Spread Maximization

a Novel Unsupervised Learning Paradigm Applied to Convolutional Neural Networks

by

tim kuipers

B.A. (Philosophy), Utrecht University, 2014 ica: 3149099

Supervisors:

prof. dr. a.p.j.m. siebes

dr. a.j. feelders

dr m.a. wiering

Master's Thesis

submitted in fulfillment of the requirements
for the degree of

Master of Science

in the programme

Technical Artificial Intelligence

at the

Faculty of Science
Department of Information and Computing Sciences
Utrecht University

August 2014 , Utrecht

ACKNOWLEDGEMENTS

I wish to thank Thomas Krak for his endless patience, counterbalance, good will and support. I thank Ad Feelders and Marco Wiering for supporting me throughout the project and providing valuable suggestions. I thank Menno van den Berg for listening to me whenever I need to talk.

Spread Maximization

a Novel Unsupervised Learning Paradigm Applied to Convolutional Neural Networks

by

Tim Kuipers

ABSTRACT

Unsupervised learning provides a way to extract features from data which can be used to pre-train Artificial Neural Networks (ANNs) improving on the performance of such networks. Convolutional Neural Networks (CNNs) are a kind of ANN designed for image processing. Existing unsupervised learning techniques for CNNs are frequently based on reconstructions of the input from the output of a network layer.

We seek to establish a new paradigm for unsupervised learning in CNNs. Can CNNs extract helpful features by training based on the output of the network rather than the input? We formulate an objective which is defined by some basic properties we would like an ANN to have. Our objective propels the outputs of a layer in a neural network to be *spread out* through the output space, while keeping the weights low.

A first interpretation of spread maximization is dichotomization, which leads to an approach which maximizes the determinant of the covariance matrix of the layer outputs, while a second interpretation is uniformization, which leads to an approach which minimizes the distance between the current output distribution and the uniform distribution.

A third overarching approach is provided which can be used for either kind of spread maximization. It is based on a generalization of Hebbian learning, which reduces to the Generalized Hebbian Algorithm (GHA) for a specific simple kind of ANN. Since GHA causes the weight vectors of the network to converge to the first n principal loading vectors, which is the objective of Principal Component Analysis (PCA), we say that the application of generalized Hebbian learning to CNNs performs Pooled Convolutional Component Analysis.

Experimental results show that spread maximization techniques can outperform conventional techniques such as Pooling Convolutional Auto-Encoders. One method for dichotomization achieves an error rate of 1.3% on the LeNet-1 network structure, beating the 1.7% it has been reported to achieve without unsupervised learning.

In order to maximize the potential of our unsupervised learning techniques, we introduce a new kind of pooling function which reduces the error of purely supervised learning on LeNet-1 to 1.44% and a supervised pre-training stage which seems to be beneficial to any type of unsupervised pre-training technique.

CONTENTS

1

INTRODUCTION

Artificial Neural Networks (ANNs)[22, 6, 19] are widely used in the fields of machine learning and pattern recognition. When training a large neural network, optimization might easily get stuck in local optima. Deep neural networks[2] are therefore often pre-trained by applying some unsupervised learning mechanism to the lower level layers.

Existing unsupervised learning techniques frequently make use of some form of reconstruction of the input of a layer, based on its outputs. Examples are Restricted Boltzmann Machines (RBMs)[5] and Auto-Encoders (AEs)[35]. These two techniques are widely used for unsupervised pre-training of deep ANNs.

We wondered whether it is possible to define an unsupervised learning criterion which is in no way based on a reconstruction of the input. Can we define an unsupervised objective which is defined in terms of the outputs of a layer and its weights? The objective we present is based on some qualities we would like ANNs to exhibit. Can an objective which only maximizes some desirable qualities of ANNs result in a well performing unsupervised learning criterion? Would the features thus extracted be intuitively interpretable and would they be helpful in classification? In order to answer these questions we limit our scope to a specific kind of ANN used for image processing: the Convolutional Neural Network (CNN)[21].

The desirable qualities are derived from undesired properties. We wouldn't like an ANN to contain duplicate features, because then we would perform unnecessary computations, which entails inefficiency. We also wouldn't like an ANN to contain features with zero weights, since these features would always output the same value, and thus never convey information about the input. Indeed, we would like the features to capture quite different information and to respond quite differently to different input. We propose new techniques which encourage just these properties. The goal of these new techniques can be viewed as *spreading out* the output data points throughout the output space. Spread is an ambiguous term—the two ways in which it may be interpreted give rise to two different approaches to Spread Maximization (SM).

The first interpretation is what we call dichotomization. It propels the output data toward the boundaries of the output space. The first technique we propose to perform dichotomization is called Eigenvolume Expansion-Dichotomization (EED). It maximizes the variance in each direction in its eigenspace by maximizing the volume of a hyperrectangle aligned with the

eigenspace of the output data. This is performed by maximizing the determinant of the covariance matrix of the output data. At a global optimum of this objective the output data is binarized, which seems to be a disadvantage.

The second interpretation of spread maximization is what we call uniformization. It causes the output data to be evenly spread out through the output space. The first technique for uniformization we propose is called Pre-Sigmoid Gaussian-Uniformization (PSGU). It models the pre-sigmoid output distribution with a multivariate normal distribution and minimizes the Kullback-Leibler (KL)-divergence between the fitted distribution and the optimal pre-sigmoid distribution which leads to a uniform output distribution.

The two different variants of SM are both showed to be underdetermined by themselves; they allow for a vast set of global optima of which only a small subset is intuitively viewed as good. This problem is solved by restricting ourselves to the global optima with small weights.

The objective of maximizing spread with the smallest weights can be transmuted to the objective of maximizing the variance and decorrelation of the output neurons under the constraint that the weight vectors have unit length, called the constrained Hebbian Objective (HO). After training the ANN with the constrained HO, we can apply a transformation to the network in order to end up with an ANN which maximizes spread.

The constrained HO coincides with the objective of Principal Component Analysis (PCA) when applied to a simple single layer Multilayer Perceptron (MLP)[32]. PCA is known to be achieved by the Generalized Hebbian Algorithm, which is an iterative procedure which causes the weight vectors of an ANN to converge to the loading vectors of the principal components.[1] We derive a formula for the constrained HO applied to CNNs, called the Convolutional Hebbian Algorithm (CHA), and use it to define a Hebbian SM technique for CNNs. CHA in combination with the network transformation forms what is called CHA-based SM.

When applying the same technique to a simple MLP layer, it is equivalent to determining the weights by PCA and whitening the activations of the output neurons.

We compare our methods to Pooling Convolutional Auto-Encoders (PCAEs), which are a form of auto-encoder for CNNs which incorporate the pooling function much like our methods do.

1.1 OVERVIEW

Chapter 2 covers ANNs. Section 2.1 covers ANNs in general, common network structures and notational basics for describing them. The next section

1 Often the loading vectors are themselves called the principal components, leading to confusing terminology.

covers CNNs, common parameter settings and again notational basics for describing common types of CNN. Here we also introduce new variants of conventional CNNs, which use a new kind of pooling function. The last section of the next chapter covers learning algorithms, the objective functions they use, regularization terms, and network initialization.

Chapter 3 covers some common unsupervised learning techniques relevant for this thesis. The first section covers PCA, and how it is related to ANNs. The next section covers Hebbian learning; it can be viewed as a predecessor to CHA-SM and is inherently related to PCA. The last section covers autoencoders, which can be viewed as an alternative to the methods we present; it serves as a baseline in order to assess the performance of our methods.

Chapter 4 covers our methods for performing SM. Two introductory sections cover the concept of spread and how CNNs should be handled when using SM. Section 4.3 covers one interpretation of SM, *dichotomization* and presents a method to perform it: EED. The next section covers the other interpretation, *uniformization*, and a method to perform it: PSGU. Finally we present in the last section an overarching method which is able to perform either kind of SM: CHA-SM.

Chapter 5 covers the experiments performed to assess the performance of the unsupervised learning techniques we introduce. The first section gives a detailed description of the experimental setup and the settings used in the experiments The next section discussed the features extracted from a very simple synthetic data set. The third section evaluates the features extracted from the MNIST dataset by themselves without adjusting them by a supervised learning phase and the last section assesses how well the unsupervised learning techniques work as pre-training techniques for a subsequent supervised learning phase. Here we also introduce a combination of the types of evaluation covered by these two sections, constituting supervised pre-training.

