been used in our network layers.

3.4.3 ReLU

We try out ReLU base functions (see Figure 3.3 for illustration) since excellent results have been achieved with them and modifications of them on many computer vision problems (e.g. [38]). The exhibit non-zero gradient on large portions of the input space and have shown excellent performance even in very deep structures. Furthermore Agostinelli et al. [1] as well as He et al. [27] train parameterized ReLU models and achieve excellent results applying them to state-of-the-art recognition benchmarks. The ReLU is in general defined [52] as

$$\sigma(a) = \max(0, a). \tag{3.42}$$

To train sums of ReLUs, we want to distribute ReLUs over the whole input space, therefore we extend the definition above making use of our location parameter c such that

$$\phi(a; c, u) = \max(0, a - c). \tag{3.43}$$

That way, we can shift the ReLUs kink from the origin to any position c on the input space. Since $\phi(a; c, u)$ is not continuous in this case, we compute its derivative analytically like we did for the base functions before, but just divide the input space in the subspacewhere

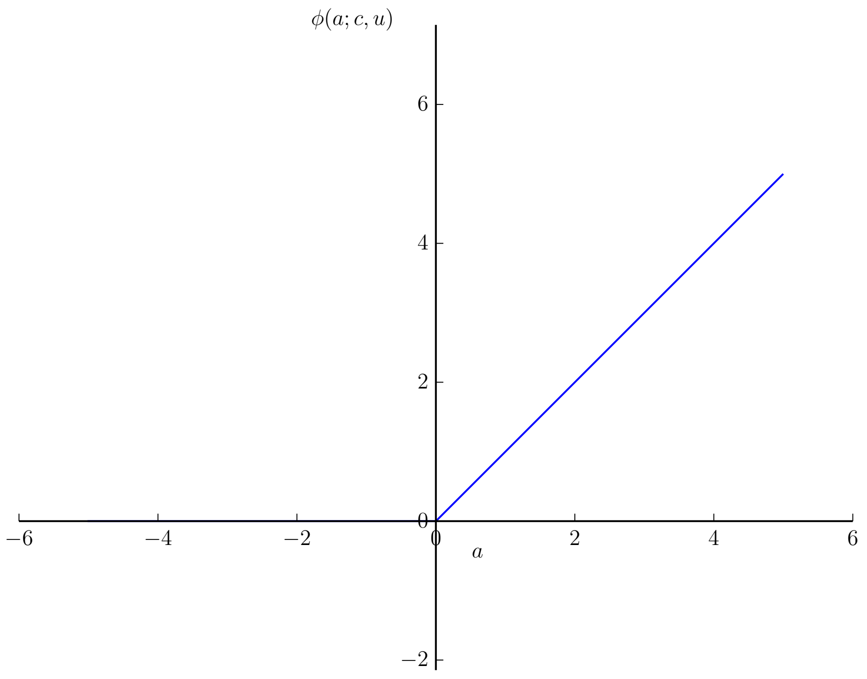


Figure 3.3: ReLU base function as it has been used in our network layers.

$\phi(a; c, u)$ is continuous on the whole subspaceand compute the correct gradient for each subspace. Note that since the gradient is undefined at c, where a kink occurs. Since the gradient $\frac{\partial \phi(a;c,u)}{\partial a}$ is constant though for all $a > c$ as well as for all $a < c$, for the point c we just add c to the subspace $a > 0$ assign the function $\frac{\partial \phi(a;c,u)}{\partial a}$ the corresponding gradient at that point. Thus we get

$$\frac{\partial \phi(a; c, u)}{\partial a} = \begin{cases} 1, & \text{if } (a \geq c) \\ 0, & \text{else} \end{cases} . \tag{3.44}$$

In the same way we obtain

$$\frac{\partial \phi(a; c, u)}{\partial c} = \begin{cases} -1, & \text{if } (c \leq a) \\ 0, & \text{else} \end{cases} . \tag{3.45}$$

Finally, since we do not make use of the shape parameter u in our ReLU base function

$$\frac{\partial \phi(a; c, u)}{\partial u} = 0. \tag{3.46}$$

3.4.4 Triangle

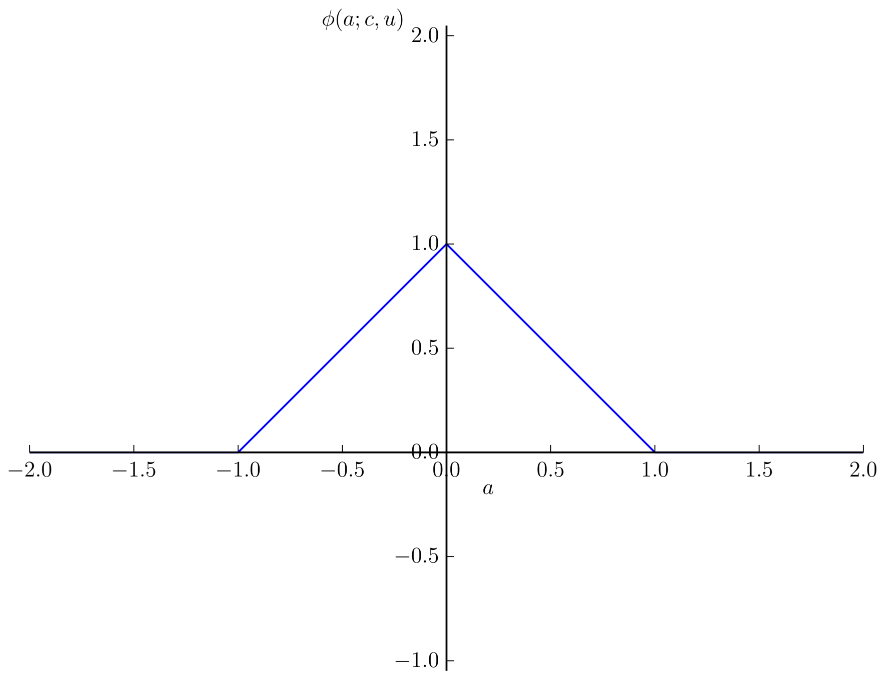


Figure 3.4: Triangle base function as it has been used in our network layers.

Triangle functions are quite similar in shape to Gaussian functions and can therefore be viewed as a rough approximation of them. Nevertheless, triangular functions are much quicker to evaluate than Gaussian functions due to their much simpler definition. We therefore expect triangles to exhibit a similar performance to Gaussians by delivering a piece-wise linear approximation of the activation function learned with Gaussian base functions. We define the triangular function as

$$\phi(a; c, u) = \max(0, 1 - |a - c|). \tag{3.47}$$

Again, we can not compute the derivatives analytically since $\phi(a; c, u)$ is not continuous over the whole input space. Therefore we divide the input space in subspaces again such that the gradient for every subspace can be computed. Then we compute the gradient for each subspace separately and assign the points where $\phi(a; c, u)$ exhibits kinks to a

Activation Function	# Of Parameters Per F Neurons Of Layer l
ReLU	$F^{(l-1)} \cdot F^{(l)}$
APL [1]	$(F^{(l-1)} + 2I) \cdot F^{(l)}$
ALU	$F^{(l-1)} \cdot F^{(l)} + I + 2$
PReLU [27]	$(F^{(l-1)} + 1) \cdot F^{l}$

Table 3.1: Numbers of parameters needed per F neurons of layer l for APL, ALU, PReLU and ReLU activation functions

neighboring subspace. That way, we obtain

$$\frac{\partial \phi(a;c,u)}{\partial a} = \begin{cases} \frac{-(a-c)}{|a-c|}, & \text{if } |a-c| < 1 \\ 0, & \text{else} \end{cases}, \tag{3.48}$$

and

$$\frac{\partial \phi(a;c,u)}{\partial c} = \begin{cases} \frac{a-c}{|a-c|}, & \text{if } |a-c| < 1 \\ 0, & \text{else} \end{cases}. \tag{3.49}$$

Finally,

$$\frac{\partial \phi(a;c,u)}{\partial u} = 0 \tag{3.50}$$

again.

3.5 Parameters Needed By The Model

To ensure a fair comparision between activation units, one has to take care to train networks with the approximately same amount of parameters. We compare our parameters per layer. Consider a fully connected network layer l with $F^{(l)}$ neurons. Table 3.1 shows a parameter comparison between a regular fully connected layer with ReLUs, as well as fully connected layers with APLs, ALUs and PReLUs of the same size respectively. If we use ReLUs as an activation function, we do not train any additional parameters, and thus need only to consider the parameters for the weights between the neurons, which amounts to $F^{(l-1)} * F^{(l)}$ parameters, where $F^{(l-1)}$ and F^{l} is the number of neurons for layers $l-1$ and l respectively.

In an ALU layer, we additionally train the parameter vectors $\mathbf{k}$, s and u, where $\mathbf{k}$ has I parameters and s and u have 1 parameter. This gives us $F^{(l-1)} \cdot F^{(l)} + I + 2$ parameters for a fully connected ALU layer. An APL layer, on the contrary, trains individual functions for every neuron through optimization of amplitudes and centers. Therefore the layer holds $F^{(l-1)} \cdot F^{(l)} + 2 \cdot I \cdot F^{l} = (F^{(l-1)} + 2I) \cdot F^{(l)}$ parameters. Finally, in a fully connected PReLU layer only a single parameter, namely the steepness, is additionally trained for every neuron, and thus its amount of parameters needed amounts to $F^{(l-1)} \cdot F^{(l)} + \cdot F^{l} = (F^{(l-1)} + 1) \cdot F^{l}$.

Based on these observations, we see that we are able to compare our method to networks with ReLU activations without taking additional measures, since the amount of additional parameters is for sensible values of I in the range of 5 to 50 insignificant. Adding a single neuron to a ReLU layer results in $F^{(l-1)}$ additional parameters in that layer, where $F^{(l-1)} \gg I$ for common networks.

Thus, by applying the parameter reduction techniques introduced in Section 3.2 to our framework, we are able to train activation functions using only a low amount of additional parameters contrary to recently proposed methods.

4 Experiments

In Chapter 3 we have proposed a general framework to train activation functions within neural networks by optimizing several parameters of a sum of base functions. In order to verify our ideas in practice, we implement our proposed neuron unit within the Caffe [33] deep learning framework. Using this implementation, we conduct experiments to verify our proposed models, and to find out under which circumstances they perform best in practice. Furthermore, we want to investigate the influence of various single parameters in the model on its overall performance. Finally, we compare our model with recently proposed similar models on state-of-the-art computer vision problems.

For our experiments we will use four different data sets. First, the Modified NIST (MNIST) data set. The MNIST data set contains gray value images of handwritten digits. LeCun et al. [40] constructed MNIST from the NIST image set [26] by mixing the original training setand test set. This was done in order to ensure a better generalization of the classifiers trained on the data set [40]. Moreover, LeCun et al. normalized scale and position of digits in the images for better classifier performance. The MNIST data set consists of 70000 gray value images of handwritten digits, where each image is 32x32 pixels in size. They are separated into a training setof 60000 images as well as a test setof 10000 images. Both sets are labeled with integers from 0 to 9.

The CIFAR-10 and CIFAR-100 data sets are subsets of the Tiny images data set [67]. The tiny images data set consists of 80 million color images, mined from the internet and normalized to a size of 32x32 pixels by Torralba et al. [67]. These images are not labeled, but only loosely associated with nouns listed in the Wordnet[51] lexical database. Krizhevsky et al. created sets of reliable labels for the images contained in the CIFAR-10 and CIFAR-100 data sets. The CIFAR-10 data set contains 60000 images belonging to 10 classes, where every class contains 6000 images. This images are again separated into 50000 training images as well as 10000 test images, which are labeled with integers from 0 to 9. Similarly, the CIFAR-100 data set contains 60000 images of 100 classes, where every class holds 600 images. The separation in training and test set is identical to the CIFAR-10 data set, the image class labels are indicated with integers between 0 and 99.

For the CIFAR-100 data set, Krizhevsky et al. supply a set of 20 coarse labels as well, which comprehend the 100 classes of the data set to larger superclasses. Nevertheless, in this thesis, we only make use of the 100 finer class labels, and ignore the coarse labels in our experiments.

Finally, the Street View House Numbers (SVHN) [53] data set contains labeled images of house numbers obtained from Google Street View ®. The SVHN data set is separated into a training data set containing 73257 images, as well as a test set of 26032 digits. Finally 531131 images, which are easier to recognize, are supplied as an additional training or validation set. All images are labeled with integers between 1 and 10, where the digit *0* is labeled with the integer 10. Since the SVHN data set is the only one that supplies a possible validation set, to be consistent with our method in between data sets, in this thesis, we only make use of the training and test set. Netzer et al. [53] provide the SVHN data set in two different formats. The first version of the data set contains all images in their original resolution, the digits are annotated with bounding boxes. The second version of the data set is more similar to the format of the MNIST data set. The digits in the second version are normalized and cropped to be centered within a color image which is 32X32 pixels in size. Again, to obtain comparable results between data sets in our experiments, we only make use of the latter version of the data set in this thesis.

We will use two different networks for our experiments. For analysis of our unit and the exploration of its parameter space, we conduct experiments on a modified version of LeNet [40], which we call ALeNet (see Table 4.1). ALeNet consists of 2 convolutional layers and a fully connected layer, where we put our activation function behind every convolutional layer as well as the fully connected layer. In all experiments where we make use of the ALeNet architecture, we train for 20000 iterations and use a momentum of $a = 0.9$, a base learning rate of $\eta = 0.01$ as well a as a weight decay of $\lambda = 0.0005$. Since we found it beneficial for the stability of the training process in preliminary experiments, we multiply the global learning rate with 0.1 for the gradient updates inside our units. This is done whenever not mentioned otherwise. All reported numbers are the average over 20 trained networks with the same architecture, but random initialization of the weight parameters, unless otherwise stated. Furthermore, for simplicity and clarity, by default we train only the amplitudes $\mathbf{k}$ in our ALU units, and add the training of shape and input range later, as we analyze the impact of these parameters. Whenever we train shape and input range parameters of our units, we will mention it explicitly in the corresponding experiment.

To investigate the influence of the used base function, initialization and normalization procedures on the performance of our units, as well as the impact of various parameters, we first train on the CIFAR-10 data set. Thereafter, we use the best configuration found from the aforementioned experiments to evaluate the performance of ALUs on the MNIST, CIFAR-100 and SVHN data sets which will all be trained on the ALeNet architecture. For reference, we show the performance of a ReLU baseline for every experiment as well: we replace the ALUs in ALeNet with a regular ReLU.

layer no	layer type	size	weight initialization	bias initialization
1	Convolutional	$(20 \times 5 \times 5)$	xavier	constant
2	ALU/ReLU	-	varying	-
3	Max-Pooling	$(20 \times 2 \times 2)$	-	-
4	Convolutional	$(50 \times 5 \times 5)$	xavier	constant
5	ALU/ReLU	-	varying	-
6	Max-Pooling	$(50 \times 2 \times 2)$	-	-
7	Fully Connected	500 units	xavier	constant
8	ALU/ReLU	-	varying	-
9	Fully Connected	10 units	xavier	constant
10	SoftMax	-	-	-

Table 4.1: ALeNet: Net Architecture used in the first analytic part of our experiments. Our unit is placed behind every convolutional or fully connected layer

layer no	layer type	size	weight initialization	bias initialization
1	Convolutional	$(96 \times 5 \times 5)$	Gaussian	constant
2	ALU/(P)ReLU/APL	-	varying	-
3	Dropout (Rate: 0.25)		-	-
4	Max-Pooling	$(96 \times 3 \times 3)$	-	-
5	Dropout (Rate: 0.25)		-	-
6	Convolutional	$(128 \times 5 \times 5)$	Gaussian	constant
7	ALU/(P)ReLU/APL	-	varying	-
8	Dropout (Rate: 0.25)		-	-
9	Avg-Pooling	$(128 \times 3 \times 3)$	-	-
10	Dropout (Rate: 0.25)		-	-
11	Convolutional	$(256 \times 5 \times 5)$	Gaussian	constant
12	ALU/(P)ReLU/APL	-	varying	-
13	Dropout (Rate: 0.25)		-	-
14	Avg-Pooling	$(256 \times 3 \times 3)$	-	-
15	Dropout (Rate: 0.5)		-	-
16	Fully Connected	2048 units	Gaussian	constant
17	ALU/(P)ReLU/APL	-	varying	-
18	Dropout (Rate: 0.5)		-	-
19	Fully Connected	2048 units	Gaussian	constant
20	ALU/(P)ReLU/APL	-	varying	-
21	Dropout (Rate: 0.5)		-	-
22	Fully Connected	10 units	Gaussian	constant

Table 4.2: DeepNet: Net Architecture used to compare ALUs to ReLU APL and Maxout units delivering state of the art performance

Finally, we compare our proposed unit to a similar state-of-the-art methods recently proposed by Agostinelli et al. [1] and He et al. [27], on the net architecture Agostinelli et al. use. In this thesis, we refer to this network as DeepNet, its structure is shown in Table 4.2. For the training of networks with the DeepNet architecture, we train for 90000 iterations and use a momentum of $a = 0.5$, which is gradually increased towards $a = 0.9$, a base learning rate of $\eta = 0.01$ as well a as a weight decay of $\lambda = 0.001$. All experiments are run on a single Nvidia® GeForce GTX 780 Ti graphics processing unit with 3 GiB Memory.

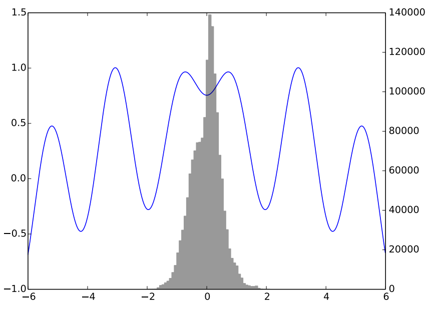
The remaining chapter is structured as follows: we start by investigating the impact of different base functions on the classifier accuracy in Section 4.1. After we have found the best performing base function for our network and data, we investigate the suitability of different amplitude initializations for our units in Section 4.2. This experiment is followed by a review on how and why decreasing the learning rate of the fully connected ALU layers affects accuracy in Section 4.3. This review results in an experiment testing the effectiveness of forced regularization as an overfitting reduction method for the amplitudes $\mathbf{k}$ in our ALU layers as described in Section 4.4, followed by an examination of the relationship between the number of used base functions I and the accuracy in Section 4.5. Thereafter we research the relationships of the input range and shape parameters s and u as well as their optimization with accuracy in Section 4.6 and Section 4.7. Finally, we investigate the performance ALeNet with our units on the MNIST, CIFAR-100 and SVHN data sets in Section 4.8 and compare its performance with similar state-of-the-art methods on DeepNet in Section 4.9.

4.1 Impact Of Base Function

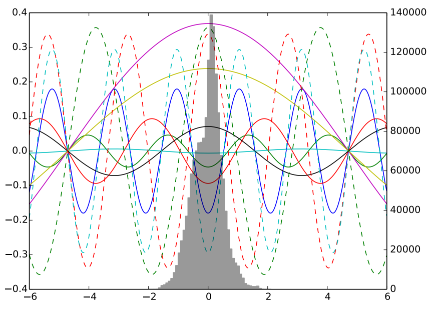
First, we want to determine which base function delivers the best results when used in ALUs in a network like the one we introduced. Therefore we train 5 similar groups of networks using the ALeNet architecture described in Table 4.1. For every set of networks, we use a different kind of activation function after the convolutional and fully connected layers: As a baseline, we use a group of 20 networks, trained with ReLUs. Furthermore, we train 4 groups of 20 networks with our own proposed units, namely ALUs, where we use a different base function for every network group. Therefore, we train 20 networks with a cosine base, 20 networks with a Gaussian base, 20 networks with a ReLU base, as well as 20 networks with a triangle base. The base function amplitudes initialized with randomly generated values which are equally distributed between -0.5 and 0.5. Randomly generated values are drawn individually for the parameters of every single network.

Figure 4.1 shows example initializations of ALU layers with cosine, Gaussian, ReLU and triangle base functions.

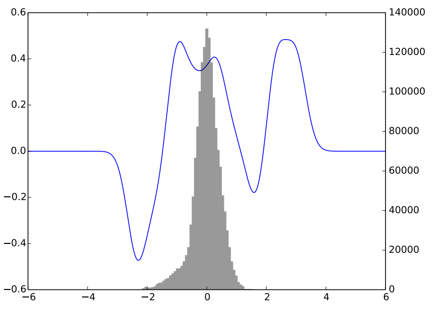
The input range parameter is set as $s = 3$ because of the so-called 65-95-99.7 rule: it states the relative amount of values lying in intervals of 1, 2, and 3 σ around the mean for Gaussian distributions. Hence, for any Gaussian distribution, 65% of values are within



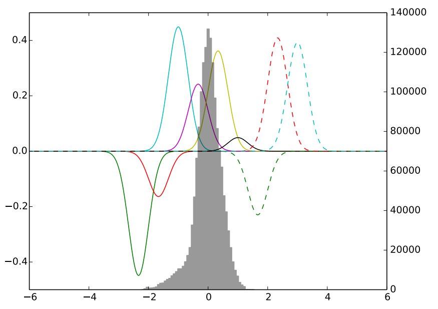
(a) Initial activation function (cosine)



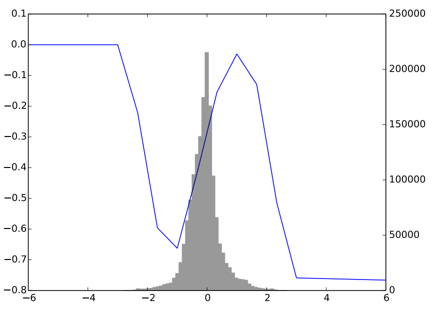
(b) base functions (cosine)



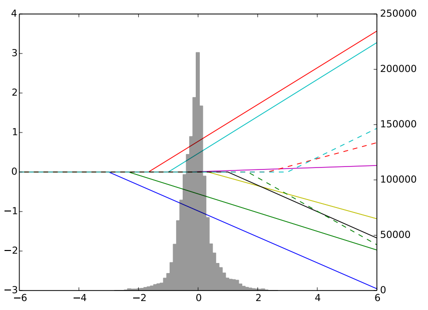
(c) Initial activation function (Gaussian)



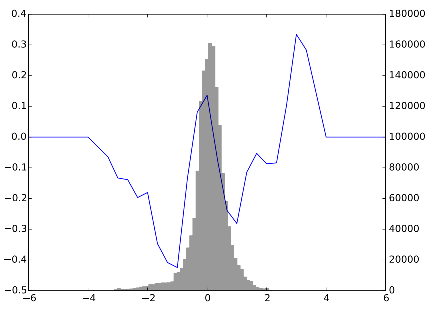
(d) base functions (Gaussian)



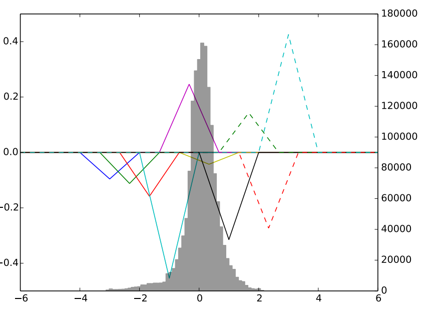
(e) Initial activation function (ReLU)



(f) base functions (ReLU)



(g) Initial activation function (triangle)



(h) base functions (triangle)

Figure 4.1: Left column shows examples for Initial activation functions of the ALU layer for different base functions when initializing amplitudes with uniform noise. The corresponding base functions are shown (in different colors for better visibility) in the right column. For reference, a histogram of the input data to the layer (i.e. the accumulated input to all units) is shown.

base function	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
cosine	64.79 ± 2.88	2.1622 ± 0.1845	0.0241 ± 0.0539
gaussian	67.01 ± 1.50	1.5443 ± 0.0969	0.0021 ± 0.0001
relu	58.75 ± 11.86	2.2376 ± 0.3109	0.1819 ± 0.5043
triangle	67.80 ± 1.50	1.5086 ± 0.1010	0.0022 ± 0.0001

Table 4.3: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test sets of the CIFAR-10 data set for our baseline network as well as the networks using ALU layers with different base functions. Results are averaged over 20 networks each.

the range of $\pm 1\sigma$ around the mean, 95% of all values of said Gaussian distribution lie within $\pm 2\sigma$ and 99.7% of all values lie within $\pm$ 3 σ. Thus if we assume the input data for our layer to be Gaussian distributed with a mean of 0 and a standard deviation of 1, setting the input domain for our activation functions to [-3,3], enables us to cover 99% of the input values given to our layer.

This rule does of course not apply to cosine base functions, where s per definition has more the interpretation of the maximal angular frequency used in the initialization. But since the training with cosine activation functions performs acceptably when initialized that way, as we will see in the results, and no obviously better value to use lies at hand, we initialize the units using cosine base functions with $s = 3$ as well.

u will be initialized like we derived it since the beginning of this thesis, dependent from the input range s, considering the spacing of base function centers:

$$u = \frac{s}{(I-1)}. \tag{4.1}$$

Note that u only applies for Gaussian base functions though, since they are the only ones incorporating a shape factor.