Chapter 6 concludes this thesis. Section 6.1 summarizes the ideas and findings of this thesis. Section 6.1.1 discusses the relevance and meaning of the experimentation results. Section 6.2 covers what questions remain open and available for future research.

1.2 NOTES

A complete list of all acronyms used in this thesis can be found in appendix A. Appendix B contains an overview of all the notational standards used in this thesis, so that one does not have to browse through all pages in order to find the meaning of a symbol. At the back of this thesis an index is provided, so that technical terms can easily be looked up.

2

NEURAL NETWORKS

The brain is what causes an animal to perform complex behavior. It is the cornerstone of mans greatest achievements. Therefore, it serves as a great inspiration to creating artificial intelligence. Artificial Neural Networks (ANNs) are models of structures which resemble a neural network such as the brain. Computational neuroscientists create ANNs in order to model the human brain, while computer scientists (a.o.) create ANNs in order to analyze or convert data.

In this chapter we explain one of the most influential types of ANNs, the Multilayer Perceptron (MLP) (section 2.1) and introduce Convolutional Neural Networks (CNNs) (section 2.2), the type of ANN which is central in this thesis. Section 2.1 and 2.2 introduce the framework of notation used in further chapters. Also, some critical analysis of what is currently common practice is performed, which might even be inspirational to someone familiar with neural networks. Moreover, some new and/or uncommon techniques are described.

2.1 SIMPLE ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANNs) abstract away from many of the specific workings of biological neural networks. Instead of sending spike trains (sequences of binary signals), ANNs send graded signals. Also the function by which an artificial neuron determines its output is chosen to be mathematically simple, while the physical processes in biological neurons are not that easily characterized.

2.1.1 *Multilayer Perceptron*

The most simple form of ANNs is the Multilayer Perceptron (MLP). This type of ANN puts heavy restrictions on the structure of the network. Neurons are organized into *layers* which are *fully connected*, i.e. the neurons in two consecutive layers are all connected. The layers are divided into the input layer a.k.a. visible layer, *hidden layers* and the output layer. This organization makes it an example of a *feedforward neural network*; the signals from the input layer are propagated forward toward the output layer, without any *recurrent* connections feeding back information to lower layers. This property is particularly suited for processing non-temporal data: the output can be computed in a single pass over the network. It also makes

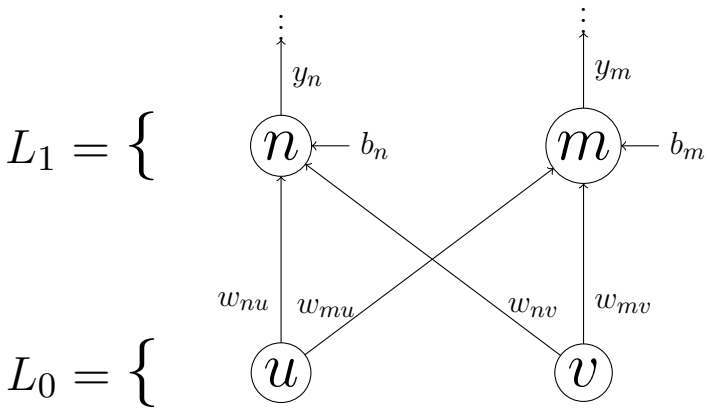


Figure 2.1: A schematic of an example of an ANN

for simple formulas, since the output can be computed by applying a single function to the input.

The strength of connectivity between two neurons is modeled by a real number, called the *weight* of the connection. Abstracting away from synaptic mechanisms such as inhibition and excitation, we allow for the weight to be both positive and negative. Note that a weight equal to zero is equivalent to having no connection at all.

The function by which the output of a neuron is determined in an MLP is quite a simple one. For each neuron that is input to a given neuron, we compute its contribution to the local *activation* by multiplying the weight of the connection by its output. The *weighted inputs* are accumulated by a sum to form the local activation of the neuron under consideration. In order to determine its output, we apply a *transfer function* σ (a.k.a. *activation function*) to the local activation. See figure 2.1 for visual support. Now we can describe this process mathematically. For each hidden layer, the output of a neuron y_j is given by equation 2.1. The output signals of the input layer are given by input data point presented to the network. The output layer neurons are governed by a function which depends on the objective of the network (see section 2.3).

We can write an equation equivalent to equation 2.1 using matrix notation.

$$\mathbf{y} = \sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{2.2}$$

The use of linear algebra leads to simple formulas. However, it is not always insightful to reduce the formula of an ANN in such a way; the relation between the elements of the vectors (and matrices) and neurons (and connections) is obscured by representing them in a structure which doesn't necessarily coincide with the structure of the network. This holds even more so for Convolutional Neural Networks, which are covered by section 2.2.

$$y_j(\mathbf{x}, \mathbf{W}) = \sigma\left(\sum_{i=1}^{N} \{w_{ji}x_i\} + b_j\right) \tag{2.1}$$

where

- $\mathbf{x}$ is the vector of input signals to this layer;
- N is the number of input neurons;
- $\mathbf{W}$ is the matrix of all connection weights $w_{..}$, where each row $\mathbf{w}_j$ is a vector of weights from all input neurons to neuron j;
- σ is the transfer function used in this layer;
- $\mathbf{b}$ is a vector of bias terms, such that b_j is the bias of neuron j.

It is because of the matrix notation that the weight on a connection from i to j is w_{ji} instead of w_{ij}; since the connections to j are given by the j^{th} row of the weight matrix, the indices need to be switched compared to what is intuitive notation.

2.1.2 *Artificial Neural Networks in General*

Above we have described one of the most simple forms of ANNs, namely the MLP. In order to characterize ANNs we describe a more general form of ANNs and define concepts by which MLPs differ from other ANNs. The kind of ANNs described below are not to be viewed as so general that they can capture the workings of any ANN, but they are general enough such that any neural network described in this thesis can be seen as an instance of one. This section serves to introduce notation and nomenclature used in further chapters of this thesis.

NETWORK STRUCTURE The network structure of an ANN can be viewed as a directed graph $G = \langle N, \multimap \rangle$, where N is the set of neurons and $\multimap$ is the connectivity relation of connections between neurons[15, Sec. 1.4]. If this graph is acyclic, we meet the requirements for a feedforward neural network. On the other hand, when the graph may contain cycles we are concerned with a *recurrent neural network*.

Another property of MLPs is that they have their neurons structured in layers, where connections can only exist between two consecutive layers; in it's basic form MLPs don't have *skip-layer connections* or lateral connections within one layer. This is also the case for the kind of CNNs we consider in this thesis.

We can add further requirements to the graph in order to satisfy the constraints of an MLP. MLPs have *fully connected layers*. Each neuron has a connection to each neuron in the next layer. We call the set of input neurons to a given neuron its *local field*.[1] Whereas for an MLP the local field of any neuron consists of the whole preceding layer, we show different kinds of network structure below, when describing CNNs.

ARTIFICIAL NEURONS Now that we have described properties of the network structure, we can move on to describe the mechanism by which the output of a neuron is computed in general. For a (non-input) neuron $j \in N$, the output is computed by mapping a transfer function over the sum of all inputs connected to neuron j multiplied by the weights of those connections, offsetted with a bias term. We associate a neuron with the mathematical variable giving its output signal. The output of a neuron can then be represented by the following mathematical formula:

$$y = \sigma\left(\sum_{i:i\multimap j} \{w_{i\multimap j} i\} + b_j\right) \tag{2.3}$$

In this formula $w_{i\multimap j}$ is the weight of the connection from neuron i to j, which was represented by w_{ji} in the previous section. We divert from the notation used there since the weights cannot in general be represented in a matrix.

Multiple connections may be governed by the same weights, constituting *weight sharing*. When connection $k \multimap l$ shares weights with $i \multimap j$ we say that both $w_{i\multimap j}$ and $w_{k\multimap l}$ evaluate to the same underlying variable, e.g., w_i^j. Here we introduce the new notation w_x^y so that we can refer to a weight variable apart from the connections which use it.

Generally we view biases as weights as well. In what follows one may assume the term 'weights' to refer to connection weights as well as biases; we explicitly indicate otherwise. The weights are generally the only parameters of the model: they are updated by the learning algorithm, in contrast to *hyperparameters*, which are generally set by hand or changed during learning, but not by the learning algorithm itself.

2.1.3 *Choice of Transfer Function*

Usually all hidden neurons in a network get assigned the same transfer function σ, though we consider using different transfer functions for different layers. It is important for the transfer function to contain non-linearities. It is a well known fact that when using a linear function in consecutive layers of an MLP, neurons become redundant. Instead of connections to

1 The local field should not be confused with the local field potential, a neurological term signifying what can be interpreted as the *activation* of an artificial neuron.

and from a neuron with a linear transfer function, we can introduce new connections which bypass the neuron and directly supply input to further neurons to which it is connected. By setting the weights on these connections to the right value, we can eliminate the neuron and have an ANN which is equivalent in functionality. See appendix C.1 for a full proof.

It is common to use a sigmoid function as transfer function. Often the *logistic sigmoid function* is used as transfer function σ:

$$\sigma^l(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

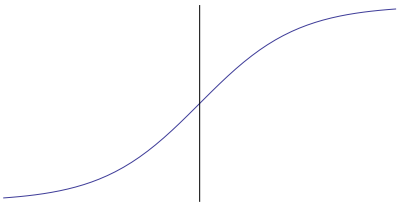


Figure 2.2: A plot of the tanh sigmoid function and/or the logistic sigmoid function.

A characteristic of sigmoid functions is that their plots look like a slanted 'S', see figure 2.2. The logistic sigmoid's output is limited to the interval $(0, 1)$: as $x \to -\infty$ or $x \to \infty$ we see that $\sigma^l(x) \to 0$ or $\sigma^l(x) \to 1$ respectively.

Another commonly used sigmoid function is the *hyperbolic tangent* . It is a soft approximation of the signum function, which is given by:

$$\text{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2.5}$$

It can conveniently be defined in terms of the logistic sigmoid:

$$\tanh(x) = 2\sigma^l(2x) - 1 \tag{2.6}$$

The main difference is that the output of the hyperbolic tangent function is limited to the interval $(-1, 1)$. The fact that it is also stretched along the x-axis is irrelevant, since multiplying all weights can lead to a new weight configuration which compensates for the stretching.

Choosing between these two transfer functions is not an easy task. There are quite some considerations to take into account. The choice might depend on the input data, the objective of the network and on practical considerations concerning signal processing.

SYMMETRY The hyperbolic tangent function is an odd function, meaning that $\tanh(-x) = -\tanh(x)$, which makes it rotationally symmetric about the origin. This means that when the neuron's local activation is inversed, the effect of the output of this neuron on the next neurons is inversed.

This is not the case for the logistic sigmoid function. For a negative local activation, the logistic sigmoid function tends toward zero, effectively negating any effect the neuron has on succeeding neurons.

For high-level neurons it makes more sense to employ the logistic sigmoid function, since we expect high-level neurons to correspond to high-level features, which are either present or not present in the input; an output value below zero would then be less intuitively interpretable.

For lower-level visual concepts the use of an odd transfer function is more reasonable. Suppose a neuron represents a horizontal line. It could then fire positively when the line is white and the background black, and negatively when the network is presented with a black horizontal line on a white background. Visual input data seems a good candidate for which to use the hyperbolic tangent as transfer function for low-level neurons.

2.2 CONVOLUTIONAL NEURAL NETWORKS

A Convolutional Neural Network (CNN)[21, 8, 23] is a special type of layered feedforward ANN designed for image processing. It makes use of the fact that the pixel inputs are ordered in a 2-dimensional grid a.k.a. a *map*. It also makes use of the fact that the same objects may appear in different regions of the image. A *convolution layer* evaluates features at regular intervals in the image, recording in what respect each feature is present at each place in the image in order to form the output of the convolution layer.

It is common practice in the use of CNNs to insert a *pooling layer* after/above each convolution layer. In a pooling layer the outputs of a convolution layer are condensed into maps of lower resolution. Figure 2.3 depicts a simplified CNN convolution and pooling layer. In section 2.2.1 we explain and discuss convolution layers, while in section 2.2.2 we explain and discuss pooling layers.

2.2.1 *Convolution Layers*

Before we formalize the notion of a convolution layer, we must introduce some new notational functionality. In order to formalize the grid ordering of pixels we assign the inputs their position in the grid; for neuron i at location (x, y) we write $i_{(x,y)}$.

Each layer L_n is subdivided into different feature maps Z_m; each neuron $i \in L_n$ belongs to some map, i.e. $i \in Z_m$ and the map only contains neurons from that layer: $Z_m \subset L_n$. The input layer generally consists of one image

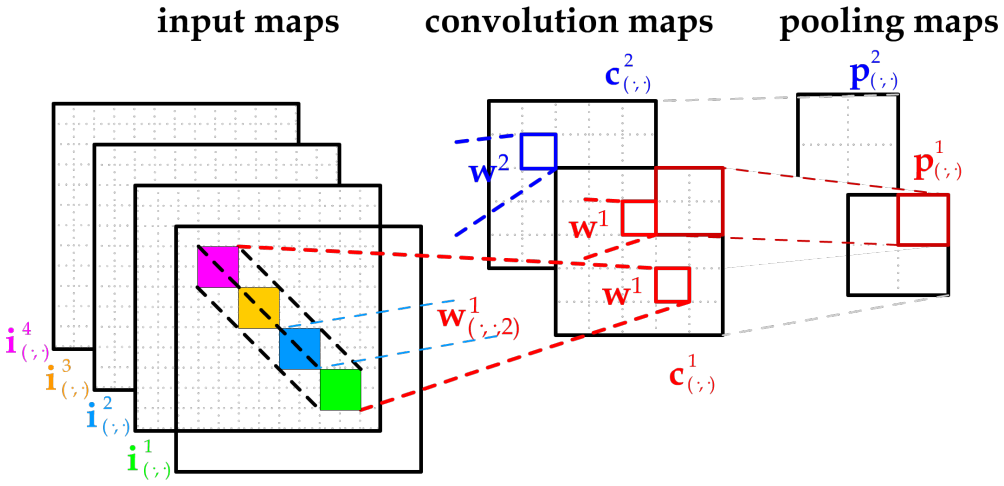


Figure 2.3: Schemata of a convolution and pooling layer containing two features. Note that while the local input field of a neuron in the convolution map is a region within each input map, the local input field of a neuron in the pooling map consists only of a patch within one convolution map. All weights and neurons are colored by feature.

which may be represented by a single feature map in case of a grayscale image and three feature maps in the case of a colour image.[2] For neuron i at location (x, y) of feature map z we write $i^z_{(x,y)}$.

We orient the grids in such a way that the position $(0, 0)$ is the upper left corner. We explain below how hidden neurons get assigned position indices (and maps) as well.

In the context of convolution, a feature f is associated with the weight configuration a neuron can have. [3] This configuration preserves the relative positioning of the connection weights. The weight configuration of a neuron n encodes for the relative ordering of the neurons which are input of the connections to n. The weight of a connection $w_{i \multimap j}$ is assigned the position of i relative to the upper left input to j:

$$w_{i^f_{(x,y)} \multimap j^z_{(x_o,y_o)}} = w^z_{(x-x_m,y-y_m,f)} \tag{2.7}$$

where

- $x_m = \min \left\{ x_i \middle| i^z_{(x_i,y_i)} \multimap j^z_{(x_o,y_o)} \right\}$ and likewise y_m

See figure 2.4 for visual aid to this notational convention.

Similarly, we index the biases with the feature to which they belong: b^f.