Table 4.3 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network sets with ALUs as well as the baseline set. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional, fully connected and ALU layers.

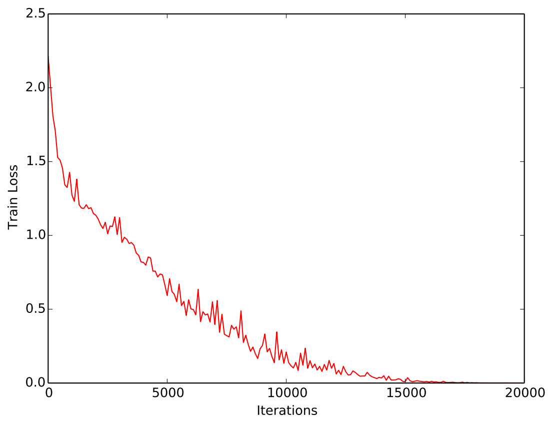
As we can see, with our current configurations, ReLU units outperform ALU units by at least two percent in terms of accuracy. The best results are achieved using triangular base functions, followed by Gaussian base functions, which are quite similar in shape and therefore deliver a similar result. On the other hand, cosine base functions show a significantly worse performance followed by ReLUs which perform the worst in this experiment.

Standard deviations of accuracy are for all networks using ALUs higher than for the baseline. This indicates that their performance is, as expected, dependent on the initialization of their base function amplitudes. The high standard deviation in accuracy for the ReLU network might point to a reason for the bad behavior of the networks from the ReLU group. This high standard deviation indicates instabilities in the training process causing the prediction of the corresponding network to fail completely. This might happen for the networks in the ReLU set since ReLUs are, contrary to Gaussian and triangles, non-zero over very large parts of the input space and, contrary to cosine base functions, not bounded in their output domain. For sums of ReLUs, very high output values will therefore occur in the forward path of the network, which might destabilize it due to large gradient steps or overflows. A possible measure against this effects, as discussed above, might be the choosing of a different initialization patterns, a smaller learning rate and less base functions. In this set of analytic experiments we will refrain from this measures though, in favor of fair comparison and feasibility.

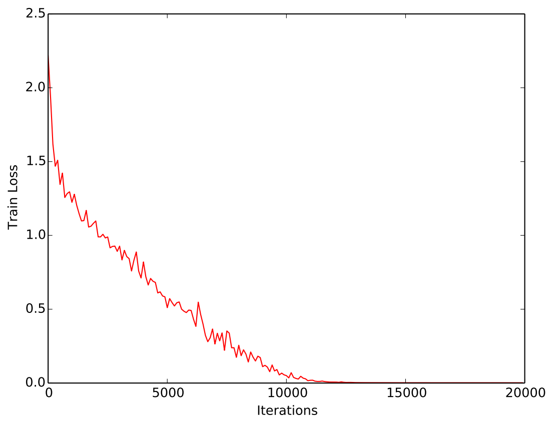
When taking a look at the test loss, one discovers an interesting fact: despite the mean accuracy of the triangle and Gaussian ALU networks being lower than the one of the baseline, their mean test losses are slightly lower than the baseline mean test loss. Several reasons might be accounted for this observation: First, while loss and accuracy measure similar things and are strongly correlated, their relationship is complex, and therefore, for two slightly different accuracies, the respective losses of the networks might show a higher difference. Therefore, one or two significantly lower losses in our Gaussian and triangle sets might have skewed the mean test loss of these network sets to be lower than the baseline test loss. The higher standard deviation of test loss on the triangle and Gaussian network sets support that explanation. A second explanation lies also in the nature of the multinomial logistic loss, which sinks as the networks confidence values in the correct class of a sample rise. Therefore it might be possible that the network, although making less correct predictions overall, is more confident concerning it's correctly classified samples than the baseline network.

When viewing the train losses, the most prominent observation is, that the train losses of the triangle and Gaussian network sets are relatively low compared to the loss of the baseline despite the baselines better performance and similar test loss scores. Since the corresponding standard deviation of the train loss is low as well, this might indicate that we slightly overfit our training data. To gain further information of the training process of ALUs, we investigate the training loss as a function of the training iteration for some example networks of each base function set, as shown in Figure 4.2. We see that ALU networks seem to reach convergence faster than the Baseline network. This hardens our suspicion that we might observe overfitting here in our networks. Another remarkable observation can be made concerning the smoothness of the training process: The networks using ALUs start out with a higher loss and their loss curves are less smooth. The reason for this phenomenon might be the more complex optimization objective due to the non-convex irregular activation functions used in the ALU networks.

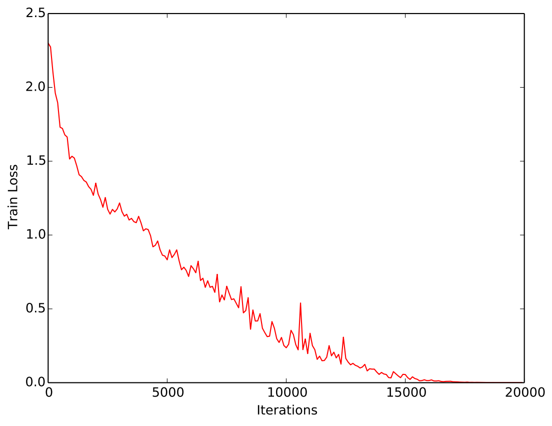
We take a look at the trained functions for all four network sets: Figure 4.3 shows the trained activations functions for a sample network out of our set of trained ALU networks with a cosine base.



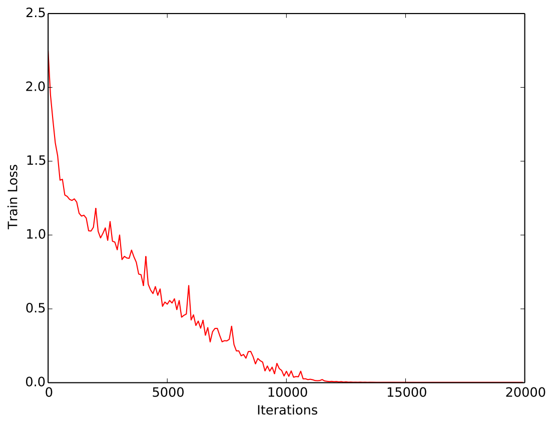
(a) Train loss, cosine base.



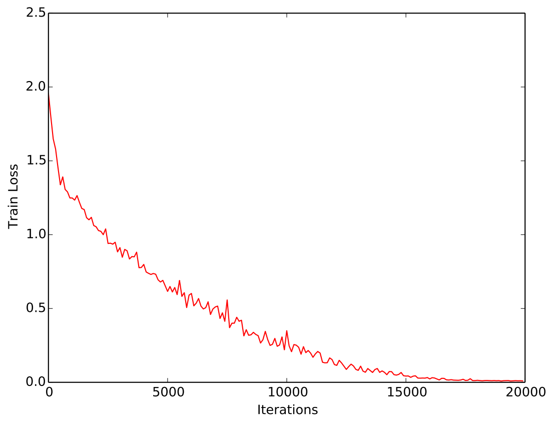
(b) Train loss, Gaussian base.



(c) Train loss, ReLU base.

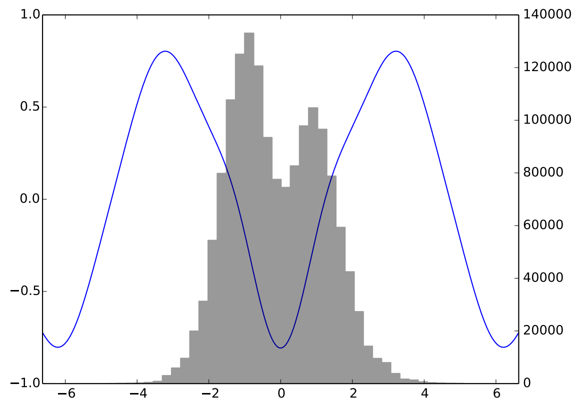


(d) Train loss, triangular base.

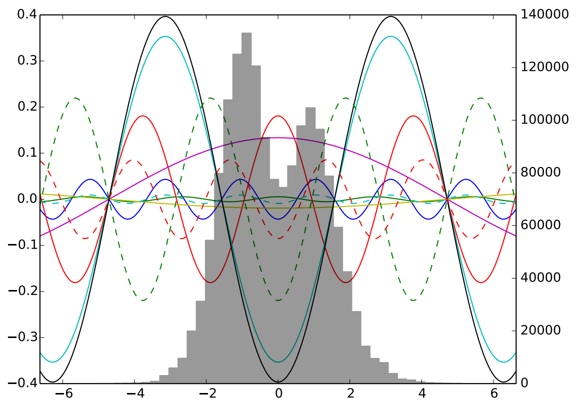


(e) Train loss, Baseline.

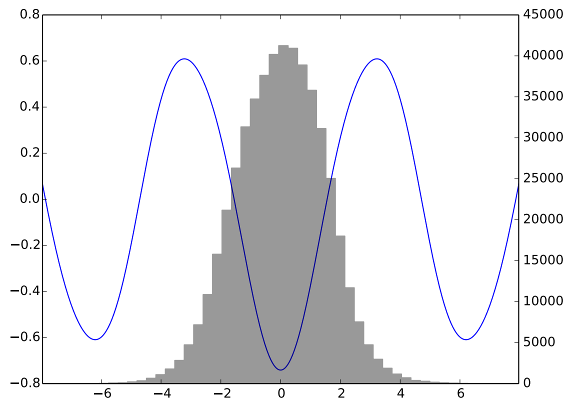
Figure 4.2: Multinomial logistic loss over 20000 iterations. We train four sample networks on the ALeNet architecture using ALUs, each trained with a different base.



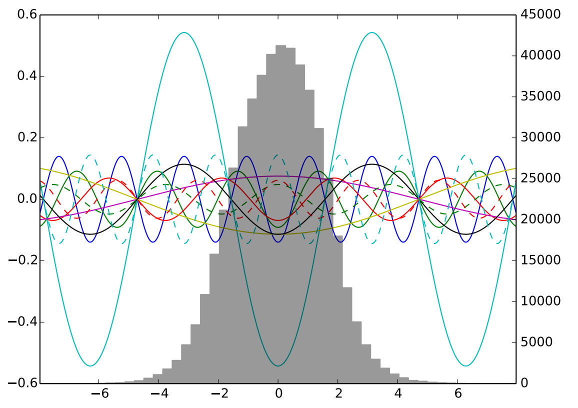
(a) Trained activation function in layer 2 (cosine base).



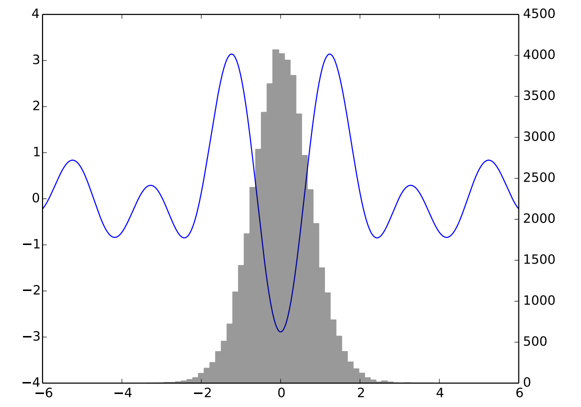
(b) Base functions in layer 2.



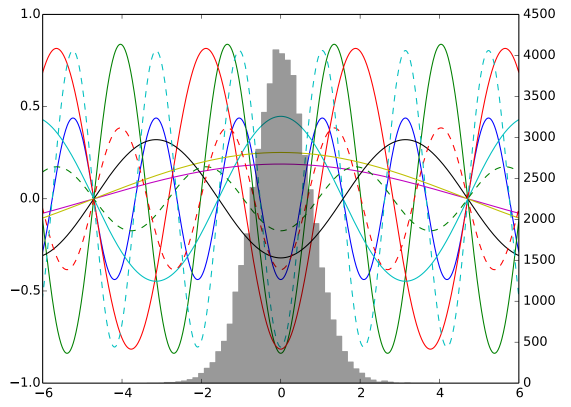
(c) Trained activation function in layer 5 (cosine base).



(d) Base functions in layer 5.



(e) Trained activation function in layer 8 (cosine base).



(f) Base functions in layer 8.

Figure 4.3: Trained activation functions of ALU layers after first convolutional layer, after second convolutional layer and after the first fully connected layer. The trained activation functions are shown in the left column, while the cosine base functions composing them are shown in the right column.

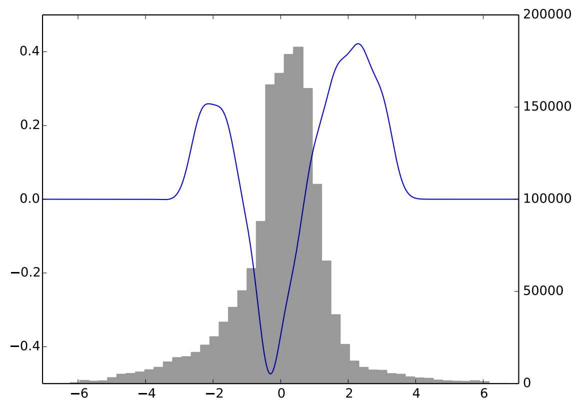
Additionally a histogram of the distribution of the input data to the ALU layer is given, i.e. this histogram is computed over the inputs of all neurons together. Thus one needs to keep in mind that the distribution of the input space for individual neurons might vary. The most common feature for all three activation functions is that they seem to enforce a very strong on-off behavior for the input space where the most samples are located. This might enable the network to order samples better to subspaces in the sense of a maximum margin classification. An additional interesting fact is the increasing complexity of the learned functions as their distance from the input layer increases: While the first function is composed of a single dominant harmonic to which the remaining harmonics simply make small additions, the second layer is more involved and creates a more complex oscillation. The third layer finally, is very difficult to interpret solely through visual inspection, but it seems that the purpose of this compositions is to attenuate the functions peaks in certain regions. This behavior might enable the network to separate between even more classes by separating between high and low positive values. Note that this is not detectable with a single linear neuron modeling a hyper plane, but it might be well detectable using combinations of neurons. Activation functions that allow higher layers to separate between more classes make sense in the context that feature representation within a neural network is enforced to be more and more high-level and therefore, expressive closer to the networks output stage. Therefore the activation function of the last layer might simply adapt to be as expressive as possible.

We find our theory of ALU units attempting to train as strongly oscillating activation functions as possible confirmed, when we take a look at the activation functions trained by a sample network from the Gaussian network set as shown in Figure 4.4.

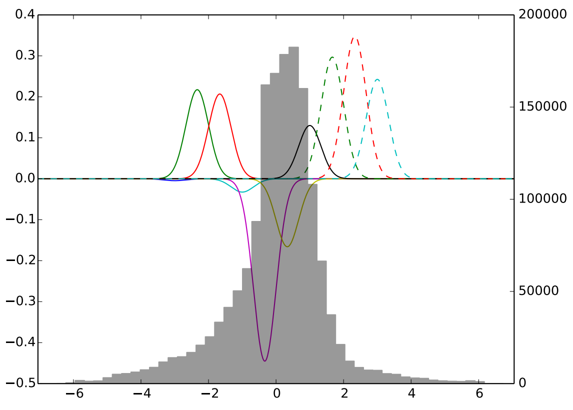
While, contrary to a cosine base, the Gaussian base functions of course do not spread out over the whole definition space but are locally confined, the trained patterns are pretty similar nevertheless: In all cases, the units attempt to maximize the distinguishability of the input samples by training functions with several sharp peaks. Additionally, we can, like for the network with the cosine base, see the spikes attenuating in the less populated regions of the input space as the layers come closer to the output layer of the network.

Little surprising, the sample of the network set trained with triangle base functions, which are roughly the shape of Gaussian, shows very similar patterns in it's trained activation functions, which can be seen in Figure 4.5.

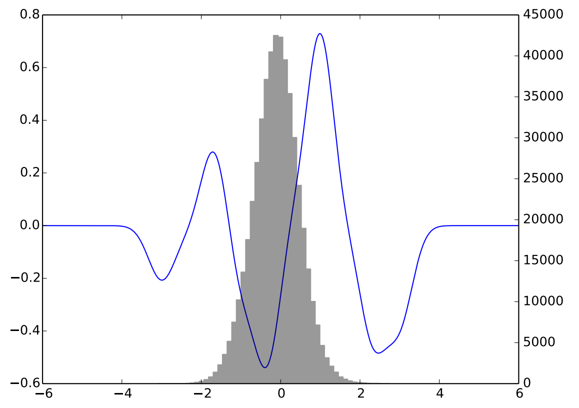
The activation functions of the sample network from the set of networks trained with ReLUs as base functions make the only exception in this pattern as shown in Figure 4.6. Since ReLUs have, except for the single kink in the origin, constant gradients in positive and negative subspaces of the input space, they are less likely to create functions with the high dynamic as seen in the plots above for the other base functions. Nevertheless, the networks trained with ReLU base functions seem to attempt to maximize the On-Off-behavior of their activation functions by forming bows. In general though, the network training this activation function will mostly classify though sparsity of input, i.e. distinguish unit outputs which are zero from those which are not. The fact that the network has



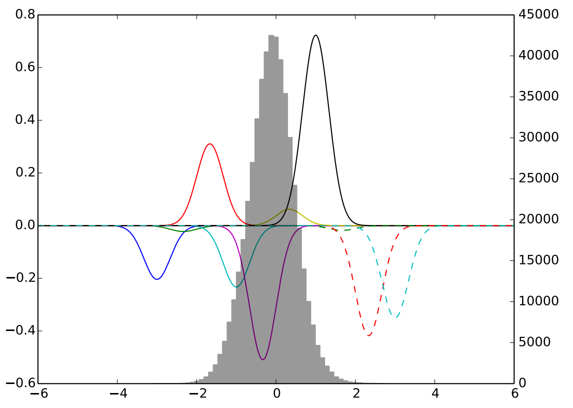
(a) Trained activation function in layer 2 (Gaussian base).



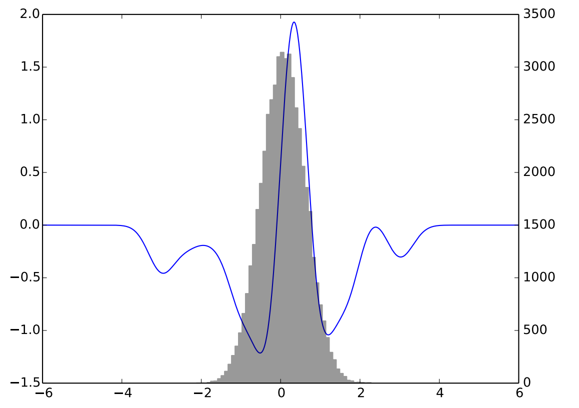
(b) Base functions in layer 2.



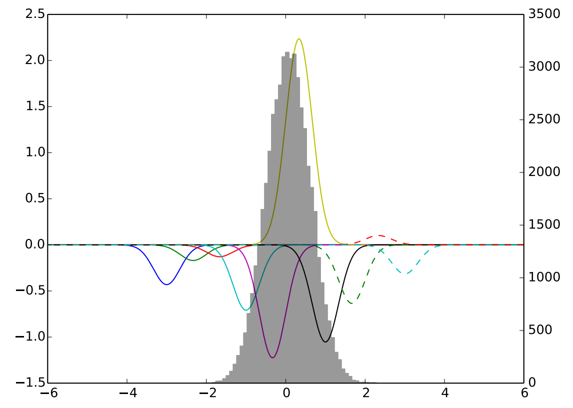
(c) Trained activation function in layer 5 (Gaussian base).



(d) Base functions in layer 5.

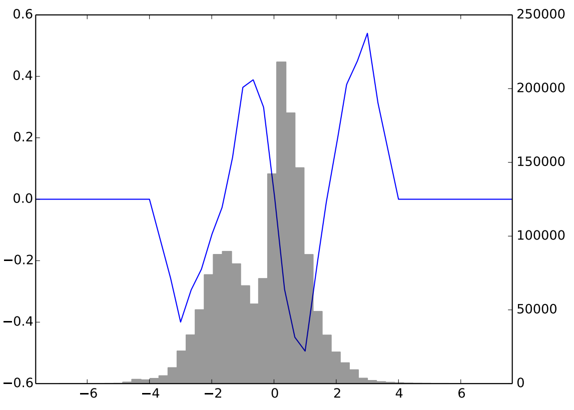


(e) Trained activation function in layer 8 (Gaussian base).

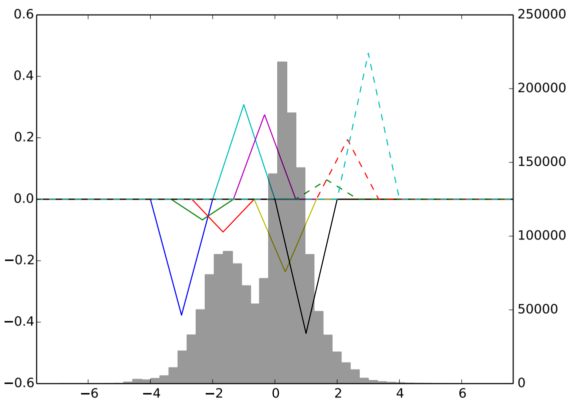


(f) Base functions in layer 8.

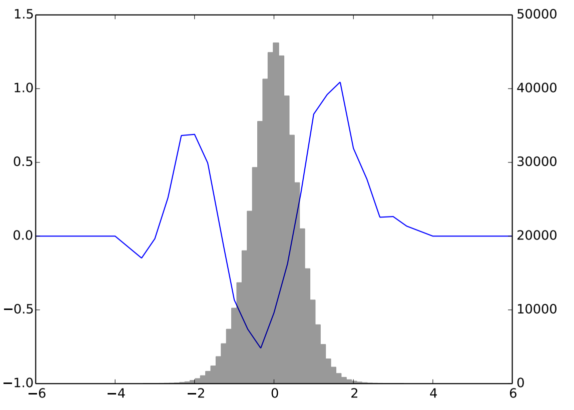
Figure 4.4: Trained activation functions of ALU layers after first convolutional layer, after second convolutional layer and after the first fully connected layer. The trained activation functions are shown in the left column, while the Gaussian base functions composing them are shown in the right column.



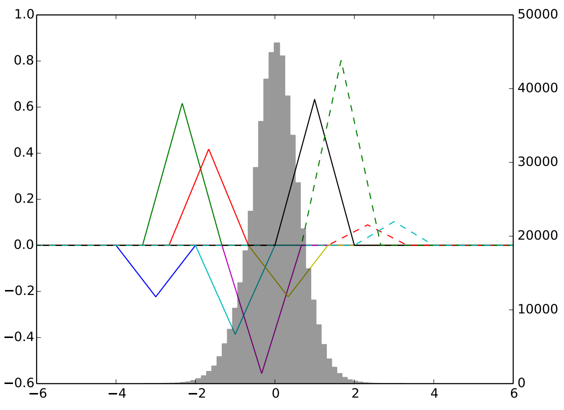
(a) Trained activation function in layer 2 (triangle base).



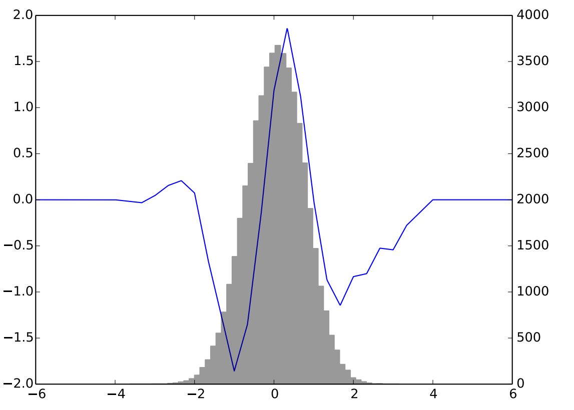
(b) Base functions in layer 2.



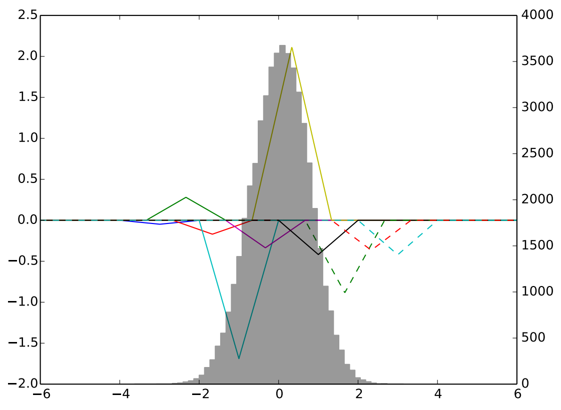
(c) Trained activation function in layer 5 (triangle base).



(d) Base functions in layer 5.



(e) Trained activation function in layer 8 (triangle base).



(f) Base functions in layer 8.