2 For a colour image we can for example use the red, green and blue values or the hue, saturation and brightness values.

3 Remember that this includes the bias.

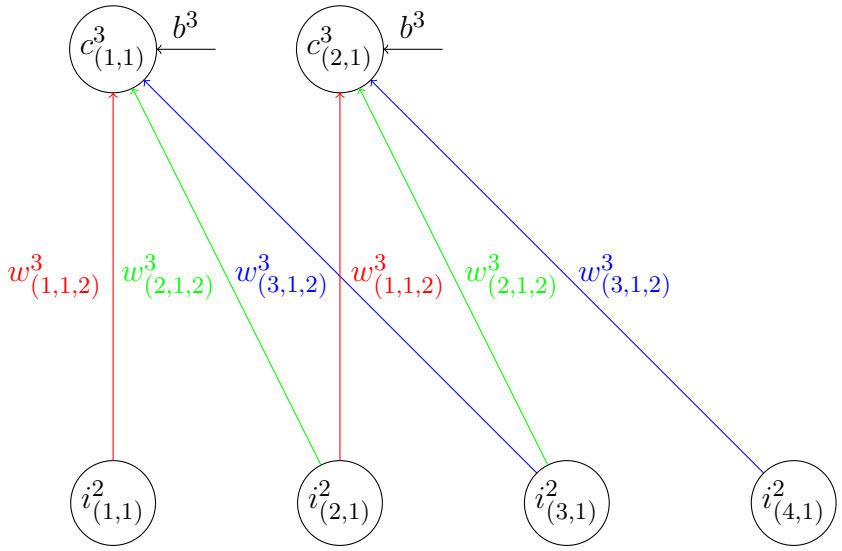


Figure 2.4: Schemata showing notation in a convolution layer for the top row of pixels in input map 2 and the corresponding weights of feature 3, giving two pixels in convolution map 3. Note that for example the connection from $i^2_{(1,1)}$ to $c^3_{(1,1)}$ has the same weight as the connection from $i^2_{(2,1)}$ to $c^3_{(2,1)}$, i.e. $w_{i^2_{(1,1)} \multimap c^3_{(1,1)}} = w_{i^2_{(2,1)} \multimap c^3_{(2,1)}} = w^3_{(1,1,2)}$ and that the location indices of the neurons in the convolution map are equal to their leftmost input, i.e $(1,1)$ and $(2,1)$. The colors indicate the weight within the weight configuration depicted.

The weight configuration f thus encodes for the weights of a neuron in abstraction of the specific neurons which connect with it. We can describe it mathematically as a tuple:

$$f = \langle b^f, \mathbf{w}^f \rangle \tag{2.8}$$

where

- b^f is the bias;
- $\mathbf{w}^f$ is a vector of weights.

We also assign a position to the output neurons of a layer. The neuron connected to the upper left neuron in the input maps gets assigned position $(0,0)$, the output neuron to its right gets assigned position $(1,0)$, etcetera.

WEIGHT SHARING Now that we have introduced the prerequisite notational machinery we can describe a convolution layer more thoroughly. Convolution consists of evaluating a feature f at regular intervals in the input map(s). For each place where we evaluate f the layer contains a unique neuron in map Z_f with the same weight configuration. This can be seen as an example of *weight sharing*; multiple neurons share the same weight configuration and thus their connections share weights.

COMMON PRACTICES Since each neuron in a convolution layer also gets assigned a position, we can order the output in maps as well. Each feature f generates a *feature map* Z_f containing neurons which evaluate f at different places. A following convolution layer then gets multiple input maps. It is common to have the same connectivity across multiple input maps; if a neuron has a connection from a neuron with location l in a feature map within the input layer, it also has connections from neurons with location l in all the other feature maps in that layer.

We call the local fields of neurons in a convolution layer *convolution fields*. It is common practice to have rectangular convolution fields. Each neuron gets input from each neuron in a rectangular sub-frame of the input maps. Quite often this rectangle is square. The size of this rectangle $(\mathrm{w_c} \times \mathrm{h_c})$ must be the same across all features in a given layer, so that the dimensions of the output maps are consistent.

As defined above, convolution is the process by which a feature is evaluated *at regular intervals* in the input to the layer. We can for example create a neuron every 3 pixels horizontally and every 2 pixels vertically with a given feature. However, it is common practice to set these two intervals to 1, so that we evaluate a feature at every position in the input maps.

The size of the output maps of a convolution layer depend on the size of the input maps (and the intervals at which we evaluate features). We generally start an output map with a neuron whose left- and uppermost input is the left- and uppermost neuron of an input map; the first neuron is connected to a sub-frame at the upper-left corner of the input maps. We then create new neurons at the given intervals, until we would create a neuron whose input frame would fall outside of the input map. For intervals of length 1, and an input map of dimensions $(\mathrm{w_i} \times \mathrm{h_i})$, this would amount to an output map of dimensions $(\mathrm{w_i} - \mathrm{w_c} \times \mathrm{h_i} - \mathrm{h_c})$.

It is thus abnormal to consider *out-of-map evaluation*, which would be the case when the output map would also contain neurons of which the input frame would fall partly outside the input maps. In figure 2.5, out-of-map evaluation is depicted in red. Convolution which makes use of out-of-map evaluation is called *full convolution*, while the other kind is called *valid convolution*.

We would like a CNN to be as good in recognizing a face which is partly in the image as it is in recognizing a face which is partly occluded by any object. In out-of-map evaluation the difficulty lies in the fact that we cannot apply the whole weight configuration to an out-of-map neuron, since some input neurons don't exist for the specified positions. We could disregard these weights and the nonexistent input neurons, but that could lead to biased conclusions. For example, a feature f which encodes for a contrast—namely one half of the frame containing a feature g while the other half doesn't—would fire when the edge of the input map contains feature g,

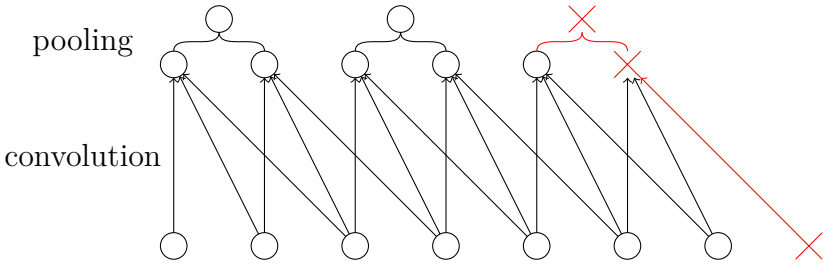


Figure 2.5: A schematic of a simplified CNN convolution and pooling layer, with one input layer and one feature. The red connections and nodes signify edge cases.

because disregarding neurons is equivalent to assuming their output is zero. We therefore disregard these edge cases.

MATHEMATICAL FORMALIZATION When formalizing a common CNN layer mathematically, we can make use of the fact that there is a one-to-one correspondence between a weight configuration f and the output map Z_f of all neurons with that weight configuration in the layer at hand. We number these such that the weight configuration is assigned the same number as the corresponding output map. We then introduce mathematical variables which are indexed by a natural number k instead of a weight configuration or a feature map.

Since a CNN is a feedforward neural network, we can derive a mathematical formula to compute the value of an output neuron given the weights and the inputs which is given by equation 2.9.

MATHEMATICAL CONVOLUTION It is the sum in the above formula which gives convolution layers and a CNNs their name. It can be seen as an instance of two dimensional *discrete mathematical convolution*.

One dimensional discrete convolution is defined by:

$$(f * g)[n] = \sum_{t \in D} f[n-t]g[t] \tag{2.10}$$

where

- D is the domain; $D = [-\infty, \infty]$ when functions f and g are defined everywhere.

It can be viewed as the process of flipping either function and sliding the resulting functions across each other; at each step t in this sliding process the output of the convolution is a weighted sum of the values of f with weights given by g (over all n where f and g are defined).

In our case the functions f and g are indexing functions, which are only defined for positive (or zero) indices within the dimensions of the multidimensional array of inputs and weights respectively. The two dimensional

$$c^k_{(x_\mathrm{p},y_\mathrm{p})} = \sigma\left(\sum_{(x,y,z)<(\mathrm{w_c},\mathrm{h_c},N)} \left\{ i^z_{(x_\mathrm{p}+x,y_\mathrm{p}+y)} w^k_{(x,y,z)} \right\} + b^k \right) \tag{2.9}$$

where

- i and c are neuron outputs representing input and output of the convolution layer;
- $i^z_{(x,y)}$ is the input of feature z on location (x, y);
- $c^k_{(x,y)}$ is the activation and output of neuron k on the location (x, y) in the pooling map, i.e. the output map of the convolution layer;
- N is the number of input maps, i.e. the number of input features;
- x_p and y_p are the coordinates of the output neuron in the pooling map;
- $\mathrm{w_c}$ and $\mathrm{h_c}$ are the width and height of the weight configuration of neuron k;
- $(x_1, x_2, \dots) < (\mathrm{w}_1, \mathrm{w}_2, \dots)$ is short for $0 \leq x_1 < \mathrm{w}_1 \wedge 0 \leq x_2 < \mathrm{w}_2 \wedge \dots$
- $w^k_{(x,y,z)}$ is the weight on the connection from feature z at location (x, y) to neuron k.

convolution employed by neuron k in a convolution layer is given by equation 2.11.

$$\sum_{z<N} f^z * g_z^k \tag{2.11}$$

where

$$\left(f^z * g_z^k\right)[x_\mathrm{p}, y_\mathrm{p}] = \sum_{(x,y)>(-\mathrm{w_c},-\mathrm{h_c})} f^z[x_\mathrm{p}-x, y_\mathrm{p}-y] g_z^k[x,y]$$

$$f^z[x,y] = i^z_{(x,y)}$$

$$g_z^k[x,y] = w^k_{(-x,-y,z)}$$

such that

$$\sum_{z<N}\left(f^z * g_z^k\right)[x_\mathrm{p}, y_\mathrm{p}] = \sum_{(x,y,z)<(\mathrm{w_c},\mathrm{h_c},N)} i^z_{(x_\mathrm{p}+x, y_\mathrm{p}+y)} w^k_{(x,y,z)} \tag{2.12}$$

Note that in the above equations, the domain $D = ([-\mathrm{w_c}, 0], [-\mathrm{h_c}, 0])$ is flipped, so that the coordinates of the convoluted output correspond to the coordinates of the left upper pixels in the convolution fields. Also the weights function g_z^k is flipped, so that the weights are oriented the same as the inputs. The eventual form fits exactly to equation 2.9.

CONVOLUTION AS MULTIPLE EVALUATION Another way to look at convolution is to view the reiterated application of a feature as multiple evaluations of the same neuron. Instead of having multiple neurons with the same weight configuration, we have one neuron which is evaluated at different positions. At every position where it is evaluated its output is recorded in the corresponding position in the feature map.

This way we can see a standard convolution layer as a reiterated application of an MLP layer. This MLP layer would contain as many features as the convolution layer contains feature maps and as many inputs as the weight configuration of a feature in the convolution layer has connection weights. We would then present a sub-frame of the original input as input of one iteration and set the outputs in the output feature map to what the MLP layer outputted.

This way of handling convolution effectively multiplies the number of input images and presents the MLP layer with smaller sub-images. The outputs of the network are then ordered into new grids. This paves the way for using objective functions which are defined on single MLP layer outputs instead of whole feature maps.

It also paves the way for introducing partial translation invariance by applying this mechanism to alternative learning methods and even to computational models totally different from ANNs. We could just apply any model to the sub-images and order the outputs of the model afterward.

TRANSLATION INVARIANCE Convolution contributes to *translation invariance*. A model is called translation invariant when its output is invariant under translations of its input. When we would shift the input image in a certain direction along the 2-dimensional grid, the output of the model would then still be the same. This means that such a model is blind for where in the input image certain objects or features occur.

The property is appropriate for models that are used for classification, since we generally want to apply a label to the whole image even though the reason for applying the label might lie in a small area within the image. Translation invariance is especially used in photos. The specific frame that is chosen to capture visual reality is arbitrary; the photographer might have just as well turned a bit to the right and then taken the photo, effectively performing a horizontal translation (or something close to such a transformation). In contrast, images of characters extracted from computer fonts are not so good a candidate for making use of translation invariance, since the character is placed in the middle of the image and take up almost the whole image.

A CNN is not translation invariant in itself; convolution only helps toward achieving translation invariance. The output of a CNN needs to be processed in a certain way to achieve it. Each output feature map Z still contains position information. We need to apply functions $s(\mathbf{z})$ to all outputs of neurons within each map Z which in a way *summarize* the information in the feature map without looking at the position information. The layer in which we apply such a function to each feature map is called a *total pooling layer*, in order to distinguish it from a normal pooling layer, which is discussed in section 2.2.2. There we also discuss some basic functions which we can use as pooling function.

Note that total translation invariance is infeasible, since we deal with images of finite dimensions, while translation invariance deals with translations of unbounded length. The *partial translation invariance* at hand is such that the output of a feature is invariant under translations of its input within the two-dimensional plane limited by the dimensions of the input map. Furthermore, in order for the pooled output of a translated image to be the same as the pooled output of the original image the output of the pooling function should not rely on the pixels which come to fall outside of the image. In the next section we covers typical pooling functions, some of which cannot be said to exhibit the above property.

2.2.2 *Pooling*

The above mentioned summarization functions are also commonly used in intermediate *pooling layers*, a.k.a. sub- or down-sampling layers, which are placed after/above convolution layers. See figure 2.3. In that context they are known as *pooling functions*. In a pooling layer, we don't summarize over

the whole feature map, however. Instead, we summarize over small areas, such that the output of the pooling layer is a condensed version of the input map. Pooling is done by subdividing all neurons in a map into a grid of small (usually 2×2) non-overlapping rectangles, called *pools*. For each pool, the pooling layer contains a neuron connected to the set of neurons contained within that pool, which we call the *pooling field*. The output of a neuron in the pooling layer is determined by applying the pooling function to the outputs of the pooling field. The output map then has a considerably smaller number of neurons, which is a substantial dimensionality reduction. This dimensionality reduction makes computation in subsequent layers less complex and therefore faster.

A pooling neuron is different from a 'normal' neuron, since it doesn't (necessarily) use the weighted sum of its inputs. We compute the output of a pooling neuron using the following formula. For any neuron j in a pooling layer, it holds that:

$$j = \sigma_j\left(a_j\right) \tag{2.13}$$

$$a_j = s_j(\mathbf{z}_j) \tag{2.14}$$

where

- a_j is the activation of the pooling neuron;
- $\mathbf{z}_j$ is the vector of neuron outputs of neurons within the pool of neurons which neuron j summarizes;
- $s_j(\cdot)$ is the pooling function used by the pooling neuron.

Here we introduce a seemingly superfluous transfer function σ_j over the activation a_j. Although this function is generally linear, we introduce it to conform more to the abstract characterization of a neuron given by formula 2.3. Below, in section 4.4.2, we consider restructuring a CNN by changing the transfer function of a pooling neuron.

Since we generally have rectangular pools of size $(\mathrm{w_p} \times \mathrm{h_p})$, we can declare that the pool of a pooling neuron j consists of a particular set of neurons:

$$\exists(u,v) : Z_j = \left\{ i^j_{(x,y)} \middle| (u,v) \leq (x,y) < (u,v) + (\mathrm{w_p}, \mathrm{h_p}) \right\} \tag{2.15}$$

Again we distribute functions over the constituents of tuples in order to make the formulas more readable. $(u,v) < (x,y)$ is short for $u < x \wedge v < y$ and $(u,v) + (x,y)$ is short for $(u+x, v+y)$.

By using the same notation for mathematical variables as defined above we can specify the workings of a pooling neuron more concretely. Again we assign the output neuron the same position as the position of the upper leftmost neuron in the pool. We can then specify a mathematical formula for determining the output of a neuron in a convolution layer; see formula 2.16.

$$p^k_{(x_\mathrm{f},y_\mathrm{f})} = \sigma^k\left(s^k\left(Z^k_{(x_\mathrm{f},y_\mathrm{f})}\right)\right) \tag{2.16}$$

$$Z^k_{(x_\mathrm{f},y_\mathrm{f})} = \left\{i^k_{(x_\mathrm{f}+x,y_\mathrm{f}+y)} \mid (x,y) < (\mathrm{w_p},\mathrm{h_p})\right\} \tag{2.17}$$

where

- i and p are neurons representing convoluted input and pooled output;
- x and y are natural numbers in the sense that $x \geq 0 \wedge y \geq 0$
- $i^z_{(x,y)}$ is the input of feature z on location (x,y), e.g., the output of a convolution layer;
- $p^k_{(x_\mathrm{f},y_\mathrm{f})}$ is the pooled output on the location $(x_\mathrm{f},y_\mathrm{f})$ in the output map of feature k of the pooling layer;
- x_f and y_f are the coordinates of the final output neuron in the output map;
- σ^k is the transfer function used in feature map k;
- s^k is the pooling function used in feature map k;
- $Z^k_{(x_\mathrm{f},y_\mathrm{f})}$ is the pool of neurons connected to the final output neuron;
- $\mathrm{w_p}$ and $\mathrm{h_p}$ are the width and height of the pool;

Deformation Invariance

Since the application of a pooling function to whole feature maps gave rise to partial translation invariance, we can expect that pooling layers also give rise to some form of invariance. It actually gives rise to a specific form of *deformation invariance*. It allows for a flexibility in the relative positioning of features with respect to each other. The input features for the subsequent pooling layer have obtained partial translation invariance within the small rectangular frames, but each input feature might have been translated in a different direction. We can thus say that a pooling layer causes a consecutive convolution layer to be invariant under deformations which consist of small translations of constituent parts in the input.