Figure 4.5: Trained activation functions of ALU layers after first convolutional layer, after second convolutional layer and after the first fully connected layer. The trained activation functions are shown in the left column, while the Gaussian base functions composing them are shown in the right column.

spread the input samples over the whole input space for layers 5 and 8, as can be seen in the corresponding input histograms, supports this assumption. Most likely an unsuitable initialization is accountable for the observed activation functions.

In a nutshell, we found out that ALUs utilizing the triangle base function delivered the best results so far, therefore we will conduct our further analytic experiments with them.The base functions we train exhibit very high dynamic and seem built to encode as much state information as possible in the network. Furthermore, have observed overfitting effects, which we will investigate in depth in later experiments. But since these the shape of these functions might be initialization dependent, and also because we have seen that initialization of our amplitudes can make the difference between success and failure of our networks, we will investigate the impact of several different initializations on the networks accuracy, well as well as its activation functions, in the next experiment.

4.2 Impact Of Initialization

In this experiment we investigate the impact of different amplitude initializations on the accuracy of ALU networks. We initialize layer amplitudes $\mathbf{k}$ with a constant value $a_i = 1$, Gaussian distributed random values, and a ReLU-like function initialization

$$k_i = \max(0, c_i), \tag{4.2}$$

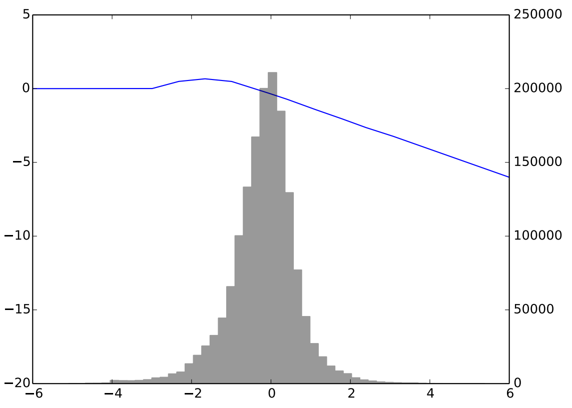
where k_i and c_i are amplitude and location of the base function with index i. Additionally, we try a similar initialization as the one proposed by Agostinelli et al. [1] for APL units. We initialize the units amplitudes $\mathbf{k_i}$ with uniformly distributed random values, as already used in Section 4.1, only that now we add a ReLU function as bias term in front of the optimizeable sum:

$$g(x) = \max(0, x) + \sum_{i=1}^{I} k_i \cdot \phi(\cdot). \tag{4.3}$$

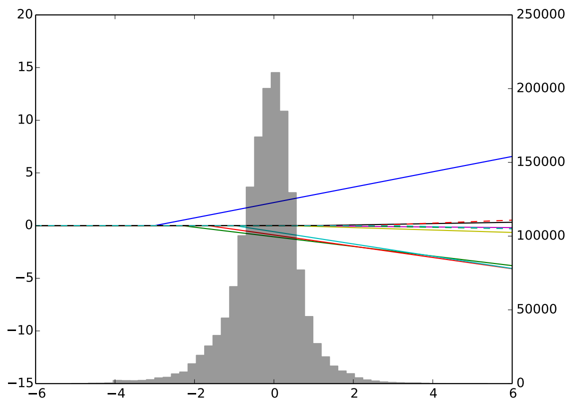
Examples for resulting activation function initializations are shown in Figure 4.7.

We again train a group of 20 networks for each initialization and compare them to the previously trained baseline and uniformly initialized ALU networks from the previous experiment.

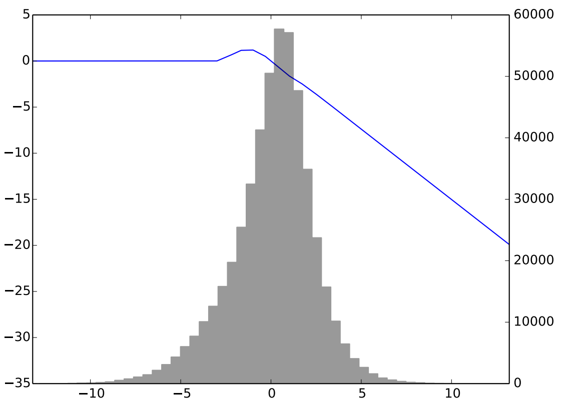
Table 4.4 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups with ALUs as well as the baseline group. Results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional, fully connected and ALU layers. Most interesting, we have slightly outperformed the baseline in terms of accuracy by initializing the ALU units in ALeNet with a ReLU-like function. The initialization using uniform random noise with a ReLU bias is only slightly worse than the former two, but shows a slightly higher standard deviation in accuracy. The networks with



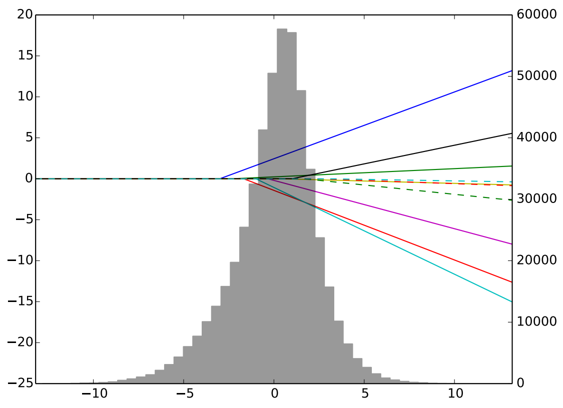
(a) Trained activation function in layer 2 (ReLU base).



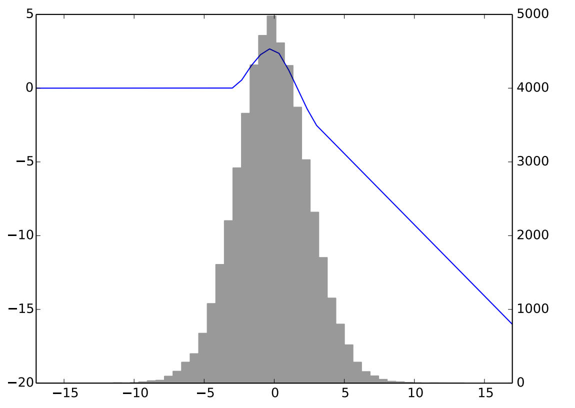
(b) Base functions in layer 2.



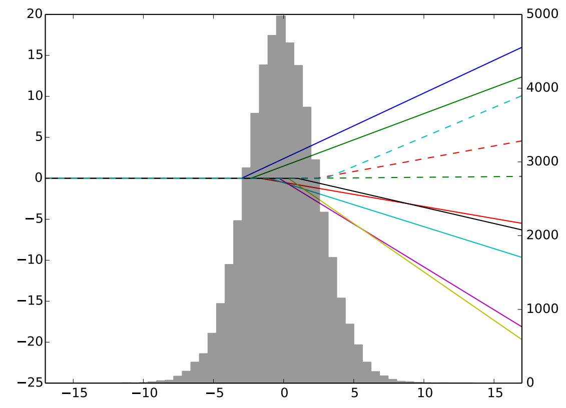
(c) Trained activation function in layer 5 (ReLU base).



(d) Base functions in layer 5.

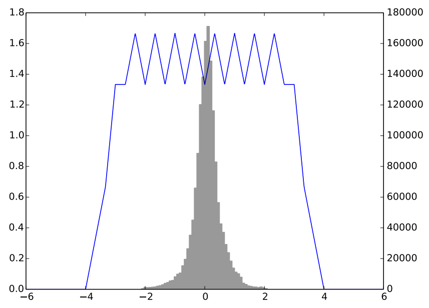


(e) Trained activation function in layer 8 (ReLU base).

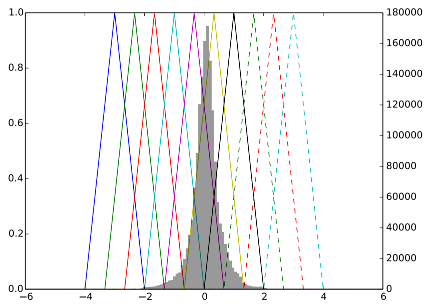


(f) Base functions in layer 8.

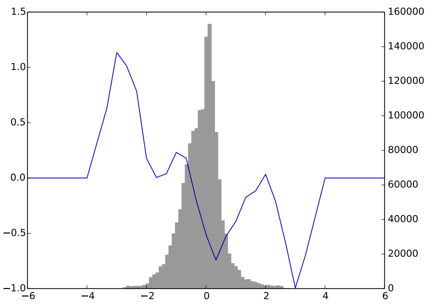
Figure 4.6: Trained activation functions of ALU layers after first convolutional layer, after second convolutional layer and after the first fully connected layer. The trained activation functions are shown in the left column, while the ReLU base functions composing them are shown in the right column.



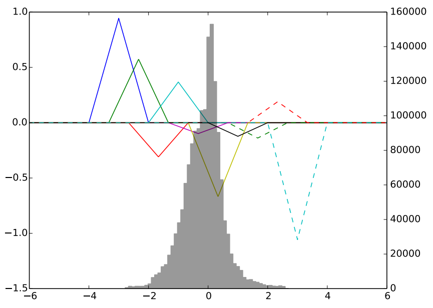
(a) Initial activation function (constant).



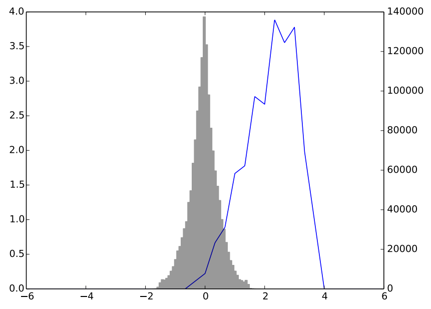
(b) base functions (constant).



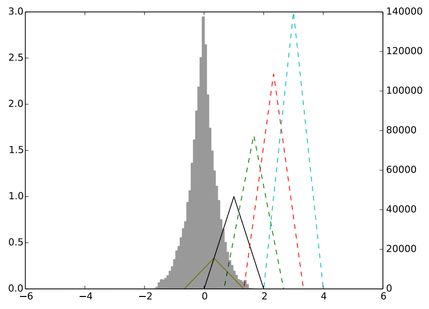
(c) Initial activation function (Gaussian).



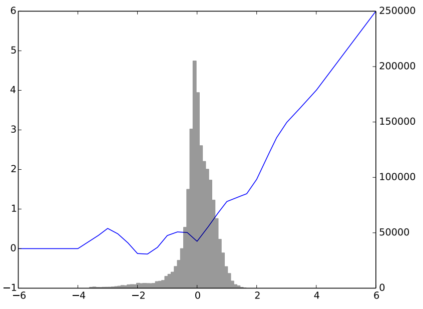
(d) base functions (Gaussian).



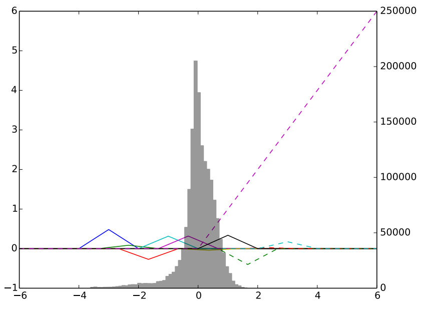
(e) Initial activation function (ReLU-like).



(f) base functions (ReLU-like).



(g) Initial activation function (uniform init.+ ReLU).



(h) base functions (triangle).

Figure 4.7: Left column shows examples for Initial activation functions of the ALU layer for constant, Gaussian, ReLU-like and uniform with ReLU bias initializations. The corresponding base functions are shown (in different colors for better readability) in the right column. For reference, a histogram of the input data to the layer (i.e. the accumulated input to all units) is shown.

initialization	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
constant	11.98 ± 4.51	2.2738 ± 0.0785	2.2717 ± 0.0832
gaussian	66.66 ± 2.30	1.5663 ± 0.1566	0.0108 ± 0.0314
relu added	69.05 ± 0.97	1.4199 ± 0.0622	0.0026 ± 0.0004
relu-like	69.37 ± 0.44	1.4105 ± 0.0306	0.0025 ± 0.0001
uniform	67.80 ± 1.50	1.5086 ± 0.1010	0.0022 ± 0.0001

Table 4.4: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test sets of the CIFAR-10 data set for our baseline network as well as the networks using ALU layers with different initializations. Triangle base functions have been used, results are averaged over 20 networks each.

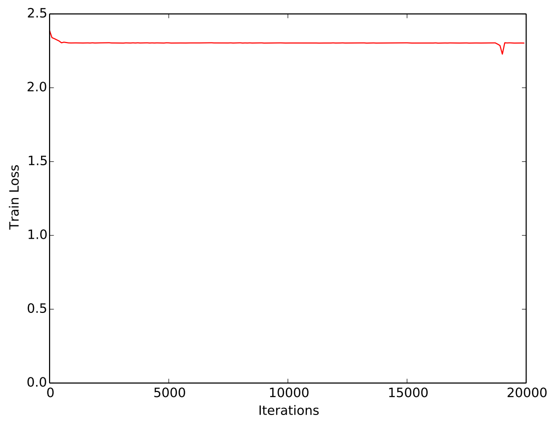
uniformly initialized accuracies without a bias function follow, outperforming networks with Gaussian and constant amplitude initializations. Little surprising, the networks with constant amplitude initialization fail completely, since the resulting activation functions show only very little gradient in interesting regions of the input space, and thus do not permit gradients to flow through these units. Again, train and test losses are for the best performing (ReLU-like and ReLU bias initialized) ALU networks slightly lower. As we elaborated in Section 4.1, this might indicate that ALU networks might be more confident about their correctly estimated classes compared to the baseline.

To examine more closely how and why the networks with constant initializations failed, we examine the train loss as a function of the training iteration for sample networks of the different initializations in Figure 4.8

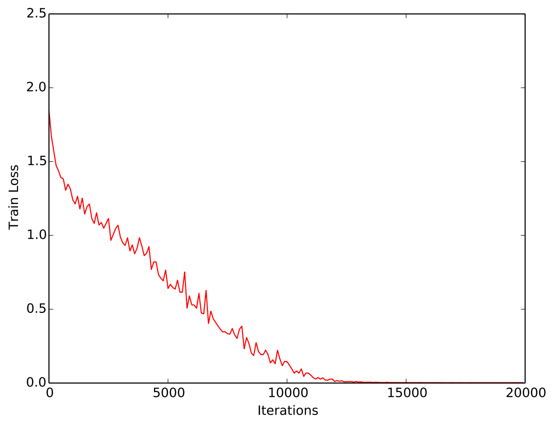
As expected, the loss for the network changes slightly in the beginning, but becomes static pretty quickly, usually an indication that the network has set it's output identically 0 since there are no local minima in reach bearing a smaller loss than loss of an identical output for every single input sample. For the remaining losses, all networks using ALUs seem to decrease the training loss more quickly than the baseline network, even the network with the Gaussian initialization, which performs significantly worse than the baseline. A possible reason for this effect might be the network changing the overall magnitude of the ALUs amplitude to decrease the loss. This is undesired behavior since the network shall minimize the loss function by learning meaningful combinations of base functions instead of changing their overall magnitude.

When taking a look at the learned activation functions for the constant networks, as displayed in Figure 4.9, we see our theory confirmed that the network has set its output to zero: It has done so by distributing the input values of layer 5 in a such a way that its output is identically zero.

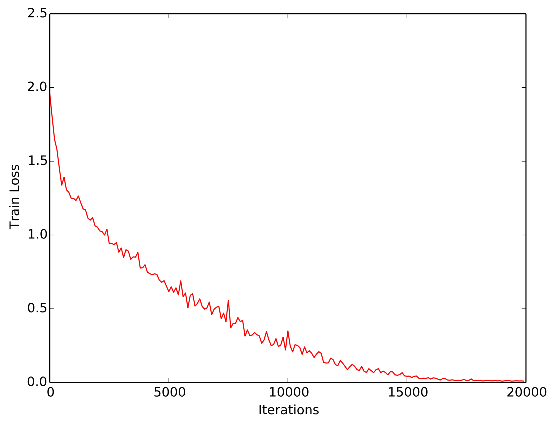
The activation functions used in the ALU network initialized with a ReLU-like initialization are shown in Figure 4.10. They have largely kept their form but exhibit additional positive and negative peaks which again make them look similar like the functions trained in the previous experiment. For the uniformly initialized ALU networks with ReLU bias,



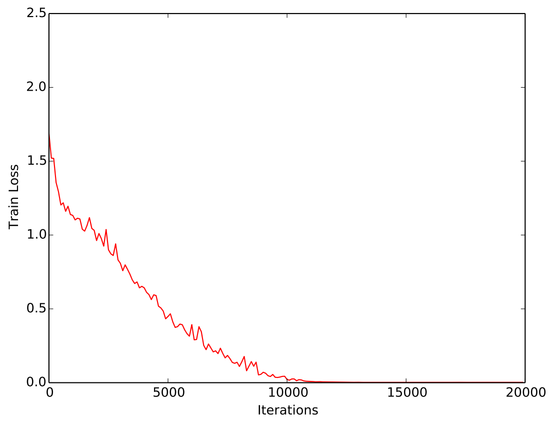
(a) Train loss, constant initialization.



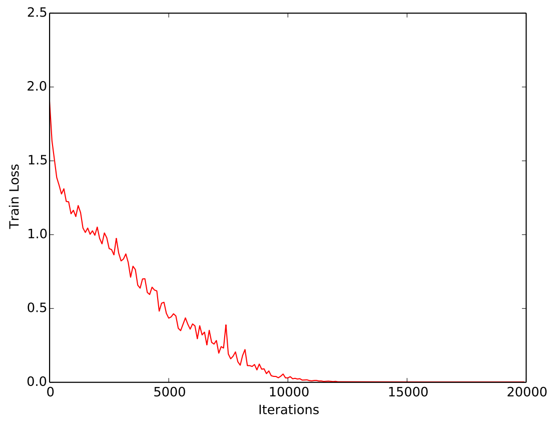
(b) loss vs. iterations, Gaussian initialization.



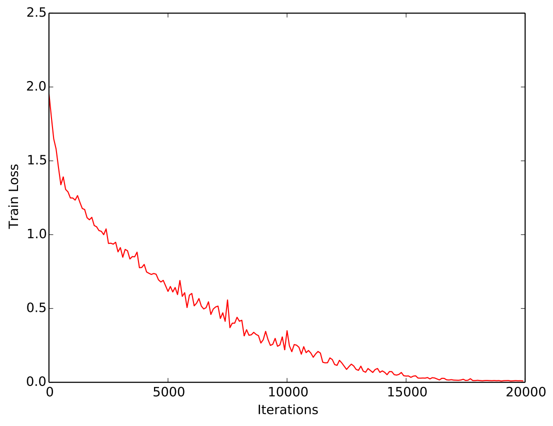
(c) Train loss, uniform initialization.



(d) Train loss, uniform i. with added ReLU.

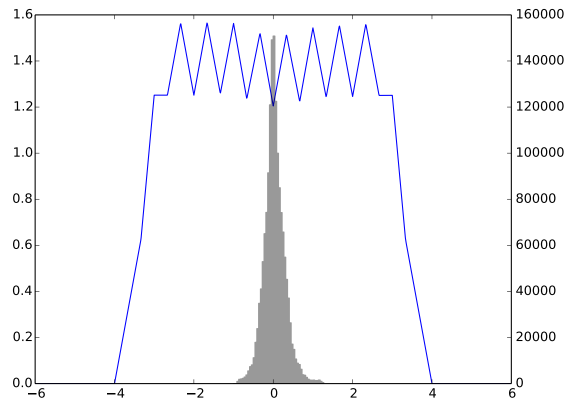


(e) Train loss, ReLU-like initialization.

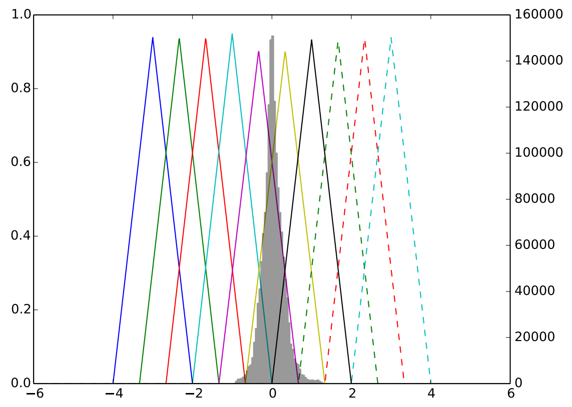


(f) Train loss, Baseline.

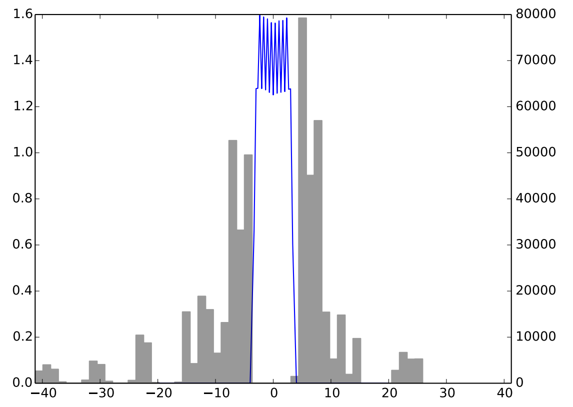
Figure 4.8: Multinomial logistic loss over 20000 iterations training five sample ALeNet networks using ALUs, as well as a ReLU-like baseline. Each ALU network is trained with a different initialization.



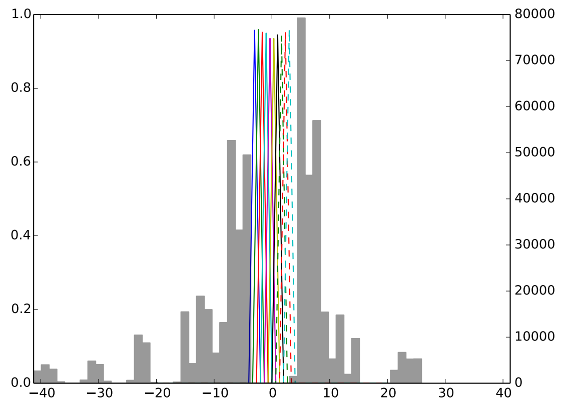
(a) Trained function in layer 2 (constant init.).



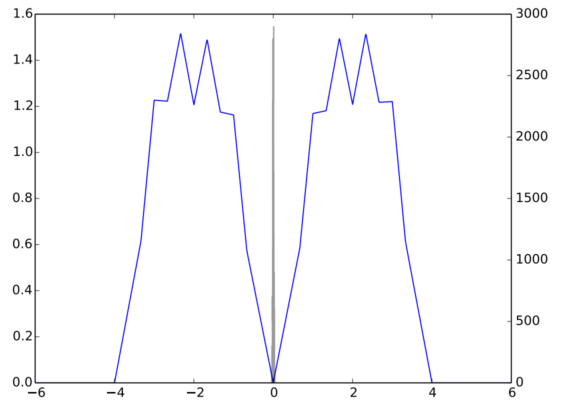
(b) Base functions in layer 2 (constant init.).



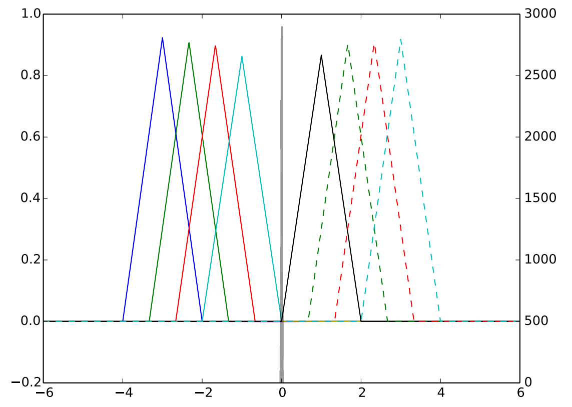
(c) Trained function in layer 5 (constant init.).



(d) Base functions in layer 5 (constant init.).



(e) Trained function in layer 8 (constant init.).



(f) Base functions in layer 8 (constant init.).

Figure 4.9: Trained activation functions of ALU layers for of a sample network with constant amplitude initialization. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

lr policy	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
ReLU	72.03 ± 0.54	1.1721 ± 0.0243	0.0073 ± 0.0003
$\frac{\eta}{10}$	69.37 ± 0.44	1.4105 ± 0.0306	0.0025 ± 0.0001
$\frac{\eta}{100}$	71.08 ± 0.70	1.2125 ± 0.0268	0.0048 ± 0.0001
$\frac{\eta}{1000}$	71.59 ± 0.57	1.1627 ± 0.0265	0.0068 ± 0.0005

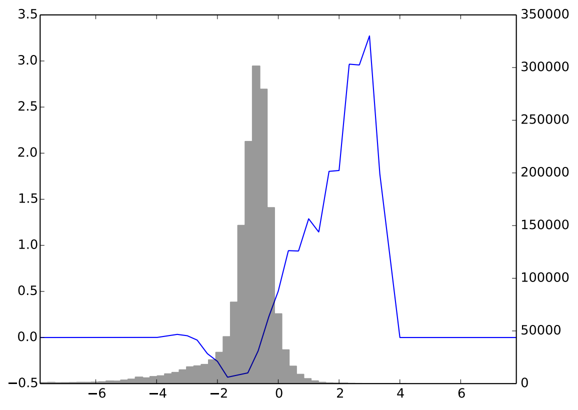
Table 4.5: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test setsof the CIFAR-10 data set for our baseline network as well as the networks using ALU layers where the learning rate has been adjusted in layer 8. Results are averaged over 20 networks each.

we observe a similar effect (see Figure 4.11). Finally, the activation functions of the network utilizing a Gaussian (seen in Figure 4.12) Initialization look similar to the ones trained with uniform initialization shown in Figure 4.5.