These deformations can also consist of every input feature being translated in the same direction and so we also gain a small partial translation invariance over the whole image. Furthermore, the deformation can consist of input features being translated toward or away from some point, constituting a small scaling invariance.

Pooling functions

The invariances listed above remain valid irrespective of what pooling function is used—as long as that function doesn't use the relative positioning of its inputs. The choice of pooling function depends on a couple of factors; each of the following options has its pros and cons. Of the following pooling functions the first is the most common, the second is quite common, while the others or pooling functions not generally found in literature. These pooling functions are introduced here as alternative to the typical pooling function which is asymmetric and exhibits discontinuities.

MAXIMUM The most common pooling function is $s(\mathbf{z}_j) = \max Z_j$, where Z_j is the set of all elements in the pool. The maximum output of all neurons in a pool is then propagated forward.

Though this may seem as the way to go, there are some serious pitfalls to using this as the (only) pooling function. One likes to think of a neuron in a way that when it fires, it tells you that a feature is present at a specific location in the input and when it doesn't fire it doesn't tell you of the presence of any feature. However, the hyperbolic tangent transfer function is perfectly symmetric (see section 2.1.3) and so firing positively can provide as much information as firing negatively. It is a mistake to think the max function neatly captures the information from the pool. [4]

Consider a pool with neurons which have outputted values $\{-1, 0, 0, 0\}$. In that case the max function would output 0 instead of -1, while an output of -1 is most informative of the input; indeed it is even equivalent to a case

4 The same should hold in case the logistic sigmoid function is used, since the two are equivalent under different weights.

with an output of 1 where the feature has all its weights negated. An output of -1 can be interpreted as the presence of the inverted feature. We might therefore just as well use min as the pooling function.

A solution to the problem might lie in the simple act of using both min and max. From each feature map we might then abstract two feature maps of reduced size, one obtained by pooling with the min pooling function and the other by max. A small issue with this solution is that the resulting down-sampling does half the work, since we have twice as many output neurons for the same feature map.

AVERAGE Another approach is to take the average over the pool as the summarized feature output: $s(\mathbf{z}_j) = \overline{Z_j} = \frac{1}{|Z_j|} \sum_{z \in Z_j} z$. This could bypass the problems above; in the above example, with pool outputs $\{-1, 0, 0, 0\}$, the summarized output would be -0.25, which is still toward the negative, which thus captures the presence of the inverted feature.

However, $\overline{Z_j}$ is unable to distinguish between situations of no information and situations of equal opposite information. It is unable to distinguish between pool outputs of $\{0, 0, 0, 0\}$ and $\{-0.9, 0.3, 0.3, 0.3\}$, while the latter contains a lot more information, which isn't passed on by $\overline{Z_j}$.

ABSOLUTE MAXIMUM The last approach discussed here is to use the absolute maximum; the pooling function then outputs the value which was greatest in absolute sense:

$$s(\mathbf{z}_j) = \arg\max_{z \in Z_j} (\operatorname{abs} z) \tag{2.18}$$

$$= \arg\max_{z \in Z_j} \left(z^2\right) \tag{2.19}$$

where

- $\arg\max_{i \in S} (f(i))$ returns the element i in set S, for which $f(i)$ produces the largest result.

When pool outputs are given by $\{-0.9, 0.3, 0.3, 0.3\}$, the summarized value would be -0.9, which seems to convey the most important information from the pool. Note that when the pool has two absolute maxima equidistant from the origin, but differing in sign, the output of the pooling function is underdetermined: it might output the negative absolute maximum just as well as the positive one.

We can use the $\arg\max$ function also in defining maximum pooling: $\max Z_j$ can be rewritten to $\arg\max_{z \in Z_j} z$.

SOFT APPROXIMATIONS The use of the max or $\arg\max$ function by the above pooling functions might be seen as unsophisticated or improper

when performing gradient ascent or similar learning mechanisms.[5] When calculating derivatives a 'hard' max function only propagates derivatives back to its maximum; suppose that $\arg\max_{z \in Z_j} f(z) = z_1$, it then holds that:

$$\frac{\partial}{\partial z_1} \arg\max_{z \in Z_j} f(z) = 1 \tag{2.20}$$

$$\forall x \in Z_j, x \neq z_1 : \frac{\partial}{\partial x} \arg\max_{z \in Z_j} f(z) = 0 \tag{2.21}$$

Although this decreases computational complexity of the work to be done in backpropagation, it introduces *jump discontinuities* in the function surface. We could be at a point in the objective function surface where there is a huge cliff right next to us, while the derivative doesn't lead us toward that high region; the derivative then wouldn't be a good indicator of the direction in which the function surface climbs most.

It is therefore a good idea to use a soft approximation to the hard arg max function. We do this by making use of one-of-K coding and introducing a formula related to the softmax activation function, which is given by:[6]

$$S(\{a_n | n \in L_\mathrm{O}\}, a_m) = \frac{c_m}{t} \tag{2.22}$$

$$c_n = e^{a_n} \tag{2.23}$$

$$t = \sum_{j}^{L_\mathrm{O}} c_j \tag{2.24}$$

where

- c_n is the contribution of neuron n to the softmax function;
- t is the total of all contributions.

We can view $\max Z_j$ as the dot product of the vector $\mathbf{z}$ of all elements of Z_j and a one-of-K coded vector $\mathbf{m}$, where only the element corresponding to the maximum is one:

$$\max Z_j = \mathbf{z}_j^\mathsf{T} \mathbf{m} \tag{2.25}$$

In order to get the soft version of the max function we replace vector $\mathbf{m}$ by a vector which is formed by performing the softmax activation function on the input vector.

5 These learning mechanisms are explained in section 2.3.2.
6 Section 2.3.2 covers the softmax activation function more elaborately.

$$\text{soft}\max Z_j = \mathbf{z}_j^\mathsf{T}\mathbf{m} \tag{2.26}$$

$$m_i = S(Z_j, z_{ji}) \tag{2.27}$$

where

- S is the softmax activation function as given by formula 2.22;
- $\mathbf{m}$ is a vector following the same ordering as $\mathbf{z}_j$.

We can expand this function and modify it in order to give a soft approximation to the arg max function:

$$\text{soft}\underset{z\in Z_j}{\arg\max}\, f(z) = \mathbf{z}_j^\mathsf{T}\mathbf{m}$$

$$m_k = \frac{e^{pf(z_k)}}{\sum\limits_{z_l\in Z_j} e^{pf(z_l)}} \tag{2.28}$$

where

- p is a hyperparameter controlling the hardness of the approximation

The derivative of this function, and its derivation, are given in appendix C.5.

This function can actually be used as an approximation to a couple of functions, depending on the value of the hyperparameter p. In the context of an ANN, where the hyperbolic tangent transfer function is used, outputs are scattered across the interval $(-1, 1)$ and pools consist of about 10^1 neurons, a setting of $p > 10^1$ is close to the arg max function. For $p < -10^1$, it is close to the arg min function. For $p = 0$, the formula is equal to the average $\overline{Z_j}$:

$$\begin{aligned}\frac{1}{\sum\limits_{z_k\in Z_j} e^{pf(z_k)}} \sum_{z_k\in Z_j} z_k e^{pf(z_k)} &= \frac{1}{\sum\limits_{z_k\in Z_j} e^0} \sum_{z_k\in Z_j} z_k e^0 \\ &= \frac{1}{\sum\limits_{z_k\in Z_j} 1} \sum_{z_k\in Z_j} z_k 1 = \frac{1}{|Z_j|} \sum_{z_k\in Z} z_k\end{aligned} \tag{2.29}$$

2.3 TRAINING ARTIFICIAL NEURAL NETWORKS

Above we have described the general form common to several types of ANN. We have not yet described ways to determine the parameters. In what way do we determine what the values of the weights and biases should be?