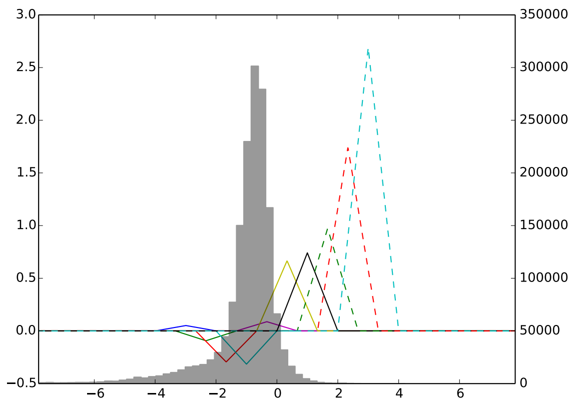
In a nutshell we have reached slightly better performance than the baseline by using ReLU-like initializations for our activation functions. Furthermore we have tried an initialization similar to the one of Agostinelli [1], which performs pretty well too, but slightly worse than the baseline, and has moreover a higher standard deviation in accuracy. The learned functions look pretty similar for ReLU, Gaussian and uniform initialization, functions with high dynamics are highly favored again. Since we want to enhance the performance of our units further, and observed in preliminary experiments the accuracy rise with decreasing learning rate of the fully connected layer, we will investigate this effect in the next experiment.

4.3 Impact of Learning Rate Adjustment In Fully Connected Layer

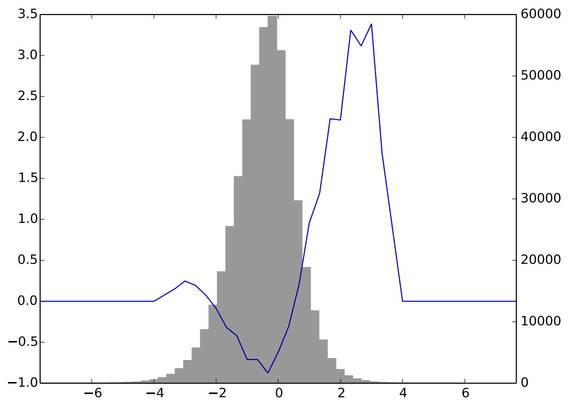
As already mentioned, in preliminary experiments we made the observation that our networks accuracy increases as the learning rate used to update the weights in the third ALU layer (layer 8) decreases. In order to investigate this effect more closely, and to improve our overall accuracy, we train the following groups of networks: A group of 20 networks with ALU units, where the units in the last, fully connected layer (layer 8), are trained with a learning rate decreased by the factor 10, compared to the remaining units. Therefore, since we already have decreased the learning rate of the network by a factor of 10 for ALU units, compared to the overall learning rate of the network, the units in layer 8 train with a learning rate of $\frac{\eta}{100}$. Furthermore, we train a second group of 20 networks, where layer 8 is trained with a local learning rate of $\frac{\eta}{1000}$ as well as a third group of 20 networks where we replace the ALUs in layer 8 by regular ReLUs. As in all our experiments performed using the ALeNet architecture (see Table 4.1), we set the base learning rate $\eta = 0.01$.



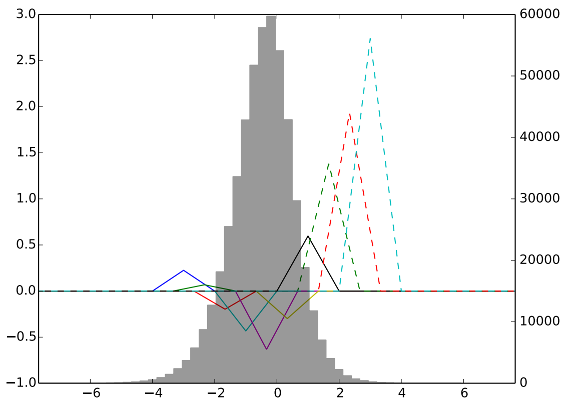
(a) Trained function in layer 2 (ReLU-like init.).



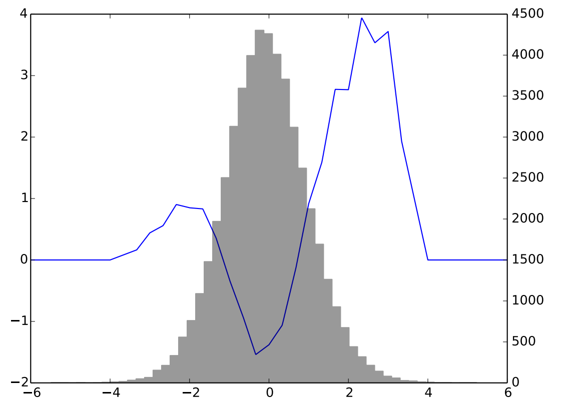
(b) Base functions in layer 2 (ReLU-like init.).



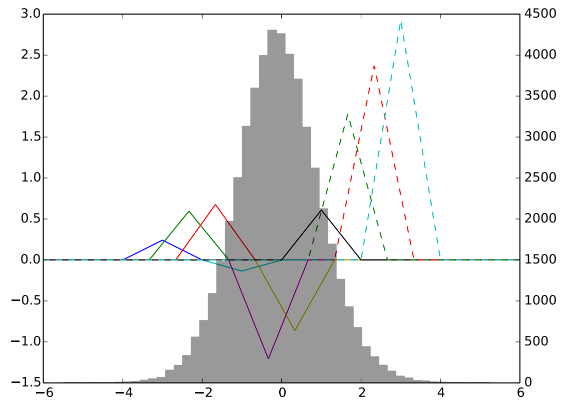
(c) Trained function in layer 5 (ReLU-like init.).



(d) Base functions in layer 5 (ReLU-like init.).

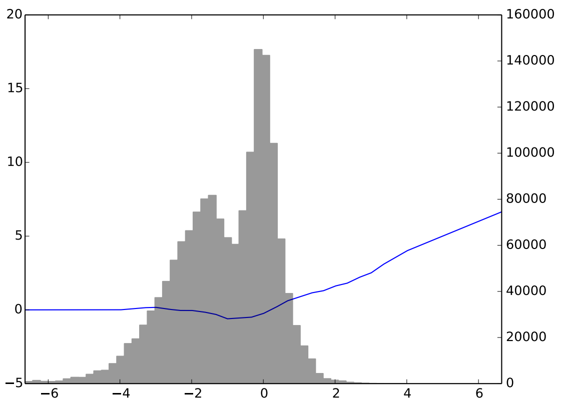


(e) Trained function in layer 8 (ReLU-like init.).

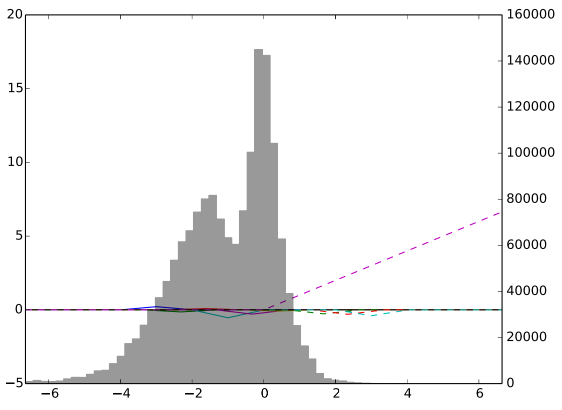


(f) Base functions in layer 8 (ReLU-like init.).

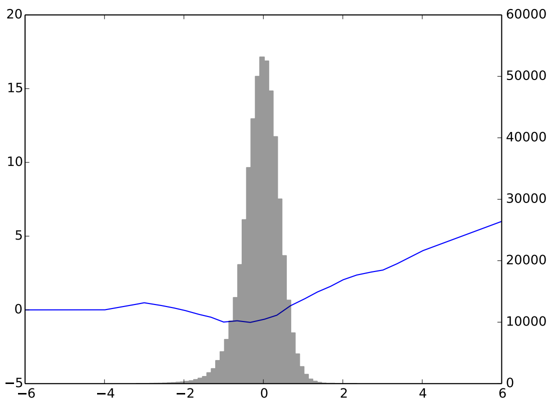
Figure 4.10: Trained activation functions of ALU layers for a sample network with ReLU-like amplitude initialization. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.



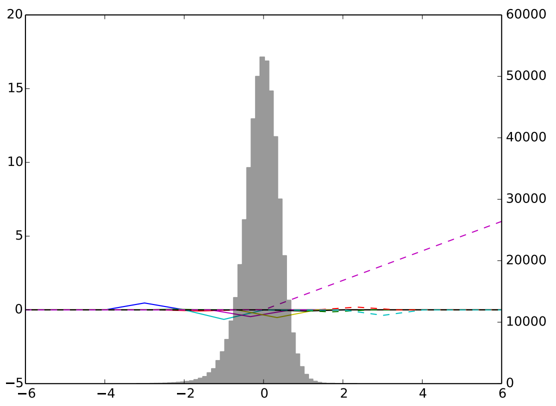
(a) Trained function in layer 2 (uniform+ ReLU init.).



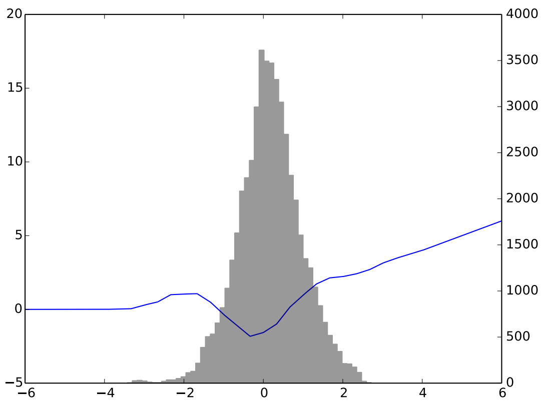
(b) Base functions in layer 2 (uniform+ ReLU init.).



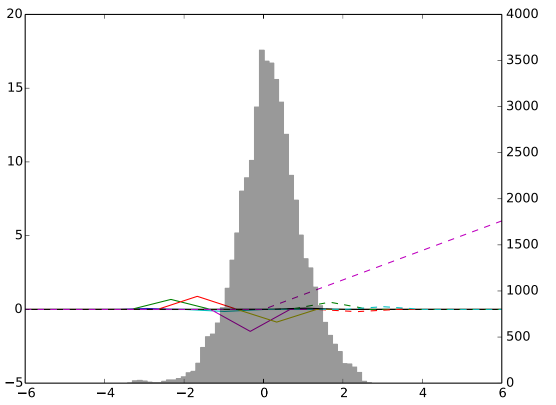
(c) Trained function in layer 5 (uniform+ ReLU init.).



(d) Base functions in layer 5 (uniform+ ReLU init.).

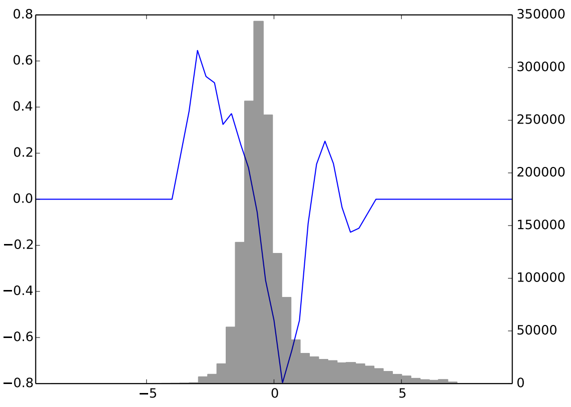


(e) Trained function in layer 8 (uniform+ ReLU init.).

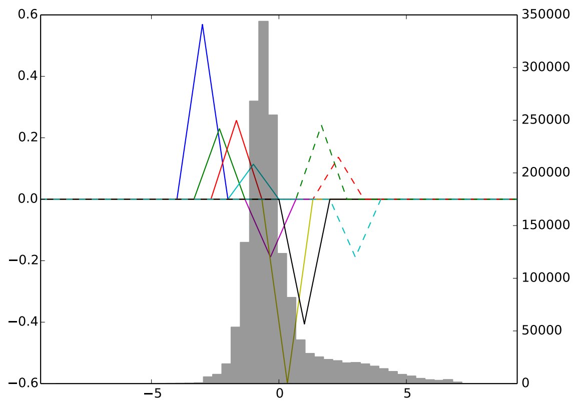


(f) Base functions in layer 8 (uniform+ ReLU init.).

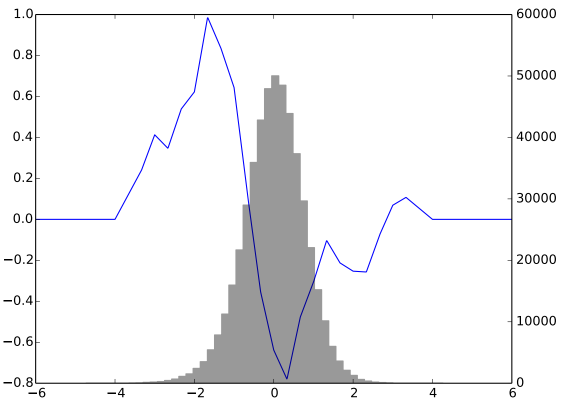
Figure 4.11: Trained activation functions of ALU layers for a sample network with a uniform amplitude initialization and an added ReLU function. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.



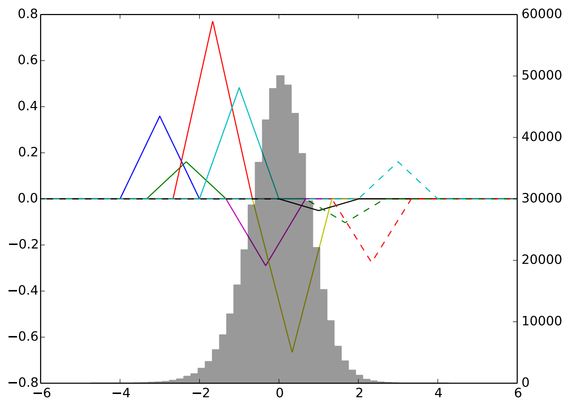
(a) Trained function in layer 2 (Gaussian init.).



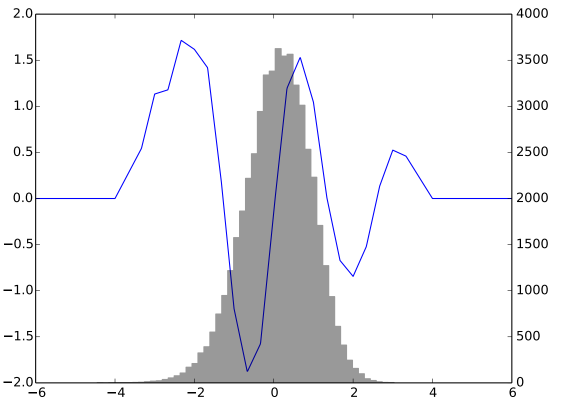
(b) Base functions in layer 2 (Gaussian init.).



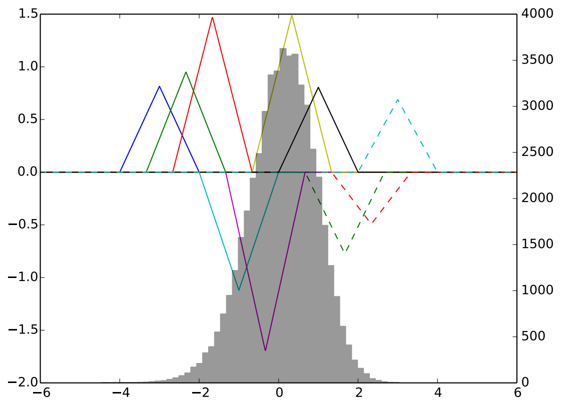
(c) Trained function in layer 5 (Gaussian init.).



(d) Base functions in layer 5 (Gaussian init.).



(e) Trained function in layer 8 (Gaussian init.).



(f) Base functions in layer 8 (Gaussian init.).

Figure 4.12: Trained activation functions of ALU layers for a sample network with Gaussian amplitude initialization. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

Table 4.5 compares mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups with ALUs to the respective values of our baseline. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional and fully connected layers.

We can see that the best accuracy is achieved by using a "mixed" ALU network with ReLU units in layer 8. The network groups with learning rates of $\frac{\eta}{100}$ and $\frac{\eta}{1000}$ show slightly worse results, which are nevertheless about one percent better than $\frac{\eta}{10}$ (which is the regular learning rate we previously used for ALUs) as well as baseline accuracy.

Interestingly the networks seem the better, the lower the learning rate is set in the fully connected layer. A possible explanation for this phenomenon might be, that small changes in the fully connected layer 8 affect all layers "below" (meaning in this case, closer to the input layer) of it. Therefore whenever a parameter update is performed in layer 8, all layers beneath layer 8 have to adjust to that change in activation. While this is true for regular networks as well, we argue that a non-convex unconstrained activation function bears much larger possible gradients, as well as a higher variation in gradient magnitude than this is the case in regular neural networks using fully connected layers with ReLUs or sigmoids. Thus the necessary adjustments might be more drastic for parameter updates in ALU layers close to the networks output. Consequently necessity to permanently make stronger adjustments to the network parameters might slow down the optimization process for ALU networks.

The networks which use ReLU units in layer 8 on the other hand, learn optimal activation functions for the convolutional layers, but do not have as high variations in gradient magnitude in the final fully connected layer, since a ReLUs gradient can only be 0 or 1. Therefore, they might achieve higher accuracies due to a faster optimization process, compared to the networks which use solely ALU units.

To verify our assumption, that higher learning rates in layer 8 slow down the training process, we again take a look at the loss on the training set during training as shown in Figure 4.13. There we can see that the contrary seems to be the case: The train loss decreases much quicker for the sample networks which are training layer 8 with a learning rate of $\frac{\eta}{10}$ or $\frac{\eta}{100}$ than the network with the slowest learning rate. The network applying ReLUs seems to lie between them in terms of convergence. A possible explanation for this observations is combination of two different effects: First, our regularization of the network might be insufficient, and the networks with the higher learning rates might minimize the loss function more quickly by changing the overall magnitude of the amplitudes, thus amplifying or attenuating the input signal. This is not desirable since we want to learn meaningful combination weights for our base functions, not simply increase or decrease them.

Furthermore, the effect of lower layers having to adjust to the changes in the fully connected ALU layer may take effect for the $\frac{\eta}{1000}$ sample network accounting for the irregularities seen in the corresponding plot at 10000 iterations. The network using ReLUs in layer 8 though does not suffer from the problems above since it does not have any optimizeable parameters in the final layer. This results in a smooth yet rapidly decreasing train loss as shown.

When we compare the activation functions trained by the sample networks in layer 8 as shown in Figure 4.14, we see that, as expected, the similarity to a ReLU increases. This is mostly due to the fact that the activations are trained slower, for the network training with $\frac{\eta}{1000}$, the activation function hardly differs from the ReLU-like initialization seen in Figure 4.7. The L2-norm of the activation function seems to decrease with the learning rate. This might indicate that the network increased the magnitude of the amplitudes in order to decrease the training loss, which is also indicated by the previously shown loss curves.

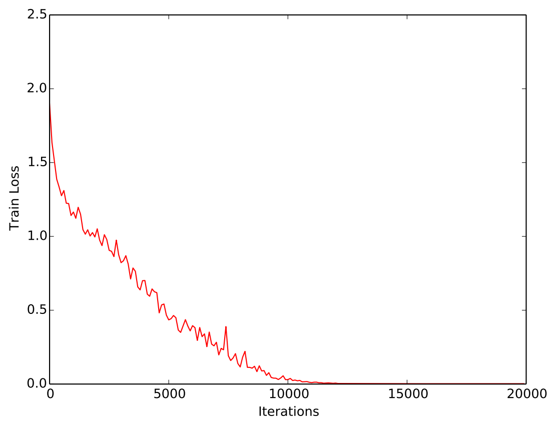
For the remaining layers, the decreasing of the activation function seems to have the opposite effect on the L2-norm of their trained activation functions: It seems to increase as the as the learning rate in layer 8 decreases, as can be seen in Figure 4.15 where we compare the activation functions trained in layer 2 for the sample networks using ReLUs as well as $\frac{\eta}{10}$ and $\frac{\eta}{1000}$ learning rates.

All in all, decreasing the learning rate in the last activation layer or even replacing the layer with a ReLU shows improvements in accuracies. The reasons therefore are suspected to be larger changes in activation functions in higher layers slowing down the training process, as well as insufficient regularization leading to a degenerated optimization process. To rule out a regularization issue, we will enforce strong regularization on our networks and investigate its impact in the next experiment.

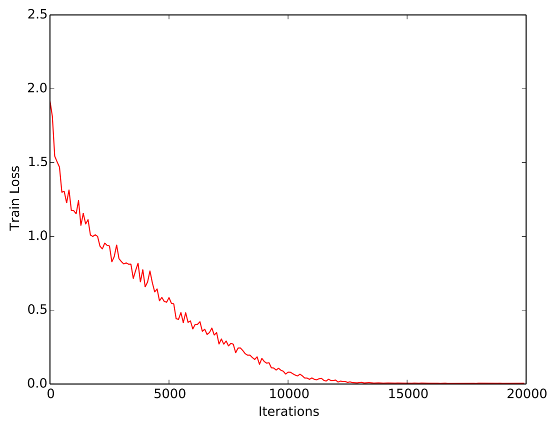
4.4 Impact Of Strong Regularization

We observe, that our networks train significantly better if the learning rate in the last layer is reduced or the layer is even replaced with a ReLU layer. We propose two different causes leading to this observation: On the one hand we suspect the network to be not sufficiently regularized, leading to the last layer minimizing the networks loss function by changing the magnitude of the weights. On the other hand we assume that high variation in gradient magnitudes in high layers (in this case, layers closer to the output layer), slow down the networks training process. Therefore, in this experiment we enforce strong regularization on the amplitudes of our ALU units to confirm or disprove the claims made above.

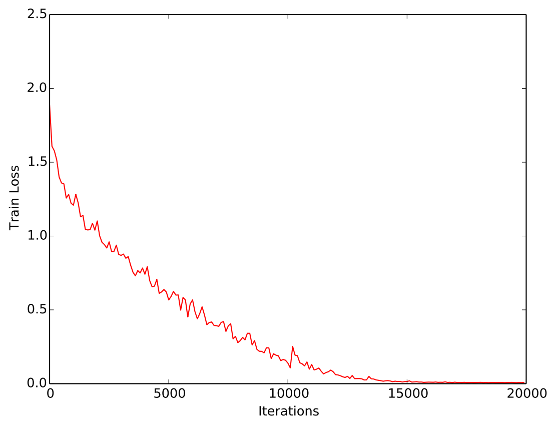
We train two groups of 20 networks, which make use of ALUs in all layers. Furthermore, we train 2 additional groups of 20 networks where we use ALUs solely for the convolutional layers 3 and 5 and place ReLUs in layer 8. We will enforce L1 regularization on one network group using solely ALUs as well as one network group using ReLUs in layer 8. In the same way, we enforce L2 regularization on the two remaining network groups. For parameters, we use the best as determined in previous experiments.



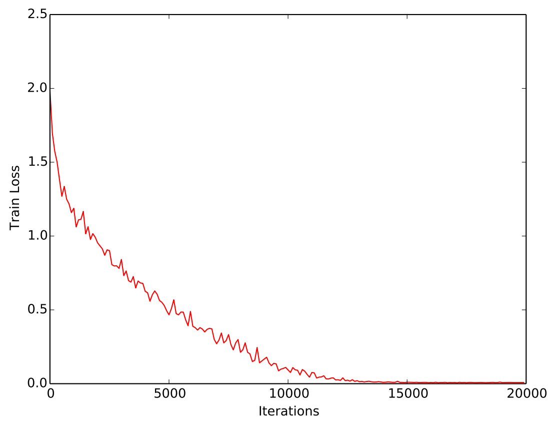
(a) Train loss, $\frac{\eta}{10}$ in layer 8.



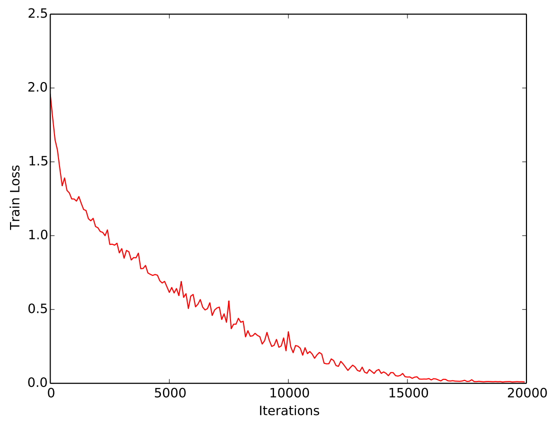
(b) Train loss, $\frac{\eta}{100}$ in layer 8.



(c) Train loss, $\frac{\eta}{1000}$ in layer 8.

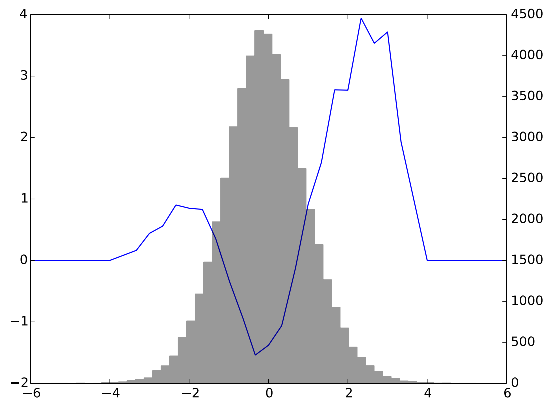


(d) Train loss, layer 8 with ReLUs

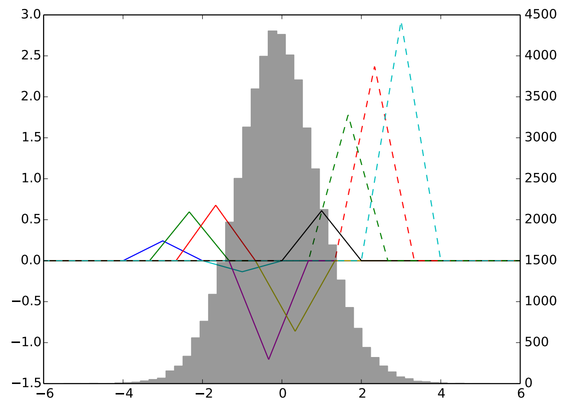


(e) Train loss, Baseline

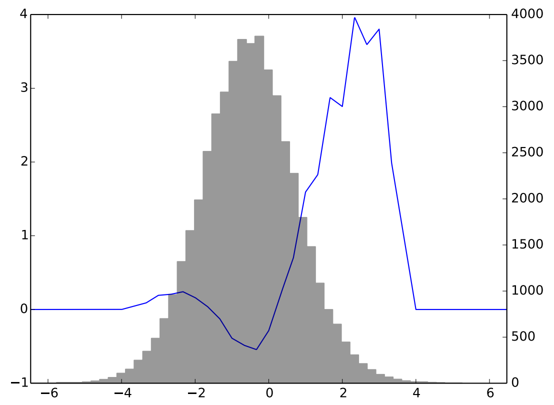
Figure 4.13: Multinomial logistic loss over 20000 iterations training five sample ALeNet networks using ALUs, as well as a ReLU-like baseline. One ALU Network trained as in the previous experiments, two ALU networks are trained with a reduced learning rate in layer 8 and for one ALU network, layer 8 has been replaced with a ReLU layer



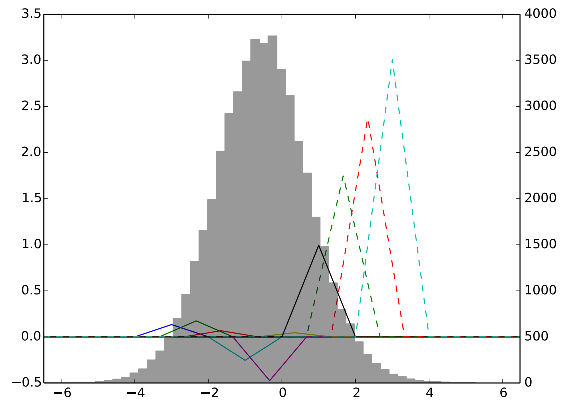
(a) Trained function in layer 8 ($\frac{\eta}{10}$ in layer 8).



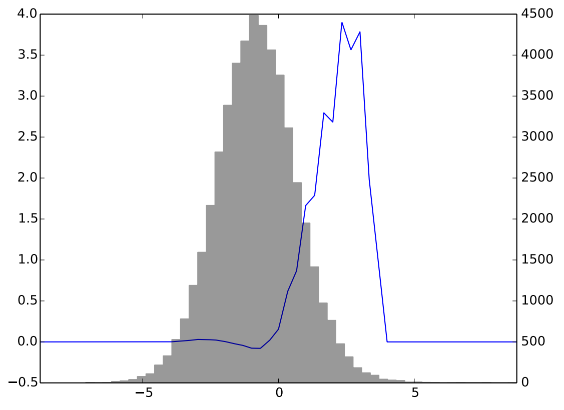
(b) Base functions in layer 8 ($\frac{\eta}{10}$ in layer 8).



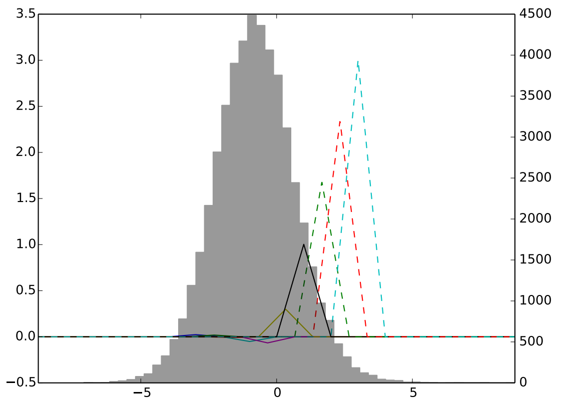
(c) Trained function in layer 8 ($\frac{\eta}{100}$ in layer 8).



(d) Base functions in layer 8 ($\frac{\eta}{100}$ in layer 8).

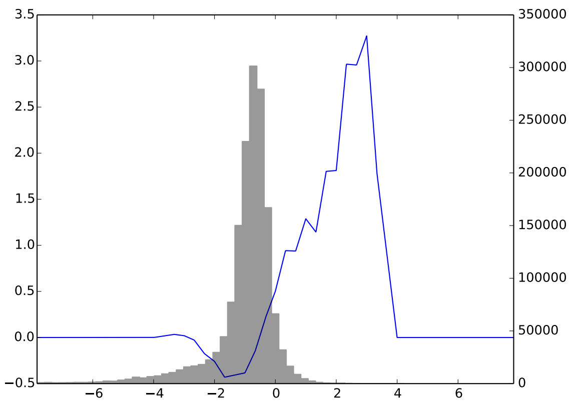


(e) Trained function in layer 8 ($\frac{\eta}{1000}$ in layer 8).

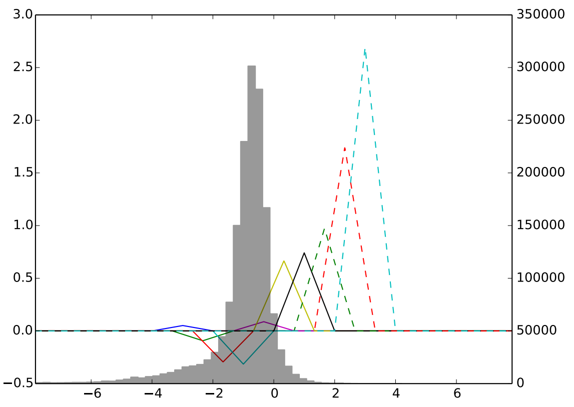


(f) Base functions in layer 8 ($\frac{\eta}{1000}$ in layer 8).

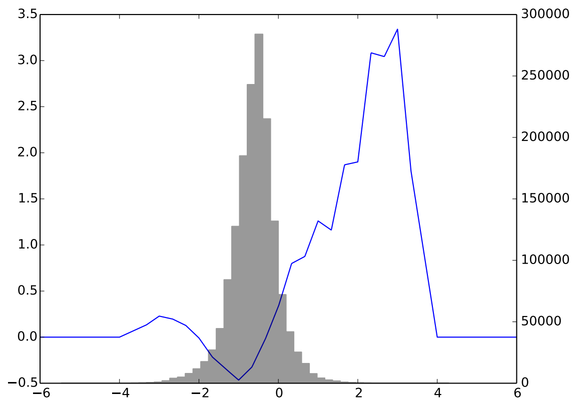
Figure 4.14: Comparison of layer 8 activation functions of ALU layers for sample networks with different learning rates for layer 8. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.



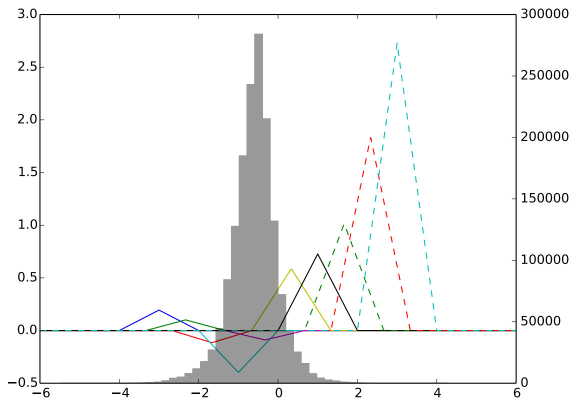
(a) Trained function in layer 2 ($\frac{\eta}{10}$ in layer 8).



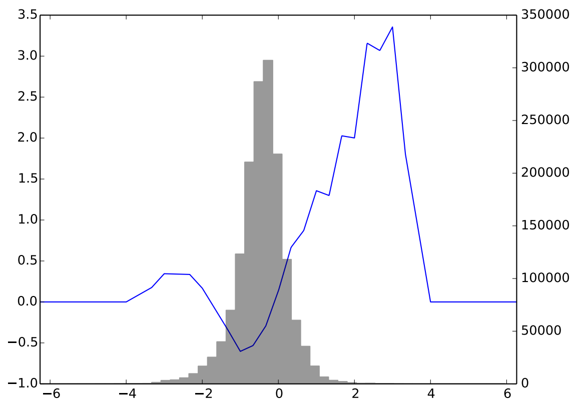
(b) Base functions in layer 2 ($\frac{\eta}{10}$ in layer 8).



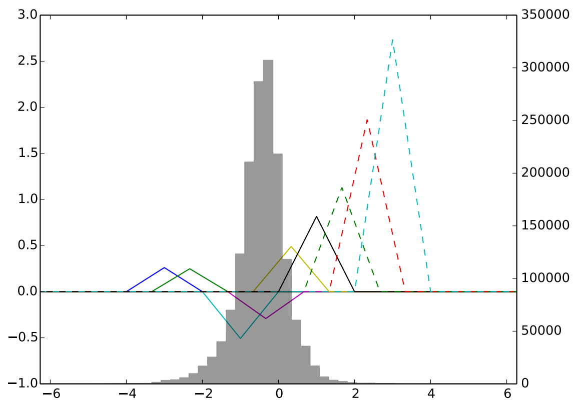
(c) Trained function in layer 2 ($\frac{\eta}{1000}$ in layer 8).



(d) Base functions in layer 2 ($\frac{\eta}{1000}$ in layer 8).



(e) Trained function in layer 2 (ReLUs in layer 8).



(f) Base functions in layer 2 (ReLUs in layer 8).

Figure 4.15: Comparison of layer 8 activation functions of ALU layers for sample networks with different learning rates for layer 8. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

layer 8	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
L1 reg.	71.89 ± 0.64	0.8212 ± 0.0173	0.4958 ± 0.0252
L2 reg.	70.80 ± 0.90	1.0088 ± 0.0303	0.0226 ± 0.0015
L1 reg. w. ReLU	71.74 ± 0.93	0.8532 ± 0.0274	0.3129 ± 0.0218
L2 reg. w. ReLU	71.53 ± 1.13	0.9969 ± 0.0358	0.0633 ± 0.0058

Table 4.6: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test sets of the CIFAR-10 data set for our baseline network as well as the networks using ALU layers with enforced L1 or L2 regularization. Triangles have been used as base functions, two network groups have been trained with ALU units exclusively, two groups use ReLUs in layer 8. Results are averaged over 20 networks each.

We enforce L1 regularization on our network groups by normalizing the amplitudes of the units (to be more precise, amplitudes of the layers, since the units are connected) after every step such that

$$\sum^{\forall i} |k_i| = 1, \tag{4.4}$$

where k_i is again the amplitude of the base function with index i. Similar, we enforce the L2 norm on our remaining two network groups by normalizing the amplitudes of our ALU layers such that

$$\sqrt{\sum^{\forall i} k_i^2} = 1. \tag{4.5}$$

Table 4.6 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups as well as the baseline group. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional and fully connected layers.

First we see that no network group managed to outperform the group of unregularized mixed networks containing ALU and ReLU units. For the regularized networks, the network group using solely ALUs with L1 regularization shows the best performance getting rather close to the group of unregularized mixed networks (see Table 4.5), followed by the network group of mixed networks with L1 regularization. L2 regularization shows slightly worse performance, furthermore here the group of "pure" networks, which use only ALUs, shows worse performance than the group of "mixed" networks with ALUs and ReLUs.

In general it can be said that L1 regularization of the amplitudes works better than L2 regularization, but it remains in question, why the ReLUs in the output stage worsen the classifier accuracy for L1 regularization and enhance it for L2 regularization. We assume that L2 regularization is less suitable than L1 regularization for our units, and thus ReLU stages pose an additional enhancement, since they are unable to overfit the training data. For the L1-regularized networks on the other hand, the ReLU stage might be less beneficial since it does not reduce overfitting. A possible explanation for the slightly worse performance of the mixed model compared to the pure model might be founded in the ALU layers weights being reprojected after gradient update. This reprojection results in a slight fluctuation of the layers output values. This of course results in slightly different input values for following layers as well. If the activation functions of the following layers are ALUs, which are continuous, the changes might be minimal, but for ReLUs, who bear a discontinuity at 0, larger changes could result, which were not foreseen by the backpropagation algorithm, since they occurred after the gradient update step. This might also explain the higher standard deviation both for L1 and L2 regularizations.

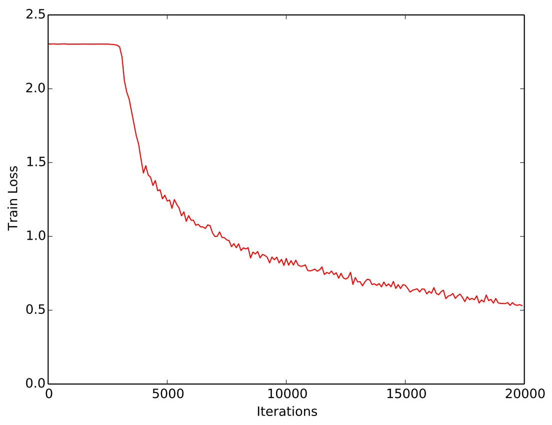
A further interesting observation is, that the average train loss of the L1 regularized network group is significantly higher than of the remaining networks, and is the only network group we have seen so far, for which the train loss of the data set approximately matches the test loss on ALeNet (see Table 4.1). This indicates that constraining the L1 norm of the amplitude the way we did, is an effective way to prevent overfitting in our networks. To investigate the train loss further, we take a look at the train loss over time as seen in Figure 4.16.

We first notice that with normalized amplitudes, the train loss does not tend to decrease as rapidly as in previous experiments. This is likely due to the weight normalization since the loss cannot be minimized anymore by changing the magnitude of the layers amplitudes, which is bound to the L1/L2 norm and hence fixed now. Another observation standing out is, that for the L1 regularized network, the train loss hardly decreases in the beginning, the learning process is delayed. This is most likely due to the normalization of the weights: since the initialization function is a ReLU, its steepness will be decreased once the amplitudes of the layer have been normalized. Therefore, its gradients will be smaller as well, permitting less gradient flow through the unit slowing down the learning process. Nevertheless, after about 4000 iterations, the network starts learning and the loss decreases rapidly, but saturates earlier than for the remaining networks, which is consistent with the higher train loss for L1 regularized networks as seen in 4.6.

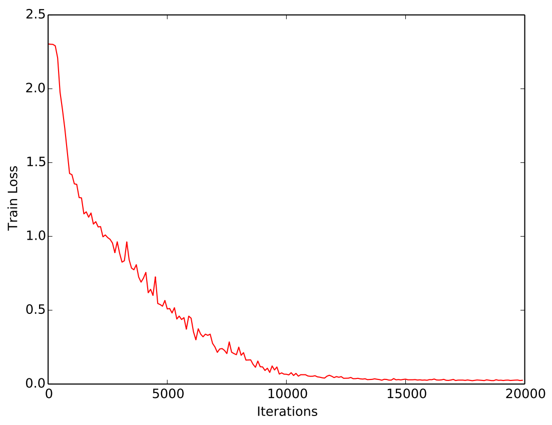
For the L2 network, we can see that their loss decreases more gently than for the unregularized networks. Furthermore, the steep, slightly delayed descent of the training loss for the L2 regularized networks might indicate a similar effect of delayed training as already discussed for the L1 regularized networks, only in a much more subtle fashion, which could be responsible for the increased standard deviation of the networks trained with strong regularization.

We take a look at the activation functions trained by the sample networks: In Figure 4.18 we can see that its trained activation functions look less ReLU like and more symmetric than for unregularized networks from the previous experiments. In fact, especially the activation function for the pure L1 regularized network seems to have a clear symmetrical form, but also the the mixed L1 regularized network as well as the pure L2 regularized network are much more symmetric. Apart from that fact, the trained activation functions of all networks look similar.

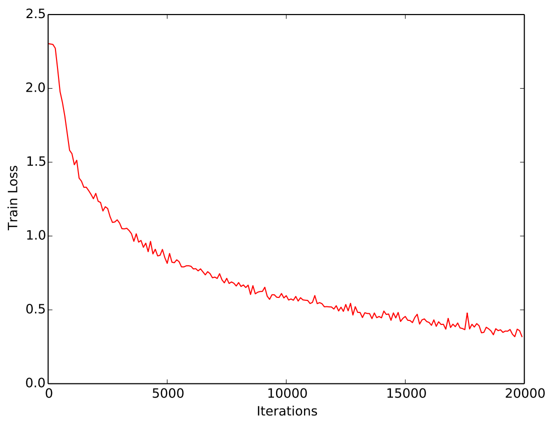
Finally, since the replacement of layer 8 improved our classifier accuracy in the previous experiment, we compare the activation functions in layer 8 for the L1/L2 regularized



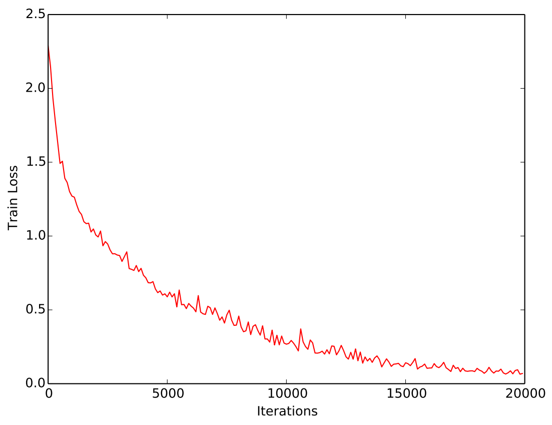
(a) L1 regularized network.



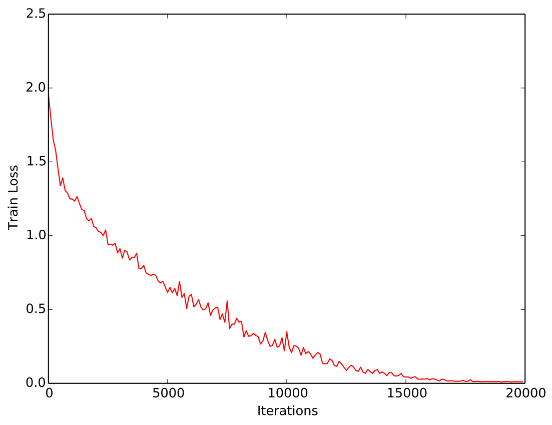
(b) L2 regularized network.



(c) L1 regularized network (ReLU in layer 8).

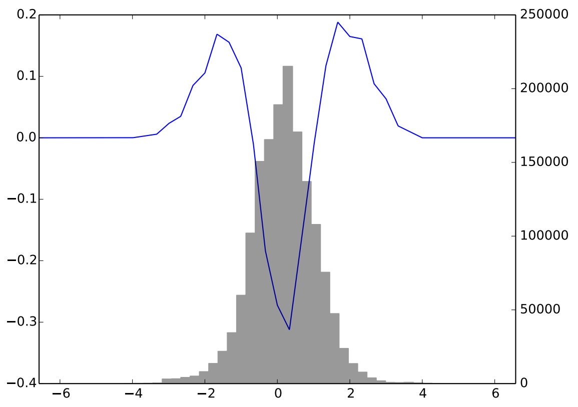


(d) L2 regularized network (ReLU in layer8).

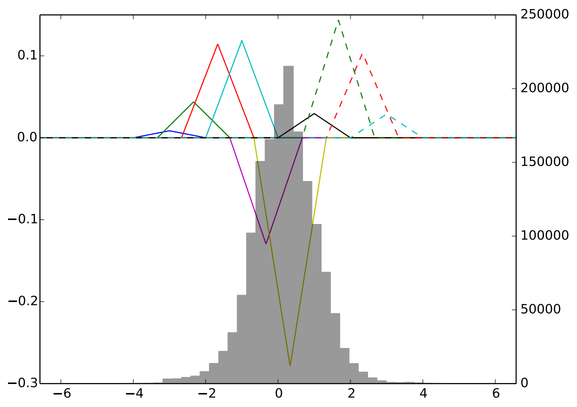


(e) Train loss, Baseline.

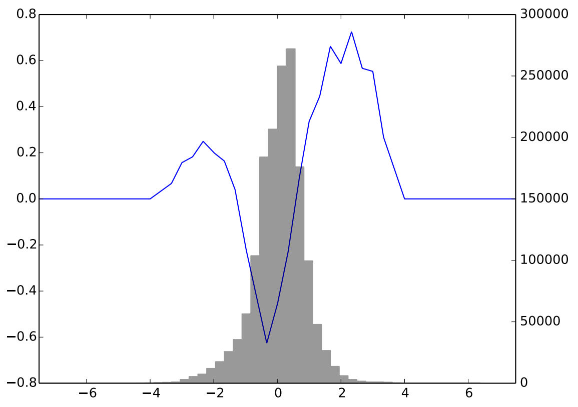
Figure 4.16: Multinomial logistic loss over 20000 iterations training four sample ALeNet networks with strong L1 and L2 regularizations. For two sample networks, the final activation layer has been replaced with a ReLU layer.



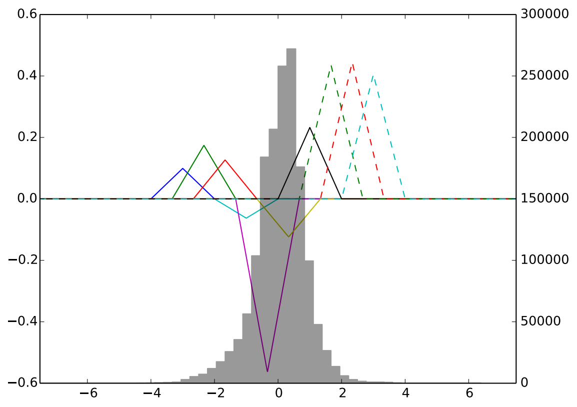
(a) Trained function in layer 2 (L1 regularized).



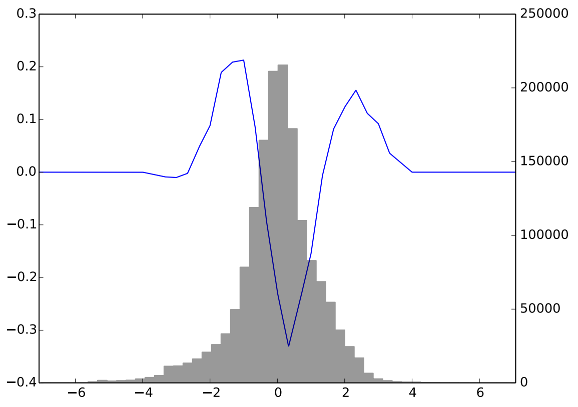
(b) Base functions in layer 2 (L1 regularized).



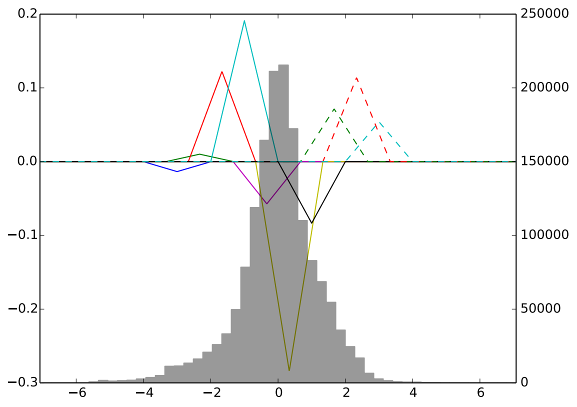
(c) Trained function in layer 2 (L2 regularized).