In order to assess the desirability of a given parameter setting, we should estimate its effectiveness toward some given goal. We might supply different goals, depending on circumstances. Two important types of goal are given by *supervised* and *unsupervised learning*.

Supervised learning tries to minimize the difference between the actual output of an ANN and what is known to be the correct output for a given data point. Each point in the data set is labeled with the values of some variables which the network should reproduce. These labels, called *teacher signals*, are then used by the learning method to modify the network parameters in order to make the network better in reproducing the labels.

Unsupervised learning merely tries to extract useful features from the input, without knowing their eventual purpose. The features extracted should represent essential properties of the input. An unsupervised learning method might for example try to extract features which best disambiguate between the different images, or it might try to extract features which work best in reproducing the input itself. Another example would be a learning method which tries to find aberrant or anomalous features.

2.3.1 *Objective Functions*

Most learning algorithms try to optimize a goal of the network which can be described in terms of a function $E(L_{\mathrm{O}})$ over all outputs $m \in L_{\mathrm{O}}$. [7] In supervised learning we typically want to *minimize* an error function, which is defined in terms of the actual output of the network in response to a data point and the desired output, over all data points. In unsupervised or semi-supervised learning we may want to *maximize* some objective function over the outputs. From here on we suppose the function is defined in *positive terms*, e.g., we negate the error function and maximize the resulting objective function, so that a higher function value can always be interpreted as better. [8]

Such optimization is not an easy task. The function surface of a feedforward ANN is typically highly irregular and so the function surface of the objective function is as well. It is often not feasible to try and find the global optimum, because an ANN typically has too many parameters to efficiently explore the whole landscape of its objective surface. We therefore stop exploring when we have found a local optimum. After a couple of random restarts of the whole network (with different parameter initializations) and finding local optima, we choose the network with the best local optimum as the end-result of the learning algorithm.

7 The 'O' in L_{O} is the letter 'O' of 'output', not the digit '0'.

8 In order to conform to existing literature [8, p. 233], we have chosen to refer to the function as $E(\cdot)$, which was chosen as abbreviation of 'error function'. Furthermore, the letter 'O' is already in use as abbreviation for 'output'.

2.3.2 *Weight Updates*

Exploration of the objective surface is done in an iterative manner. Most learning algorithms take steps in an appropriate direction and see what the objective surface looks like there in order to assess what further step to take. The direction and size of the step is determined by what we call the *update rule*.

Gradient Ascent

One of the simplest learning algorithms for feedforward neural networks is *gradient ascent*. Each iteration it takes a step in the direction of steepest ascent over the function surface. That is to say: it follows the derivative of the objective function; more specifically, for each parameter we take a step proportional to the partial derivative of the objective function with respect to that parameter.

In the deduction of the partial derivative w.r.t. lower layer parameters we make extensive use of the chain rule for differentiation:

$$\frac{\partial f(g_1(x),g_2(x),\ldots,g_k(x))}{\partial x} = \frac{\partial f(p)}{\partial g_1(x)}\frac{\partial g_1(x)}{\partial x} + \frac{\partial f(p)}{\partial g_2(x)}\frac{\partial g_2(x)}{\partial x} + \cdots + \frac{\partial f(p)}{\partial g_k(x)}\frac{\partial g_k(x)}{\partial x} \tag{2.30}$$

For the top layer, computing the partial derivatives w.r.t. the parameters is quite straightforward. The partial derivatives are given by:

$$\frac{\partial E(L_O)}{\partial b_n} = \sum_m^{L_O} \frac{\partial E(L_O)}{\partial m}\left(\frac{\partial m}{\partial a_n}\frac{\partial a_n}{\partial b_n}\right) = \frac{\partial E(L_O)}{\partial a_n} = \delta_n \tag{2.31}$$

$$\frac{\partial E(L_O)}{\partial w_{k\multimap n}} = \sum_m^{L_O} \frac{\partial E(L_O)}{\partial m}\left(\frac{\partial m}{\partial a_n}\frac{\partial a_n}{\partial w_{k\multimap n}}\right) = \frac{\partial E(L_O)}{\partial a_n} k = \delta_n k \tag{2.32}$$

where

- $\delta_x = \dfrac{\partial E(L_O)}{\partial a_x}$ denotes something we call the *local objective* at neuron x, or *local error* in the case of a negatively defined objective function.

In order to compute the local objective of a hidden neuron it suffices to look only at the local objectives of the neurons it connects with:

$$\text{for } k \notin L_O : \delta_k = \frac{\partial E(L_O)}{\partial a_k} = \sum_{n:k\multimap n} \frac{\partial E(L_O)}{\partial n}\frac{\partial n}{\partial a_k} = \sum_{n:k\multimap n} \delta_n w_{k\multimap n} \tag{2.33}$$

From the local objective values which we now can compute at each neuron, we can compute the partial derivative of the objective function with respect to the biases and weight using formulas 2.31 and 2.32.

One might think that weight sharing would be problematic in calculating the derivatives, because the derivative $\frac{\partial E(L_{\mathrm{O}})}{\partial b_n}$ depends not only on n. However, we can just sum them up by making use of the chain rule of multivariate calculus. For example, when neuron n and m share bias $b_n = b_m$ (and that bias is only used by those two neurons) we get $\frac{\partial E(L_{\mathrm{O}})}{\partial b_n} = \delta_n + \delta_m$.

Note that before we can compute the local objective at a certain hidden neuron we need to have computed all local objective values at neurons with which it is connected. We therefore need to start computing local objectives at the output and from there move back toward the input while computing local objectives from previous local objectives and the weights in between. This is why the technique of calculating the partial derivatives in such a way is known as *backpropagation*—the local objectives are propagated backward and accumulated in the local objectives of the lower neuron. [9]

Now that we know how to compute the partial derivatives of the objective w.r.t. the parameters, the update rule is fairly simple:

$$p \mapsto p + \Delta p \tag{2.34}$$

$$\Delta p = \eta \delta_p = \eta \frac{\partial E(L_{\mathrm{O}})}{\partial p} \tag{2.35}$$

where

- p is either a weight parameter from $w.$ or a bias parameter from $b.$;
- η is a hyperparameter for the size of the step taken in the given direction.

This concludes the simple gradient ascent learning algorithm. Many other learning algorithms also depend on the partial derivatives in the parameters. An example of a learning algorithm similar to gradient ascent is called '*momentum*'. In this algorithm each parameter is assigned a velocity, which decreases every step due to friction, but is increased by the current partial derivative of the objective w.r.t. that parameter:

$$\Delta p = v^{\tau+1} = \alpha v^{\tau} + \eta \delta_p \tag{2.36}$$

where

- v is the velocity built up;
- $0 \leq \alpha \leq 1$ is a hyperparameter that governs the momentum and so the friction.

9 Sometimes the term 'backpropagation' is used to denote gradient ascent itself, though this confuses terminology.

At every weight update a fraction of the velocity from the previous update time step remains, constituting momentum. When we formulate the objective negatively, the function surface is inverted and so we are looking for a minimum instead of a maximum. In such a context the momentum method can be viewed as modelling a ball rolling around on the function surface. It can then surpass a shallow local minimum due to the velocity it had built up and end up in a lower local minimum.

2.3.3 *Classification*

Let us consider a typical type of supervised learning scenario. Suppose the teacher signals of the data set consist of a class; each data point is labelled with one of K classes to which it is known to belong. *Classification* is the goal for an ANN to correctly classify the data points.

SOFTMAX ACTIVATION FUNCTION In order to abstract away from the specific classes assigned to the data, we can represent a specific class label using *one-of-K coding*. In such a coding scheme each class is first assigned a number $1 \leq k \leq K$. The coding scheme converts a class label which is assigned number k to a vector $\mathbf{d}$ of length K for which $d_k = 1$ and zero elsewhere. For classification, we assign each output neuron a variable in the desired vector $\mathbf{d}$.