(d) Base functions in layer 2 (L2 regularized).



(e) Trained function in layer 2 (L1 regularized, ReLU stage).



(f) Base functions in layer 2(L1 regularized, ReLU stage).

Figure 4.17: Trained activation functions of ALU layers with L1/L2 regularization for a sample networks with and without final ReLU activation layer. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

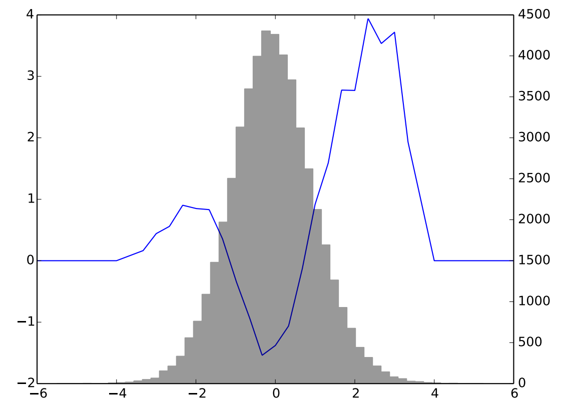
normalization	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
I = 5	70.92 ± 0.85	1.2258 ± 0.0416	0.0097 ± 0.0017
I = 10	72.03 ± 0.43	1.1731 ± 0.0216	0.0073 ± 0.0005
I = 15	70.75 ± 1.11	1.2735 ± 0.0665	0.0066 ± 0.0014
I = 20	65.69 ± 3.89	1.3567 ± 0.1741	0.2757 ± 0.2971
I = 50	14.13 ± 12.52	2.2182 ± 0.2564	2.1751 ± 0.3849

Table 4.7: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test sets of the CIFAR-10 data set for our baseline network as well as the networks using ALU layers with a different number of triangle base functions I. Results are averaged over 20 networks each.

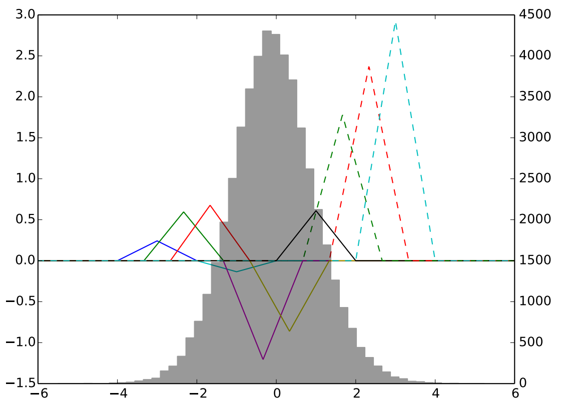
sample networks directly to the layer trained with an unregularized network. While all functions have a very similar shape, the regularized functions seem to be more symmetric and made up of less base functions than the unregularized one. Due to the assumable more directed optimization process, layer input space seems to be less spread over the input domain and tied closer together compared to the unregularized networks, where the histograms of the unnormalized sample networks are broader.

All in all we have achieved almost the same performance in pure ALU networks as we achieved before in mixed ALU networks by applying L1 regularization. Pure networks with enforced L2 regularization brought improvement over unregularized pure networks as well, but performed worse than L1 regularized networks. For L1 regularized networks, replacing ALUs with a ReLUs in layer 8 decreased performance, while performance was slightly improved for the insufficiently regularized networks enforcing L2 regularization.

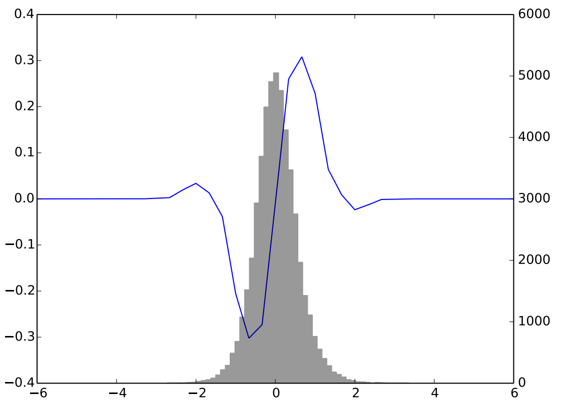
From these observations we derive the following conclusions: Our pure ALU networks are, as suspected insufficiently regularized in the fully connected layers. Replacing the fully connected layers with ReLU layers is a possible solution to improve the networks performance. Alternatively the network can be regularized by enforcing the L1 norm of the ALU layers amplitudes to be constant, i.e. 1. A possible effect of gradients with highly varying magnitude in higher ALU layers thus slowing down the optimization process in lower ALU layers, as previously discussed, remains possible, but seems to have little impact since we only achieved a slight improvement with the unregularized mixed network group over the network group that was trained with enforced L1 regularization. Nevertheless, since pure ALU networks with enforced regularization show on average a slight decrease in accuracy on compared to simple mixed ALU/ReLU networks, we decide to conduct our further experiments with a unregularized mixed network, with ALUs behind the 2 convolutional layers and layer 8 is replaced by a ReLU layer.



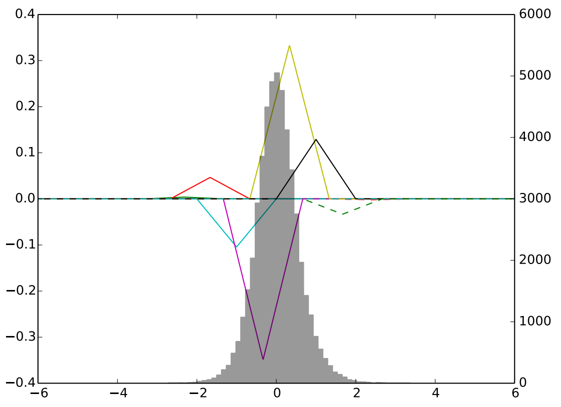
(a) Trained function in layer 8 (unregularized).



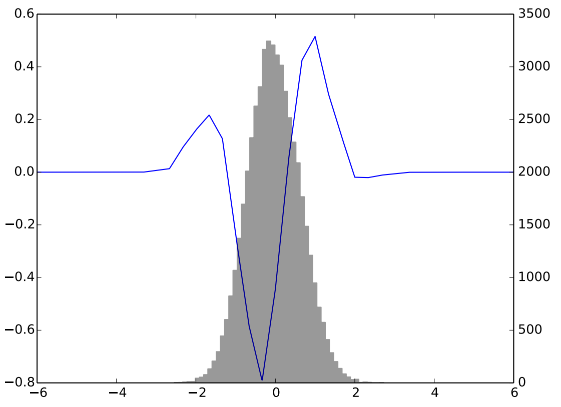
(b) Base functions in layer 8 (unregularized).



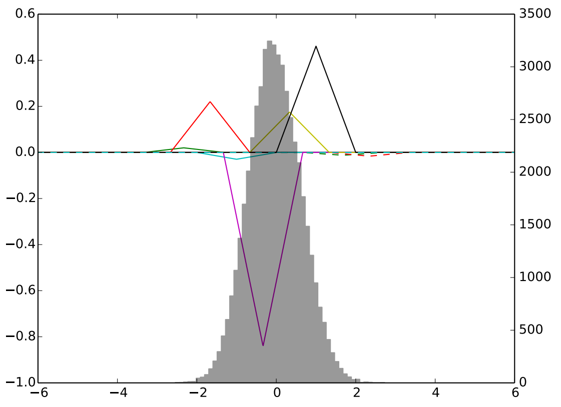
(c) Trained function in layer 8 (L1 regularized).



(d) Base functions in layer 8 (L1 regularized).



(e) Trained function in layer 8 (L2 regularized).



(f) Base functions in layer 8 (L2 regularized).

Figure 4.18: Trained activation functions of ALU layers with L1/L2 regularization for a sample networks. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

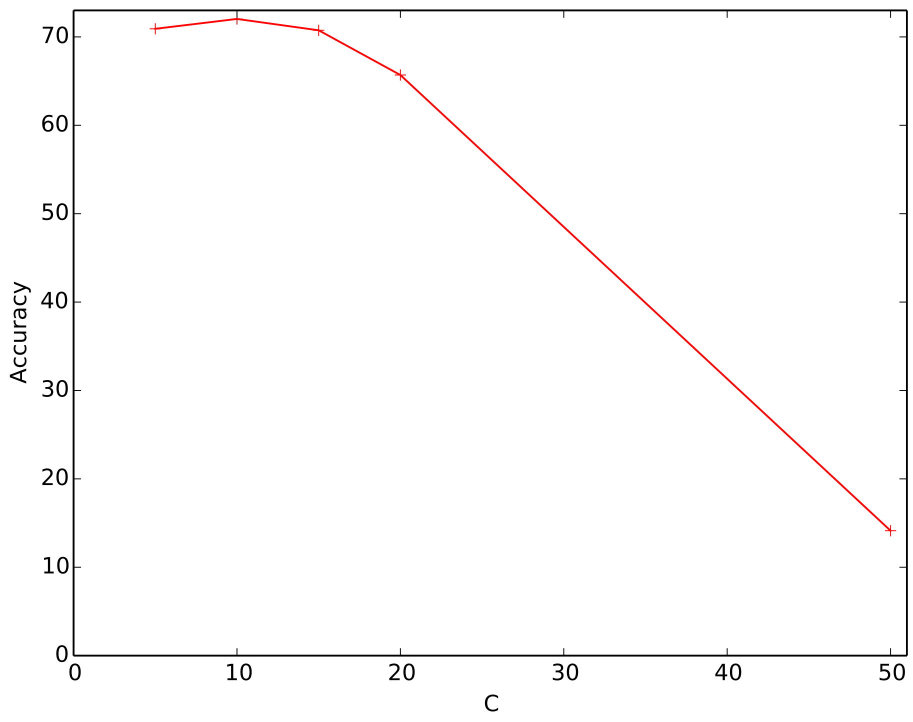


Figure 4.19: Mean test accuracy on CIFAR-10 data set as a function of the number of triangle base functions used in ALU layers. Results have been averaged over a group of ALeNet networks with 2 ALU layers after the convolutional layer as well as a ReLU layer as a fully connected layer.

4.5 Impact Of Number Of Base Functions

In this experiment we investigate how varying the number of base functions affects accuracy. We train 5 network groups of 20 networks using 5, 10, 15, 20 and 50 triangle base functions in their ALU layers respectively. Since we have obtained the best accuracy results so far by using ReLU units in layer 8, we do so again for all networks. The remaining parameters are set to the ones showing the best performance so far. We initialize our amplitudes again with a ReLU-like function.

Table 4.6 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups as well as the baseline group. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional and fully connected layers. We can see that our initial setting using $I = 10$ Gaussians actually works better than all the additional I we tried. While performances for $I = 5$ and $I = 15$ are only slightly worse, accuracy gets significantly worse for $I = 20$ and seems to fail completely for $I = 50$. We observe the same trend when we visualize our results by plotting the average accuracy vs. the number of base functions used, as seen in Figure 4.19: There is a peak at $I = 10$, the accuracy decreases for lower I and decreases further for higher numbers of I until the networks fail at $I = 50$.

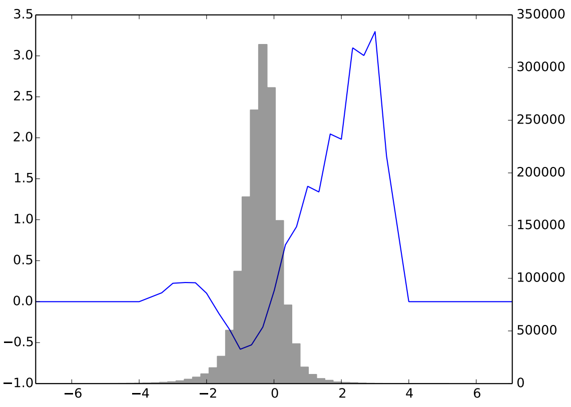
To further investigate why the networks of the group with $I = 50$ Gaussians seem to fail we take a look at the learning curves of sample networks taken from each group, shown in Figure 4.21: Sample networks with $I < 20$ train normally, the network using $I = 20$ starts to train very late, the network with $I = 50$ base functions does not seem train at all.

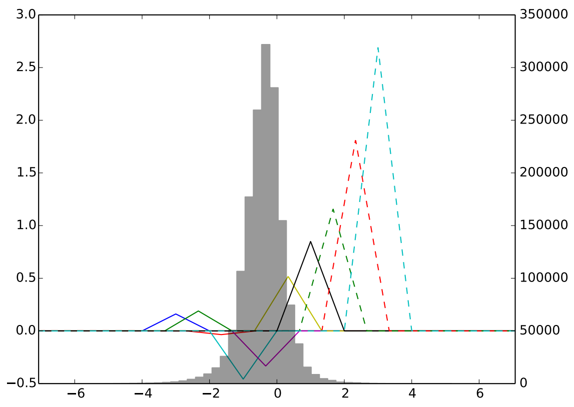
To gain deeper insight about the underlying reasons, we take a look at the trained activation functions in layer 2 (see Figure 4.27): When we compare the activation functions trained with ALUs after the first convolutional layers with 10, 20 and 50 activation functions, we see that, while the network with 10 triangle base functions trains fine, the network with 20 units seems to be less developed and as expected, the activation function using 50 base units hardly trained at all. Most likely, the reason for this behavior is that placing 50 base functions in a rather confined space ($s = 3$), yields to very high initial output values of the layers. Thus, to minimize loss, the network will steer the layers input in a way such that its output is zero for all training samples. This is the local optimal decision, since very high outputs of the layer yield very high losses.

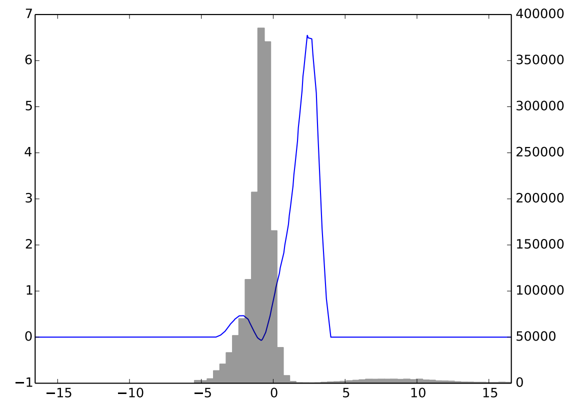
To confirm our suspicion, we plot the activation functions learned in layer 5 (see Figure 4.22) and find our suspicions confirmed: as we can see, the network steers away the layers inputs from the positions of our base function in order to set the layers output to 0.The essential problem here is, that the triangle functions we chose to train, do not adjust their shape, as for example a Gaussian would. Thus placing 50 triangles in a quite confined space, leads to very high output values in the layer, destabilizing the network. All in all, we found out that for our data and network, $I = 10$ triangle base functions seems to be a reasonable choice, and that the optimal choice of the number of base functions is as well dependent on the input input range s, and thus on the distribution of the input space of each layer. Therefore, we can see a slight drawback of our scaled initialization method here for activation functions of fixed width, since the breadth of the activation function u, the input range s as well as the number of gaussians I are always interdependent here. While a too low number of base functions will lead to an insufficient approximation of the activation function, a too high number will lead to too large initial output values in the layers and prevent a meaningful optimization process. Considering that the optimal input range is highly influent concerning the optimization process, we investigate the input range s and its effects on accuracy in the next experiment.

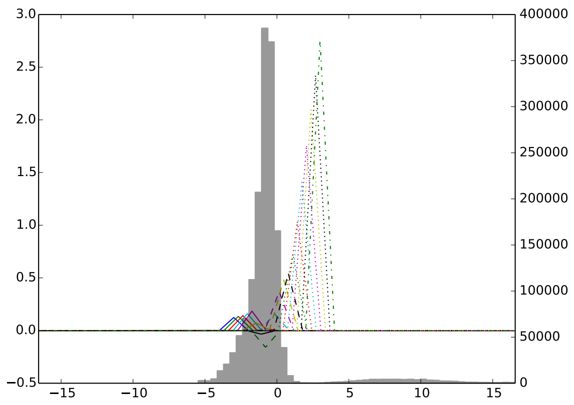
4.6 Impact Of Different Input Range

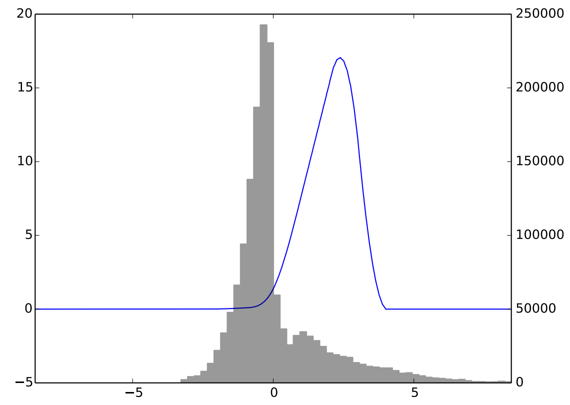
Finding the correct input range s is important for classifier accuracy, since at least when we use locally confined base functions, we are unable to learn non-zero activation functions which approximate the optimal activation function on the whole input domain. Thus we


(a) Trained function for layer 2, I=10.


(b) Base function for layer 2, I=10.


(c) Trained function for layer 2, I=20.


(d) Base function for layer 2, I=20.


(e) Trained function for layer 2, I=50.

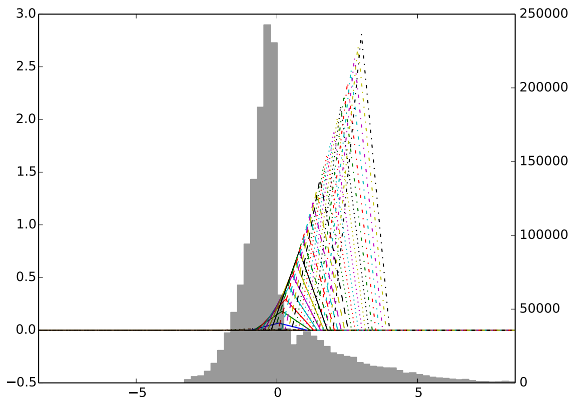
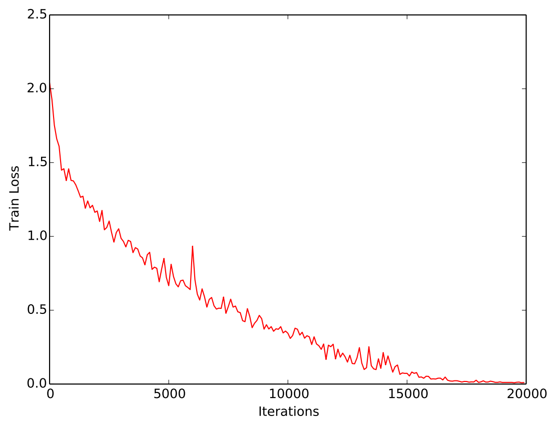
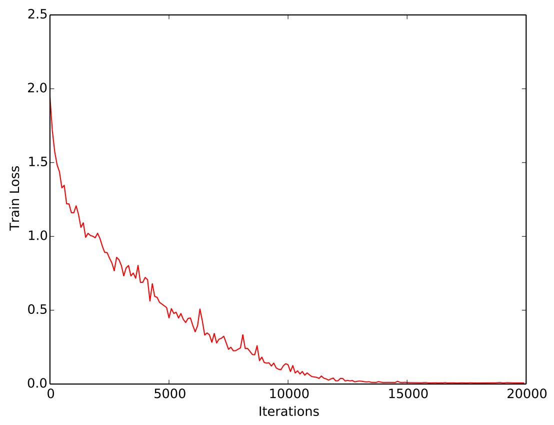

(f) Base function for layer 2, I=50.

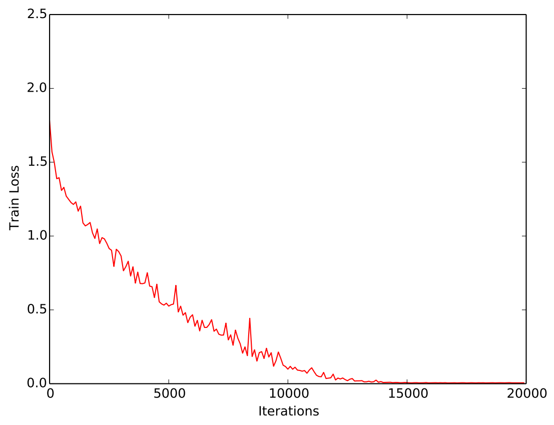
Figure 4.20: Comparison of layer 2 activation functions of ALU layers for sample networks with 10, 20 and 50 triangle base functions. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.



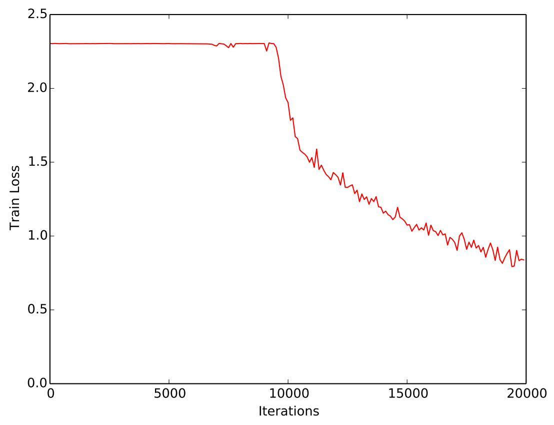
(a) I=5



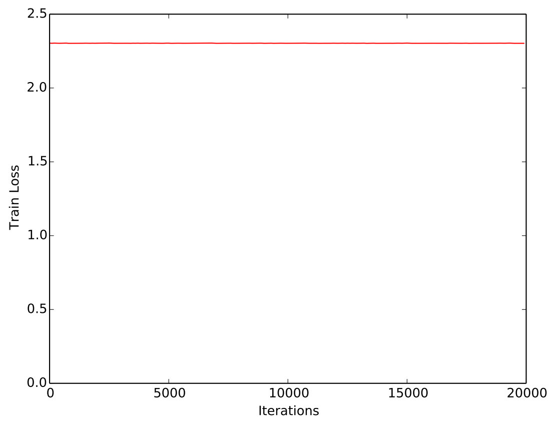
(b) I=10



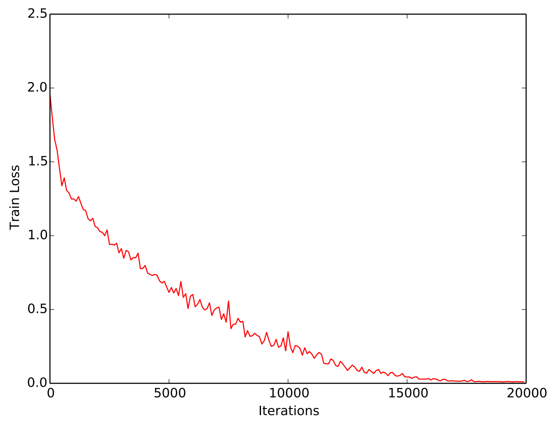
(c) I=15



(d) I=20

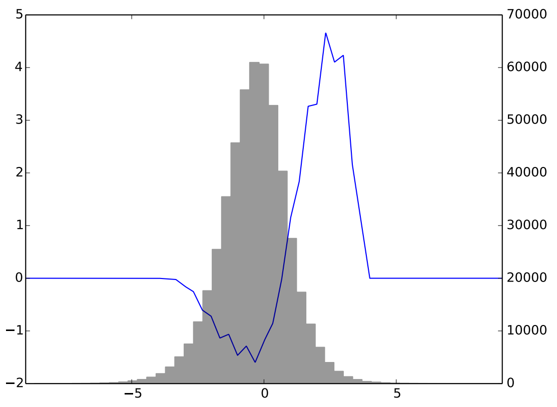


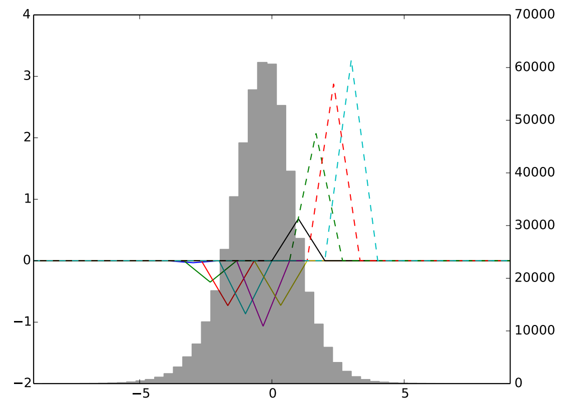
(e) I=50

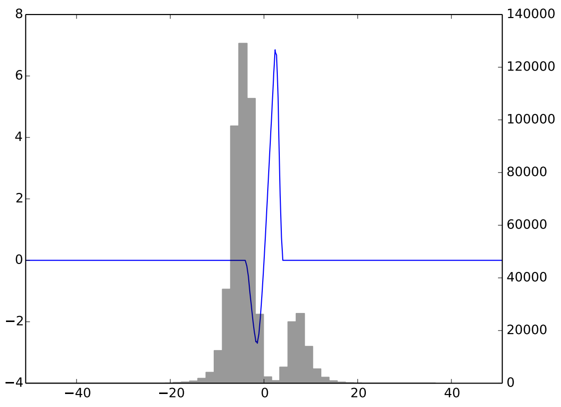


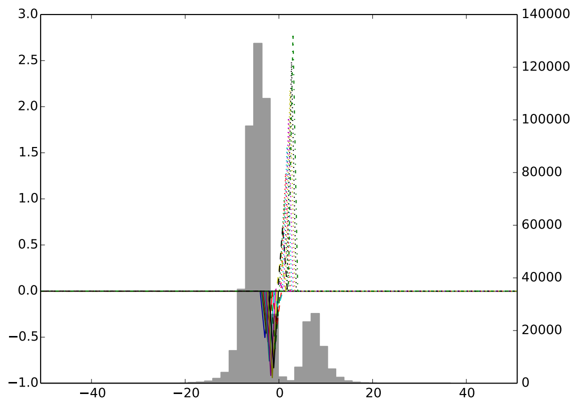
(f) Train loss, Baseline

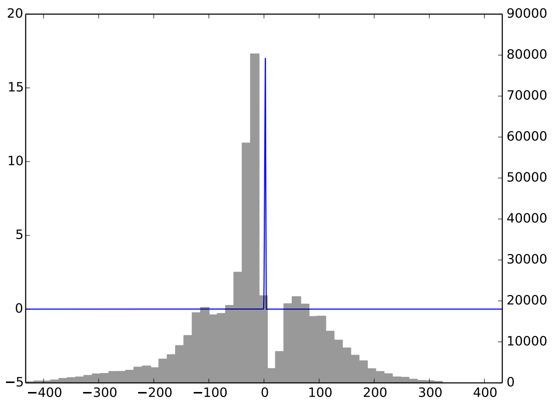
Figure 4.21: Multinomial logistic loss over 20000 iterations for four sample ALeNet networks with different numbers of base functions I.


(a) Trained function for I=10


(b) Base function for I=10


(c) Trained function for I=20


(d) Base function for I=20


(e) Trained function for I=50

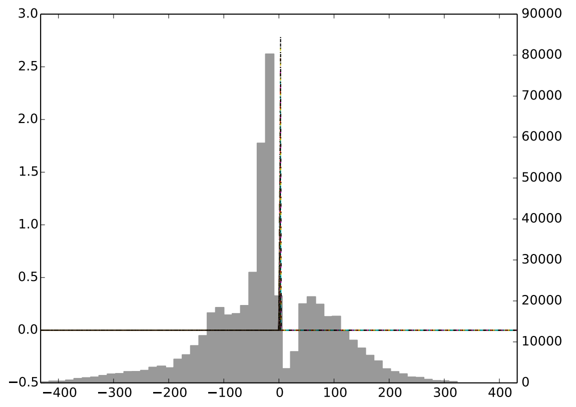

(f) Base function for I=50

Figure 4.22: Comparison of layer 2 activation functions of ALU layers for sample networks with 10, 20 and 50 triangle base functions. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

normalization	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
s = 1	69.28 ± 1.06	1.4454 ± 0.0622	0.0060 ± 0.0026
s = 3	72.21 ± 0.49	1.1639 ± 0.0196	0.0073 ± 0.0003
s = 5	72.88 ± 0.56	1.1153 ± 0.0246	0.0096 ± 0.0006
s = 7	72.08 ± 0.60	1.1780 ± 0.0277	0.0071 ± 0.0003
optimized	73.06 ± 0.73	1.1892 ± 0.0356	0.0050 ± 0.0004

Table 4.8: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test setsof the CIFAR-10 data set for our baseline network as well as the networks using ALU layers with different input ranges as well as trainable input range. Results are averaged over 20 networks each.

need to confine ourselves to a subspace where our input values most likely occur. Since this subspace does vary with different data sets and networks, as a consequence we train s jointly with the amplitudes $\mathbf{k}$ of our base functions as we proposed in Section 3.1. Therefore, in the first part of this experiment we investigate how the choosing of different input ranges impacts the training performance. We train 4 groups of 20 networks using the ALeNet architecture, where we train each group with a different input range out of $s = 1, 3, 5, 7$. Moreover, in the second part of the experiment, we train an additional group of 20 networks, where the input range s is added as a trainable parameter in the training process. We initialize s with the best observed setting obtained from training the 4 network groups mentioned earlier.

Table 4.11 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups with input ranges $s = 1, 3, 5, 7$ as well as one group with a trainable input range and the baseline group. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional and fully connected layers.

When taking a look at the trained activation functions in Figure 4.24 we find support for our previous claims: For $s = 1$, the base functions are crammed into their predefined input domain, highly overlap, and are unable to capture the whole input space of the layer. For $s = 5$, the base functions span over a domain which is sufficient to capture the whole input space, and partly overlap with their neighbors. Finally, for $s = 7$ only a small number of base functions can cover the input space, and therefore the network performs worse than before.

So in a nutshell we found out that the s is a parameter which is mostly influenced by the input distribution of the data. If s is chosen too small our ALUs will be unable to cover the whole input space. On the other hand, if s is chosen too large, an insufficient amount of base functions remain to generalize in the input space. Moreover, we trained s from $s = 5$ which was the best input range we found, and managed to improve average accuracy slightly. All in all it can be said that the additional optimization of s brings slight improvement in terms of average accuracy, which remains questionable if we take

normalization	accuracy	test loss	train loss
baseline	69.30 ± 0.62	1.5530 ± 0.0516	0.0092 ± 0.0007
$u_{opt} = 0$	72.10 ± 0.62	1.1716 ± 0.0215	0.0074 ± 0.0003
$u_{opt} = 0.1$	71.44 ± 0.40	1.2133 ± 0.0215	0.0068 ± 0.0003
$u_{opt} = 0.3$	70.19 ± 0.52	1.3247 ± 0.0292	0.0061 ± 0.0004
$u_{opt} = 0.5$	67.11 ± 2.21	1.4987 ± 0.0509	0.0490 ± 0.0995
$u_{opt} = 1$	30.95 ± 21.61	1.8407 ± 0.5002	1.7517 ± 0.6114
$u_{opt} = 3$	10.03 ± 0.09	2.3025 ± 0.0013	2.3020 ± 0.0015
u optimized	71.93 ± 0.47	1.3337 ± 0.0296	0.0034 ± 0.0002
u and s optimized	72.24 ± 0.69	1.3205 ± 0.0369	0.0032 ± 0.0001

Table 4.9: Mean and standard deviation for accuracy scores and multinomial logistic loss on training and test sets of the CIFAR-10 data set for our baseline network as well as the networks using ALU layers with Gaussian base functions with different shape parameters as well as trainable shape parameter and jointly trainable shape parameter and input range. Results are averaged over 20 networks each.

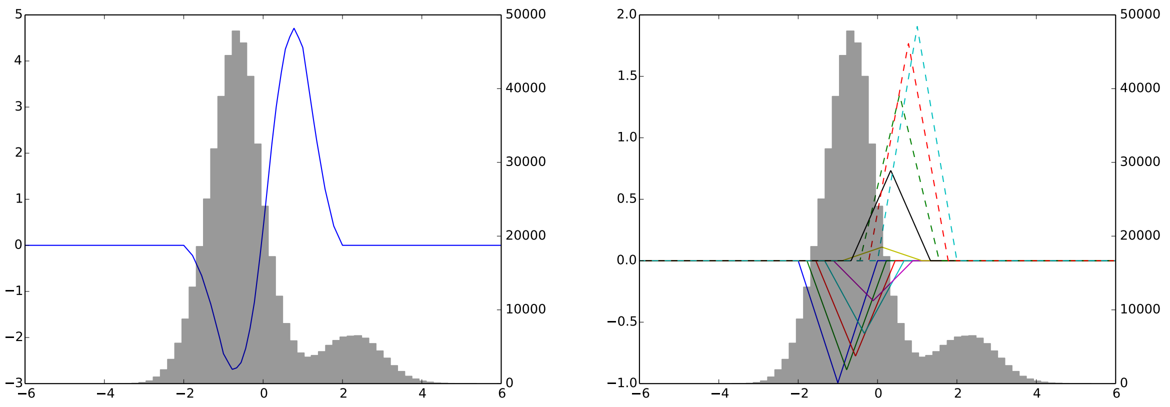
into account the complex weight update equation as derived in Section 3.1 and longer training times.

4.7 Impact Of Different Shape Parameters

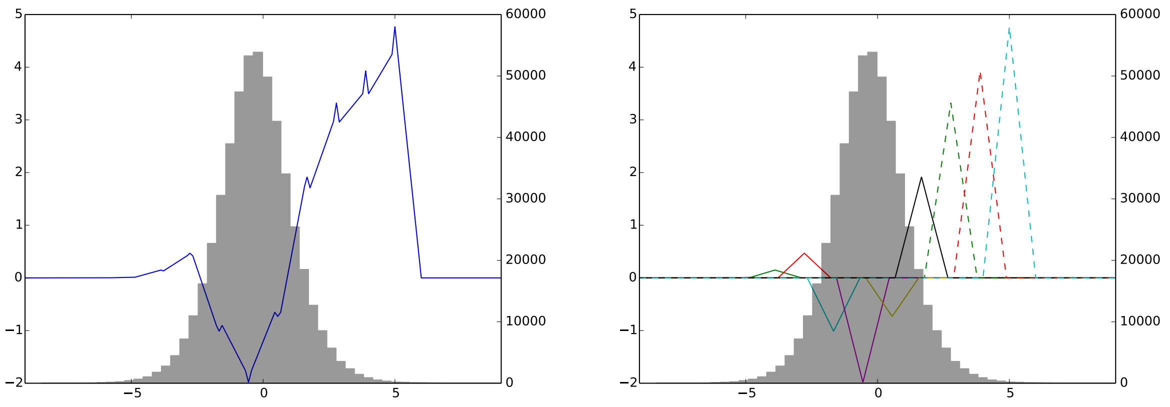
In this experiment, we will investigate the impact of optimizing the shape parameter of base functions as proposed in Section 3.1. Since Gaussian base functions are the only ones we proposed, which utilize the shape parameter, we conduct the following experiment using Gaussian base functions. We train network groups of 20 networks with ALUs using a shape bias of $u = 0.1$, 0.3, 0.5, 1 and 3 respectively. Thereafter, we initialize u with the best result obtained and train 2 more groups of 20 networks, where we optimize amplitudes $\mathbf{k}$ and shape u in one group, and amplitudes $\mathbf{k}$, shape u and input range s in the other group. We train the CIFAR-10 data set on ALeNet networks (see Table 4.1) with two layers of ALU units after the convolutional layers as well as ReLUs after the fully connected layer. For parameters we use the ones we observed to result in highest accuracy values so far.

Table 4.11 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups with shape bias $u = 0.1$, 0.3, 0.5, 1 and 3, as well as two groups with a trainable shape and input range parameters and the baseline group. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional and fully connected layers. It shows that optimization with static u works best without any additional bias: Accuracy decreases with increasing shape bias.

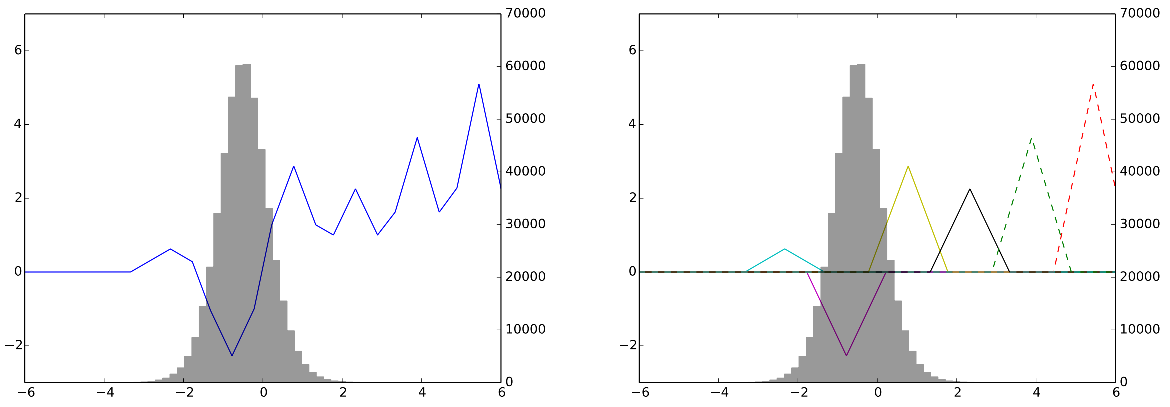
The possible reason for decreasing accuracy with increasing u might be that the u dependent from s is already rather high, since $s = 5$ is already chosen rather large in relation to the average input space as we saw it in the previous experiments. If the base



(a) Trained function in layer 5 for $s = 1$. **(b)** Base function in layer 5 for $s = 1$.



(c) Trained function in layer 5 for $s = 5$. **(d)** Base function in layer 5 for $s = 5$.



(e) Trained function in layer 5 for $s = 7$. **(f)** Base function in layer 5 for $s = 7$.

Figure 4.23: Comparison of layer 2 activation functions of ALU layers for sample networks with $s = 1, 5, 7$. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

functions get too broad, optimization might become more difficult because the parameters of the base function i (especially, the amplitude k_i) in this case, has to be fit to decrease the loss for a larger amount of input values. This more extensive fitting process might again slow down optimization and therefore lead to lower accuracies. Nevertheless, such behavior does not explain the failure of the network group with $u = 1$ and $u = 3$: The most likely reason for the failure of these networks is again a too high initial output of the layer which again results in the network steering the input space of the layer in a way such that the output of the layer is 0 for all training vectors.

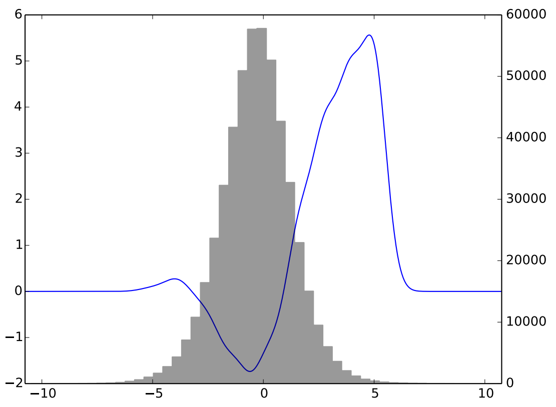
The network group where u has been jointly optimized together with $\mathbf{k}$ shows a slightly worse performance compared to the best performing network group without optimization of u ($u = 0$). On the contrary, the network group where $\mathbf{k}$, u and s have been optimized jointly, shows slightly better performance. A possible explanation for this observation might be, that adding additional parameters slows down the optimization process, given the more complex optimization problem. This more complex optimization process only pays off if it enables us to find a better local minimum. This seems to be the case when optimizing u as well as s, but not when solely optimizing u.

Again, to verify our claims, we take a look at the trained activation functions in layers 2 and 5 for u = 0, 0.3 and 3. As expected, while the activation functions for $u = 0$ and $u = 0.3$ do not have any unusual traits and look quite similar to activation functions we have trained earlier using triangle base functions, the network has again adjusted the input space of layer 5 to circumvent any non-zero output from it in order to minimize the loss in a locally optimal way.

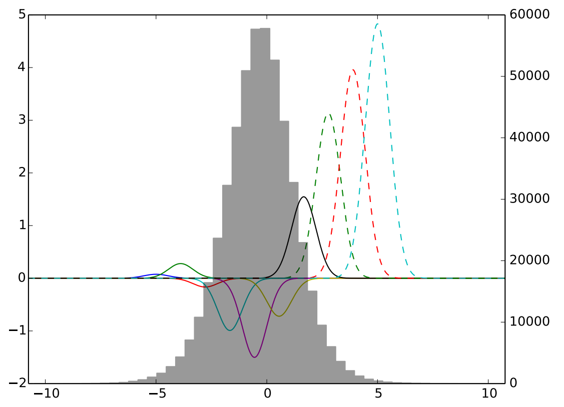
To comprehend the results of this experiment, we have found out that the best initialization for a non-trainable shape parameter is $u = 0$, but this might also be connected to the fact that s is already set rather large compared to the breadth of the input space. Furthermore, we found out that the optimization of the additional parameters s and u brings only small improvement over a static optimization, but with the additional drawback of complicated parameter update formulas and higher training times. Thus, since the optimization of u and s did not pay off compared to a regular parameter grid search, for the remaining experiments we solely optimize $\mathbf{k}$ as done before.

4.8 Comparison On Different Data Sets

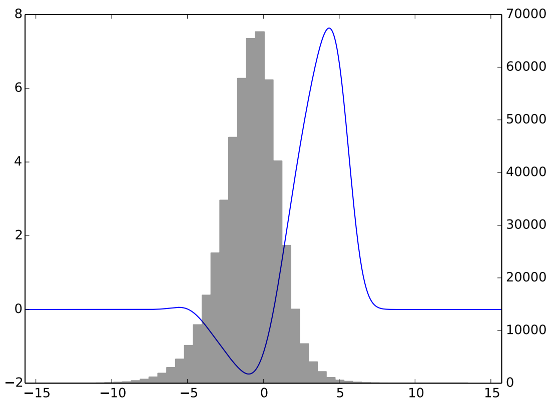
To see how ALUs perform on different data sets, we use the same network as for the previous experiments, but run it on different data sets. We again use the best configuration observed so far, except for input range and shape optimization, since we found out in Section 4.6 that input range optimization brings only little improvement at the cost of a complex optimization process and high training times. Furthermore we again use a "mixed" configuration of ALeNet (see Table 4.1) where we use ALU units behind the convolutional layers and ReLUs in the fully connected layer. As a baseline, we again use an ALeNet architecture with ReLUs in all layers. We train our ALU and baseline network



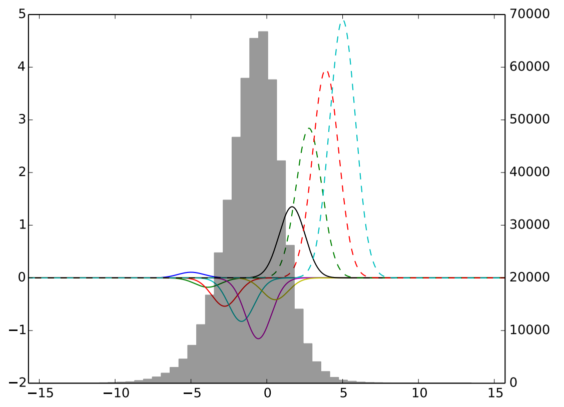
(a) Trained function in layer 5 for $u = 0$.



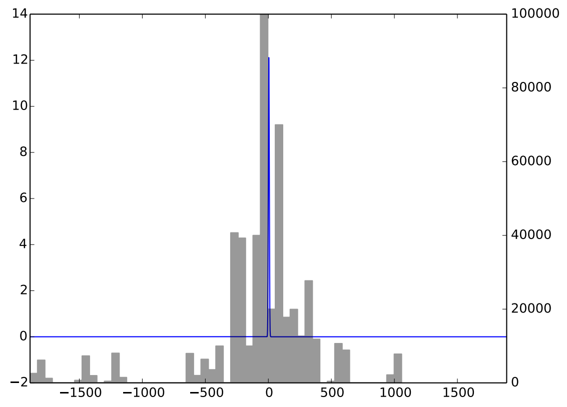
(b) Base function in layer 5 for $u = 0$.



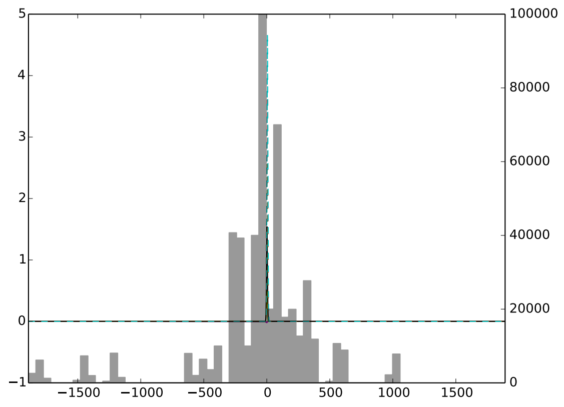
(c) Trained function in layer 5 for $u = 0.3$.



(d) Base function in layer 5 for $u = 0.3$.



(e) Trained function in layer 5 for $u = 3$.



(f) Base function in layer 5 for $u = 3$.

Figure 4.24: Comparison of layer 2 activation functions of ALU layers for sample networks with $u = 0, 0.3, 3$. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

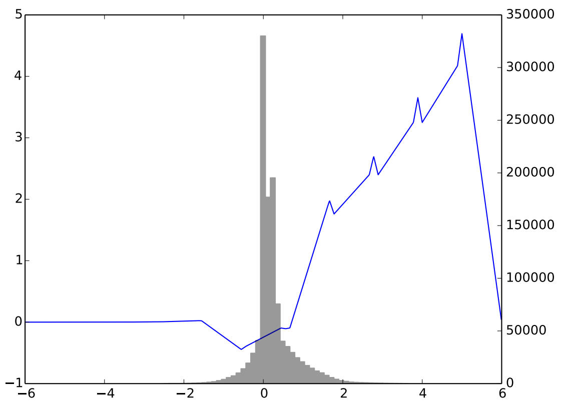
	MNIST	CIFAR10	CIFAR100	SVHN
ALU	99.24 ± 0.05	72.88 ± 0.56	38.26 ± 2.87	91.27 ± 0.21
ReLU	99.21 ± 0.04	69.30 ± 0.62	31.38 ± 1.17	89.98 ± 0.26

Table 4.10: Mean and standard deviation for accuracy scores of our network groups with ALUs trained on the MNIST, CIFAR-10, CIFAR-100 and SVHN data sets as well as the corresponding baseline network groups. Results are averaged over 20 networks each.

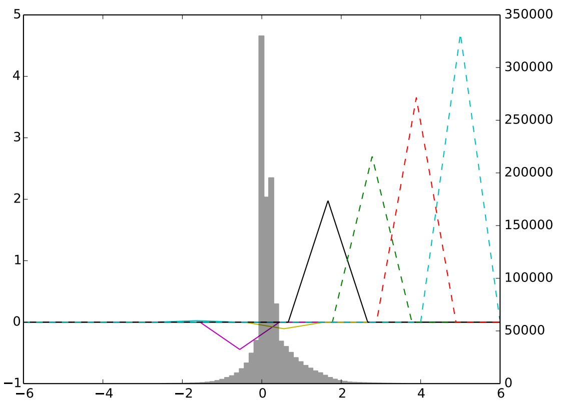
groups on the MNIST[40], CIFAR-100[37] and SVHN[53] data sets . For the CIFAR-100 data set, we additionally performed the following modifications to the network: We extended the final output layer to have 100 neurons instead of 10 due to the 100 different classes of the data set. Furthermore, we had to decrease the base learning rate from $\eta = 0.01$ to $\eta = 0.001$ and increase the number of training iterations from 20000 to 100000. For MNIST and SVHN on the other hand, the network and training process is identical to the ones in our previous experiments.

Table 4.11 shows mean and standard deviation of the accuracies as well as test loss and train loss for each of our trained network groups with triangle base functions for the MNIST, CIFAR-10, CIFAR100 and SVHN data sets as well as the corresponding baseline groups. Therefore, results have been obtained by averaging over 20 networks trained with identical parameters apart from a random initialization of weights in convolutional and fully connected layers.

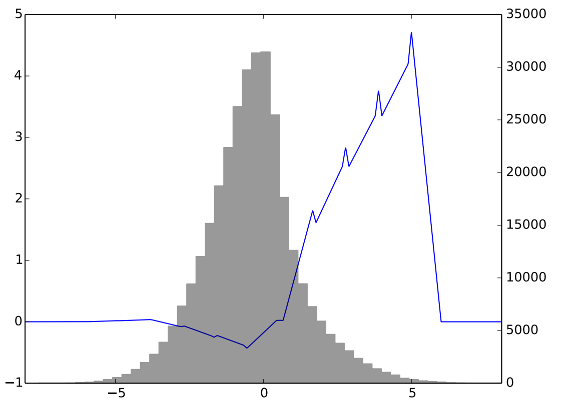
We see that, except for the MNIST data set, where the improvement in average accuracy is vanishing, we manage to outperform the respective baseline network group on all data sets. An interesting observation is that ALUs seem to bring a significant performance gain on harder problems such as the CIFAR-100 data sets, while there is no improvement detectable on data sets which are largely seen as solved, such as the MNIST data set. We take a look at the trained activation functions for the different data sets to see if they distinguish from the trained activation functions we have seen so far: We can see that, while the input spaces of the networks vary, all the activation functions look pretty similar as the ones we have observed before: While the ReLU-like initialization persists, all functions seem to train a kink such that $\sigma(x) < 0$ around the origin. This again supports our theory that high dynamic and distinguishability is an important factor for the activation functions we train. In a nutshell we have seen that, while ALU units do not seem to bring any performance gain for simpler classification problems, but significant gain for harder problems on the other hand. We further have observed that the trained activation functions are similar in shape for our network (ALeNet), independent from any data sets. All in all, based on our results of the analytic experiments, the additional overhead needed in the application of ALUs due to more complex optimization and higher training times compared to ReLU networks might pay off where the descriptive power of regular ReLUs is insufficient, and deeper ReLU network can not be trained, or the amount of parameters is constrained due to hardware limitations.



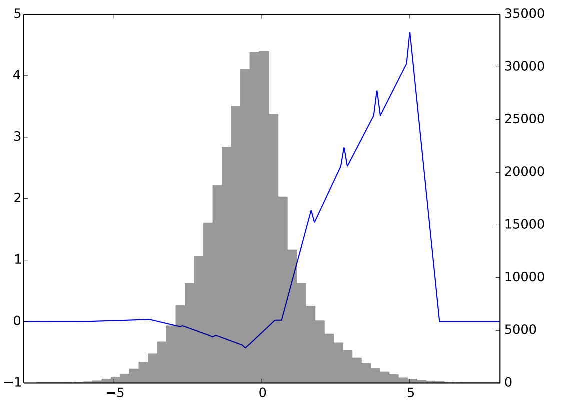
(a) Trained function for layer 2, MNIST data set.



(b) Base function for layer 2,MNIST data set.



(c) Trained function for layer 5, MNIST data set.

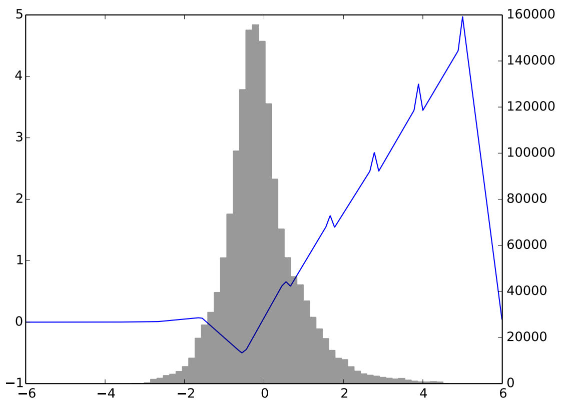


(d) Base function for layer 5, MNIST data set.

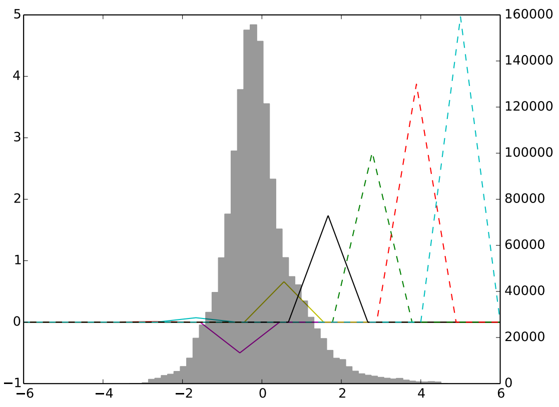
Figure 4.25: Comparison of layer 2 activation functions of ALU layers for sample network trained on the MNIST data set. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

4.9 Comparison To State Of The Art Methods

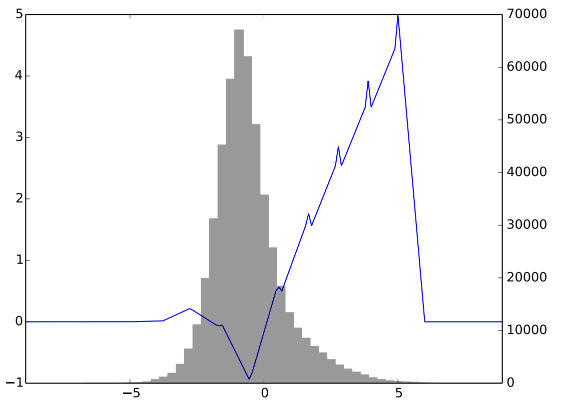
In this experiment we compare our unit to the proposed units to the Adaptive Piecewise Linear (APL) Units as proposed in [1] as well as PReLUs [27]. We use the same network architecture as in [1], to which we will refer in this experiment as DeepNet (see Table 4.2) and train four networks on each of the following data sets: CIFAR-10 [37], CIFAR-100 [37] and SVHN[53]. For each data set, we train one network using ReLU units as a baseline, two networks using APL-units and PReLUs respectively behind every convolutional and fully connected layer, as well as a mixed network using ALUs behind the convolutional layers and ReLUs behind the fully connected layers. For the initialization of APL units



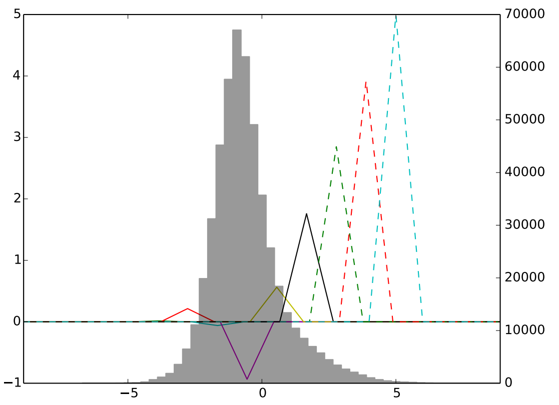
(a) Trained function for layer 2, CIFAR-100 data set.



(b) Base function for layer 2,CIFAR-100 data set.



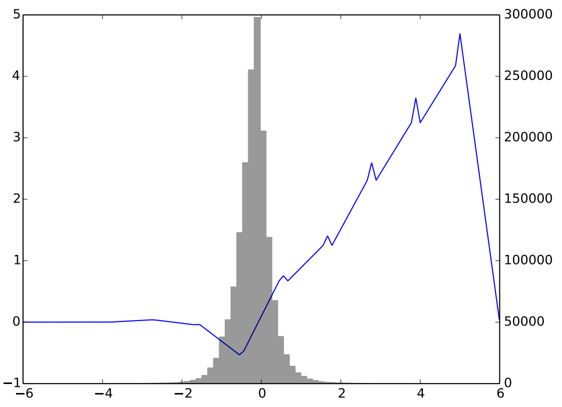
(c) Trained function for layer 5, CIFAR-100 data set.



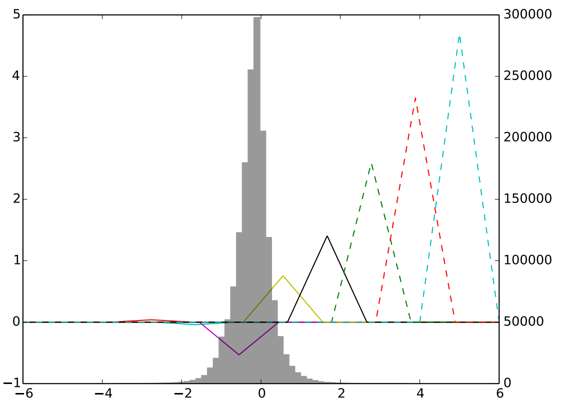
(d) Base function for layer 5, CIFAR-100 data set.

Figure 4.26: Comparison of layer 2 activation functions of ALU layers for a sample network trained on the CIFAR-100 data set. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

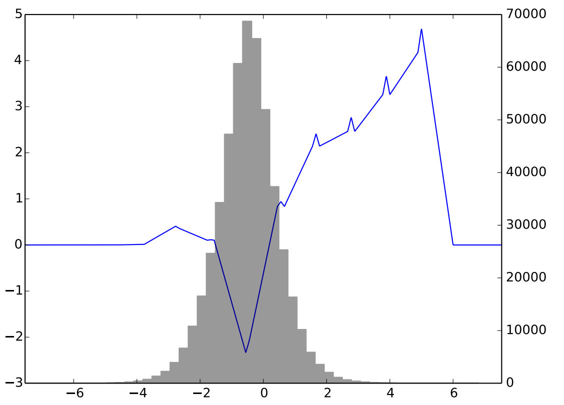
we stick with the parameters used by Agostinelli et al. [1]. Concerning the choice of base functions for our ALUs, we will use ReLUs since they were the only base function that delivered acceptable performance in preliminary experiments. We suspect that the reason for this behavior is twofold. On the one hand, deep networks like the ones we use in this experiment, tend to stretch out the input space. In case of our unit using confined base functions, and thus covering solely a confined input space, this might result in the units becoming inactive for a large amount of samples due to many input points getting shifted out of the active subspace of the trained activation function. Furthermore we argue that a constant, monotonically non-decreasing gradient over large portions of the input space is mandatory for the training in very deep networks, since the probability of gradient



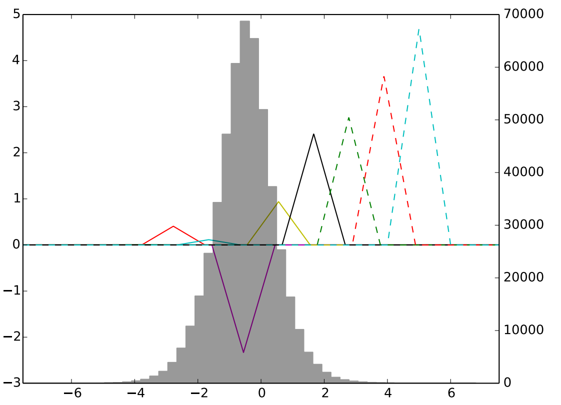
(a) Trained function for layer 2, SVHN data set.



(b) Base function for layer 2, SVHN data set.



(c) Trained function for layer 5, SVHN data set.



(d) Base function for layer 5, SVHN data set.

Figure 4.27: Comparison of layer 2 activation functions of ALU layers for a sample network SVHN data set. The trained activation functions are shown in the left column, while the base functions composing them are shown in the right column.

problems will increase with the number of layers trained in a network. Luckily, ReLUs are non-zero for large portions of the input space, and exhibit on this portions a monotonically non-decreasing gradient and can be used as base functions in deep networks as well.

We use 5 ReLU base functions and initialize the ALUs amplitudes with a single nonzero ReLU with a kink at x=0. Therefore, we effectively start training with a single ReLU at every layer, which has shown to be favorable in deep networks. We set the input range parameter to $s = 3$, the shape parameter is invalid for ReLU units and therefore ignored.

Table 4.11 shows accuracies for CIFAR-10, CIFAR-100 and SVHN data sets trained on DeepNet, using ALU, APL, ReLU and PReLU activation functions. As we can see, APL and ALU perform almost equally on CIFAR-10 data, followed by our ALU which

Method	CIFAR-10	CIFAR-100	SVHN
DeepNet - ReLU	87.49	55.62	95.95
DeepNet - APL [1]	88.38	58.43	96.31
DeepNet - ALU	87.89	56.64	95.90
DeepNet - PReLU [27]	88.37	57.18	96.06

Table 4.11: Accuracies for APL, ALU, PReLU and ReLU activation functions trained within DeepNet (see Table 4.2)

still shows about 0.4% improvement in accuracy over the ReLU. For the CIFAR-100 data set, the gaps between the different methods are larger: APL perform best on CIFAR-100 with more than 1% improvement in accuracy over PReLUs, who again outperform ALUs by about 0.5%. *ALUs* on the other hand, outperform the ReLU baseline by 1% again. Finally, on the SVHN data set, a different picture is drawn: ALUs are about on par with ReLUs, and are outperformed by APLs and PReLUs which are both about 1% better.

All in all, this supports our hypothesis that training activation functions brings larger improvements in performance for harder data sets such as CIFAR-10 and CIFAR-100, whereas it is less useful on data sets which are already largely solved using ReLU units. Despite the fact that ALUs are outperformed here, it has to be noted, that we use significantly less parameters than the APL and PReLU methods and only I (= number of base functions per layer) parameters more per layer than ReLUs. Thus we show that ALU units are also suited to improve performance in deep networks, especially where state-of-the-art performance is required, but the number of parameters is limited e.g. through memory.

5 Conclusion

5.1 Conclusion

In this thesis we proposed a general framework to train a wide variety of non-convex continuous activation functions in neural networks as a part of the regular training process and applied it to various state-of-the-art image recognition problems. We gave a short survey on the current efforts on performing image recognition using convolutional neural networks (CNNs) in general, as well as the application of new units in CNNs, who train their own activation functions in specific. Then we showed that current state-of-the-art approaches on training activation functions can be viewed as particular configurations of a sum of arbitrary base functions. Consequently, we introduced our own framework to train activation functions in CNNs as a parametric sum of arbitrary base functions and showed how to incorporate the training of its amplitude, location and shape parameters into the regular network training process. Furthermore, we introduced techniques to enforce a meaningful optimization process and keep the amount of additional parameters low. Finally we implemented our proposed framework as a unit we call Approximating Learnable Unit (ALU) and compared its performance to Rectified Linear Units (ReLUs) [52] as well as Parametric Rectified Linear Units (PReLUs) [27] and Adaptive Piecewise Linear Units (APLs) [1] on two different network architectures.

First, we trained CNNs with 2 convolutional layers and a fully connected layer with ALU and ReLU units respectively. On this architecture, our unit outperformed the ReLU baseline on the CIFAR-10 [37], CIFAR-100 [37] and SVHN [53] data sets and showed equal performance on the MNIST [40] data set. Additionally, to compare our unit to the state-of-the-art PReLU and APL units, we trained a second class of networks using a deeper architecture featuring 3 convolutional and 2 fully connected layers, where we applied Dropout [29] before and after every pooling layer. Here, our units performed similar to PReLU and APL units and outperformed ReLUs on the CIFAR-10 and CIFAR-100 data sets. On the SVHN data set, our units were on par with the ReLUs while PReLU and APL units exhibited slightly better performance.

In practical terms, we found that the most efficient way to train our activation functions is to train amplitude parameters of our base functions only, and keep input space and shape parameters fixed. Optimizing input range and shape parameters showed only little improvement at the cost of increased implementation effort. Thus we the conclude that the optimization of input range and shape parameters does not pay off when following the initialization strategy we chose for shape and input range.

As one would expect when training activation functions, we faced some overfitting issues, which seemingly concerned rather fully connected layers than convolutional layers. We found two ways to deal with the observed overfitting effects. One possible solution is to use our trainable activation functions only for convolutional layers and apply ReLUs in the fully connected layers instead, since convolutional layers seem to exhibit less overfitting when trained with our units. The second possibility is to renormalize the amplitudes of our units such that their L1 norm is fixed after every gradient update, which can be interpreted as a measure of enforced regularization.

The activation functions we trained look similar for different networks and data sets. While we are not able to recognize the trained activation function as one of the commonly used, a definite feature we discovered was a strong on-off behavior, contrary to ReLUs, which are sparse.

Concerning the pick of the optimal base function our experiments showed mixed results. While triangles showed best performance on smaller networks, we were unable to train our models with triangle, Gaussian and cosine base functions for deep networks. Thus we report the best results on the deeper network using ReLUs as base functions, which is most likely due to their constant gradient on very large portions of the input domain.

In a nutshell, we have shown that the training of activation functions using this general method improves the performance of CNNs on state-of-the-art recognition problems at little additional cost in terms of parameters. While from the base functions we considered only ReLUs were found applicable on deep networks within our framework, and there is still room for improvement compared to current state-of-the-art methods, the framework and techniques derived in this thesis provide a solid foundation for future work in this field. Possible future work might include further investigations on suitable base functions for deep networks and the application of the introduced methodologies on current state-of-the-art methods, such APLs optimizing base function positions on a fixed grid controlled by an optimizable input range. Further, interesting research might include implementing trained functions as a fixed activation function and studying their performance to find new suitable activation functions.

Bibliography

[1] Agostinelli, F., Hoffman, M., Sadowski, P., and Baldi, P. (2014). Learning activation functions to improve deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR).* (page 3, 4, 5, 19, 21, 23, 24, 25, 27, 28, 36, 39, 44, 53, 59, 85, 86, 88, 89)

[2] Anguelov, D., Lee, K., Gökturk, S., and Sumengen, B. (2007). Contextual identity recognition in personal photo albums. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1)

[3] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. In *Proceedings of European Conference on Computer Vision (ECCV).* (page 2, 5, 21)

[4] Belongie, S., Malik, J., and Puzicha, J. (2002). Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522. (page 2, 5, 21)

[5] Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning.* (page 14)

[6] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag New York, Inc. (page 8, 10, 12, 19, 20)

[7] Boden, M. (2008). *Mind As Machine: A History of Cognitive Science.* Oxford University Press, Inc. (page 5)

[8] Boyd, S. and Vandenberghe, L. (2004). *Convex optimization.* Cambridge university press. (page 9, 11)

[9] Broomhead, D. and Lowe, D. (1988). Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, DTIC Document. (page 27, 34)

[10] Bryson, A. E. (1961). A gradient method for optimizing multi-stage allocation processes. In *Proceedings of the Harvard University Symposium on Digital Computers and Their Applications.* (page 16)

[11] Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). Brief: Binary robust independent elementary features. In *Proceedings of European Conference on Computer Vision (ECCV).* (page 2, 5, 21)

[12] Chatfield, K., Philbin, J., and Zisserman, A. (2009). Efficient retrieval of deformable shape classes using local self-similarities. In *Proceedings of International Conference on Computer Vision (ICCV).* (page 2, 5, 21)

[13] Chen, Y., Yu, W., and Pock, T. (2015). On learning optimized reaction diffusion processes for effective image restoration. In *IEEE Conference on Computer Vision and Pattern Recognition.* (page 25, 35)

[14] Dahl, G. E., Sainath, T. N., and Hinton, G. E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Proceedings of International Conference on Speech and Signal Processing (ICASSP).* (page 20)

[15] Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1, 2, 5, 21)

[16] Dean, T., Ruzon, M., Segal, M., Shlens, J., Vijayanarasimhan, S., and Yagnik, J. (2013). Fast, accurate detection of 100,000 object classes on a single machine. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1)

[17] Deng, J., Ding, N., Jia, Y., Frome, A., Murphy, K., Bengio, S., Li, Y., Neven, H., and Adam, H. (2014). Large-scale object classification using label relation graphs. In *Proceedings of European Conference on Computer Vision (ECCV).* (page 1)

[18] Dollar, P., Belongie, S., and Perona, P. (2010). The fastest pedestrian detector in the west. In *Proceedings of British Machine Vision Conference (BMVC).* (page 1)

[19] Dreyfus, S. E. (1962). The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45. (page 16)

[20] Elliott, D. and Keller, F. (2013). Image description using visual dependency representations. In *Proceedings of Empirical Methods in Natural Language Processing.* (page 1)

[21] Farhadi, A., Hejrati, M., Sadeghi, M. A., Young, P., Rashtchian, C., Hockenmaier, J., and Forsyth, D. (2010). Every picture tells a story: Generating sentences from images. In *Proceedings of European Conference on Computer Vision (ECCV).* (page 1)

[22] Felzenszwalb, P., McAllester, D., and Ramanan, D. (2008). A discriminatively trained, multiscale, deformable part model. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 2)

[23] Fischler, M. and Elschlager, R. (1973). The representation and matching of pictorial structures. *IEEE Transactions on Computers.* (page 1)

[24] Fleuret, F. and Geman, D. (2001). Coarse-to-fine face detection. *International Journal of computer vision*, 41(1-2):85–107. (page 1)

[25] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. In *Proceedings of the International Conference on Machine Learning.* (page 3, 21, 24, 25, 27)

[26] Grother, P. J. (1995). Nist special database 19 handprinted forms and characters database. (page 41)

[27] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of International Conference on Computer Vision (ICCV).* (page 1, 3, 4, 5, 17, 19, 24, 25, 27, 28, 36, 39, 44, 85, 88, 89)

[28] Heisele, B., Serre, T., and Poggio, T. (2007). A component-based framework for face detection and identification. *International Journal of Computer Vision (IJCV)*, 74(2):167–181. (page 1)

[29] Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580.* (page 20, 89)

[30] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. Master's thesis, Technische Universitaet Muenchen. (page 18)

[31] Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 148(3):574–591. (page 22)

[32] Jégou, H., Douze, M., and Schmid, C. (2010). Improving bag-of-features for large scale image search. *International Journal of Computer Vision (IJCV).* (page 1)

[33] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093.* (page 41)

[34] Jing, Y. and Baluja, S. (2008). Visualrank: Applying pagerank to large-scale image search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1877–1890. (page 1)

[35] Karpathy, A. and Fei-Fei, L. (2014). Deep visual-semantic alignments for generating image descriptions. *arXiv preprint arXiv:1412.2306.* (page 1)

[36] Kelley, H. J. (1960). Gradient theory of optimal flight paths. *American Rocket Society Journal*, 30(10):947–954. (page 16)

[37] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report. (page 5, 19, 84, 85, 89)

[38] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems.* (page xv, 2, 3, 14, 23, 29, 36)

[39] Kulis, B. and Grauman, K. (2009). Kernelized locality-sensitive hashing for scalable image search. In *Proceedings of International Conference on Computer Vision (ICCV).* (page 1)

[40] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the Institute of Electrical and Electronics Engineers.* (page xv, 3, 21, 22, 23, 27, 29, 41, 42, 84, 89)

[41] LeCun, Y., Bottou, L., Orr, G., and Müller, K. (2008). Efficient backprop. In *Neural networks: Tricks of the trade.* Springer. (page 12, 15, 18)

[42] LeCun, Y., Kavukcuoglu, K., and Farabet, C. (2010). Convolutional networks and applications in vision. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS).* (page 22)

[43] Leung, T. K., Burl, M. C., and Perona, P. (1995). Finding faces in cluttered scenes using random labeled graph matching. In *Proceedings of International Conference on Computer Vision (ICCV)*, pages 637–644. IEEE. (page 1)

[44] Leutenegger, S., Chli, M., and Siegwart, R. Y. (2011). Brisk: Binary robust invariant scalable keypoints. In *Proceedings of International Conference on Computer Vision (ICCV).* (page 2, 5, 21)

[45] Lin, M., Chen, Q., and Yan, S. (2014). Network in network. In *Proceedings of the International Conference on Learning Representations (ICLR).* (page 3, 5, 14, 19, 24)

[46] Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's thesis, University of Helsinki. (page 16)

[47] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision (IJCV)*, 60(2):91–110. (page xv, 2, 5, 21)

[48] Maas, A., Hannun, A., and Ng, A. (2013). Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing.* (page 19)

[49] McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics.* (page 14)

[50] Mikolajczyk, K., Schmid, C., and Zisserman, A. (2004). Human detection based on a probabilistic assembly of robust part detectors. In *Proceedings of European Conference on Computer Vision (ECCV).* Springer. (page 1)

[51] Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the Association for Computing Machinery.* (page 41)

[52] Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning.* (page 19, 36, 89)

[53] Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning.* (page 42, 84, 85, 89)

[54] Opelt, A., Pinz, A., and Zisserman, A. (2006). A boundary-fragment-model for object detection. In *Proceedings of European Conference on Computer Vision (ECCV).* (page 2)

[55] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks.* (page 11, 12)

[56] Rogez, G., Rihan, J., Ramalingam, S., Orrite, C., and Torr, P. (2008). Randomized trees for human pose detection. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1)

[57] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors. (page 16)

[58] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252. (page 24)

[59] Sermanet, P., Kavukcuoglu, K., Chintala, S., and LeCun, Y. (2013). Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1)

[60] Sinha, A., Banerji, S., and Liu, C. (2012). Novel color gabor-lbp-phog (glp) descriptors for object and scene image classification. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing.* (page 2, 5, 21)

[61] Sivic, J., Zitnick, C. L., and Szeliski, R. (2006). Finding people in repeated shots of the same scene. In *Proceedings of British Machine Vision Conference (BMVC).* (page 1)

[62] Springenberg, J. and Riedmiller, M. (2013). Improving deep neural networks with probabilistic maxout units. *Computing Research Repository.* (page 3, 21)

[63] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958. (page 20)

[64] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 3, 5, 14, 19)

[65] Szeliski, R. (2010). *Computer Vision: Algorithms and Applications.* Springer-Verlag New York, Inc. (page 1, 2, 5, 22)

[66] Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1)

[67] Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970. (page 41)

[68] Viola, P. and Jones, M. J. (2004). Robust real-time face detection. *International Journal of Computer Vision (IJCV)*, 57:137–154. (page xv, 1, 2)

[69] Werbos, P. (1982). Applications in advances in nonlinear sensitivity analysis. In *Proceedings of the International Federation for Information Processing Conference.* (page 16)

[70] Wu, Z., Ke, Q., Isard, M., and Sun, J. (2009). Bundling features for large scale partial-duplicate web image search. In *Proceedings of Computer Vision and Pattern Recognition (CVPR).* (page 1)

[71] Zeiler, M. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Proceedings of European Conference on Computer Vision (ECCV).* (page 3, 27)

[72] Zhang, L., Chen, L., Li, M., and Zhang, H. (2003). Automated annotation of human faces in family albums. In *ACM International Conference on Multimedia.* (page 1)

[73] Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *arXiv:1506.06724 (CVPR submission).* (page 1)