The output neurons of the ANN can reflect the structure of such a one-of-K label type. We do this by supplying an output function which respects certain properties one-of-K coding has: just like a one-of-K vector, the output neurons outputs must add up to 1[14, Sec. 11.3]. The output function typically used to ensure this property is called the *softmax activation function* S and is given by:

$$S(\{a_n | n \in L_{\mathrm{O}}\}, a_m) = \frac{c_m}{t} \tag{2.37}$$

$$c_n = e^{a_n} \tag{2.38}$$

$$t = \sum_{j}^{L_{\mathrm{O}}} c_j \tag{2.39}$$

where

- c_n is the contribution of neuron n to the softmax function;
- t is the total of all contributions.

CROSS ENTROPY The error function which is minimized in such a case is the *cross entropy* between the one-of-K coding of the true class label, $\mathbf{t}$,

and the actual outputs of the network which are used to predict the class: $\mathbf{p}$. Cross entropy is defined by:

$$H(\mathbf{t}, \mathbf{p}) = -\sum_{k=1}^{K} t_k \log p_k \tag{2.40}$$

We use the negation of the cross entropy as objective function to be maximized. Combined with a softmax activation function, this leads to a local error given by $\delta_n = t_k - p_k$. [10] In appendix C.3 we derive the derivative of the softmax activation function w.r.t. its inputs; the resulting derivative is then used to prove the formula for the local error of cross entropy when using a softmax activation function in appendix C.4.

2.3.4 *Weights*

This section covers weight initialization and regularization. First we discuss how to initialize the weights of an ANN, next we discuss why large weights form a problem and finally how that problem might be solved by augmenting the objective function with regularization terms.

INITIALIZATION Before we start training an ANN, we need to initialize the parameters of the model. It is customary to initialize the weights randomly around zero. We might take each starting value from a Gaussian distribution with zero mean and appropriate standard deviation. We should be careful not to take initial parameters from a Gaussian distribution with too large standard deviation, since large weights are problematic, as is discussed below.

Another possibility is to take the initial values from a uniform distribution within some appropriate bounds. That way we ensure that no weight is initialized with a value which is too large.

The appropriate bounds are generally chosen to coincide with how we expect the parameters to be distributed for a maximum in the objective function surface. From this starting point the learning mechanism then typically converges to the closest local maximum. Because the objective function surface may contain many local maxima, we may rerun the initialization and training phase in order to find multiple local maxima, from which we then pick the best. These reruns are commonly referred to as *random restarts*.

However, in chapter 4 we consider an objective function for which we know the maximum closest to the origin to be the preferred maximum. We therefore consider initializing weights very close to the origin.

Note that biases serve a different role than connection weights. While biases cause the output to be shifted toward one of the extremities of the output space, the contribution of weights to the output depends on the

10 Note that when we would minimize the objective, the local derivative would be given by $\delta_n = p_k - t_k$.

input. We can therefore choose to initialize the biases differently from the weights. We consider initializing the biases to zero in chapter 4.

The problem of large weights

In some cases a learning method might produce very large weights, which can be viewed as an undesirable characteristic. Therefore we take measures to prevent large weights.

There are several reasons why large weights might be a bad thing. One reason is that learning too large weights amounts to *overfitting*. With large weights, a small difference in the input might result in a big difference in the output of the network. While under the given objective the network might have performed optimally for the training data set, its *generalization* power might actually be lower than a similar neural network with smaller weights. The network would then be tuned too much to the specificities of the training data.

Another reason is that large weights cause rigidity in the learning phase. When the weights on a connection toward a neuron are large, chances are that the activation of that neuron is also (absolutely) large. The derivative of both the logistic sigmoid function and the hyperbolic transfer function is very small in such a case. This means that in learning methods similar to gradient ascent, it would take quite a while to move away from the area in which the function surface is very shallow. Large weights can therefore make the network too rigid.

Regularization

In order to solve the problem of large weights we can add a *regularization term* R to the objective function $E(L_O)$. The objective function then becomes $\tilde{E}(L_O) = E(L_O) + R$, where R is a function not just over the outputs of the network. We can still use the technique of backpropagation to determine the partial derivatives of the first part of the evaluation function. The partial derivatives for the regularization term can then just be added to the partial derivatives computed in order to form the partial derivatives of the whole evaluation function.

One method to regulate the weights using a regularization term is called '*weight decay*'. It is given by the following simple formula:

$$R = -\lambda \frac{1}{2} \sum_{v \in W} v^2 \tag{2.41}$$

where

- W is the set of all weight variables;
- λ is a hyperparameter governing the force by which to keep the weights low.

The partial derivative of this function with respect to a weight w is then given by $\frac{\partial R}{\partial w} = -\lambda w$ Generally λ takes on small values such that generally $R < E(L_{\mathrm{O}})$.

The addition of the weight decay regularization term can be seen as overlaying the function surface of the objective function with a mountain with its top at the origin. This causes gradient ascent not to wander too far from the origin. The λ hyperparameter then governs the height of the mountain and as such the impact of the regularization term on the function surface of $\tilde{E}(L_{\mathrm{O}})$. For large λ we would get a function surface which is dominated by the mountain, which would cause a local optimum at the origin. This would mean that learning causes the network to converge to all zero weights. It is important to finely tune the λ hyperparameter.

The change in the function surface is an important observation. No longer would we find local optima of our objective function. Instead we generally find local optima which are only slightly shifted toward the origin compared to local optima of the objective function.

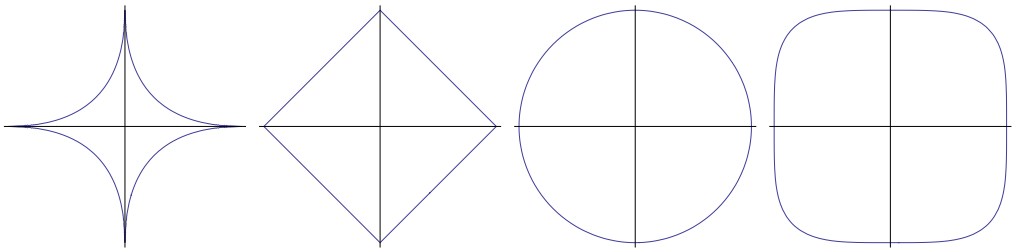
However, the problem of ever climbing ridges may persist and we might be drawn too much toward the origin. These two problems come down to the problem that λ is either too high, or too low. When it is too high we get the problem described above that the only parameter setting we get is one with all zero weights—the origin. The objective function surface might be such that there is a ridge which climbs with a rate higher than $\sum v^2$; when $\lambda < 2$, the function surface then still contains ever climbing ridges.

Another problematic case is when the objective function surface contains a large shallow valley in the area where the weights can be initialized. In order not to converge to the origin, we then have to make λ very small, but then weight decay ceases to have its positive effects.

Weight decay can be generalized to regularization terms following the following form:

$$R = -\lambda \frac{1}{2} \sum_{v \in \mathrm{im}(w)} |v|^q \tag{2.42}$$

The q hyperparameter gives control of how we valuate certain weight combinations. See figure 2.6 for visualizations of different settings of q. The contours in the plot give all points which get the same regularization value.



(a) $q = 0.5$ (b) $q = 1$ (c) $q = 2$ (d) $q = 4$

Figure 2.6: Contour plots for generalized weight decay as given by formula 2.42 for different settings of hyperparameter q. The axes are weight parameters.

For $q = 2$ we value all weight settings equidistant from the origin the same, while for $q < 2$ we promote smaller weights for a given weight setting. For $q > 2$ we promote weight settings under which all weights are large, compared to setting where some are smaller. [8, Ch. 3.1.4]

Often a value $0 < q < 2$ is chosen, so that small weights are promoted. Weights belonging to connections, as opposed to biases, which are very small can be neglected without much loss of functionality of the neural network. We can then remove these connections, so that there is less computation in evaluating the network.

Weight Restrictions

Another way to handle the problem of large weights is to introduce weight restrictions. A simple way to restrict the weights is to restrict each weight to an interval, e.g., $[-2, 2]$. Whenever a weight falls outside the interval after a weight update, we just put it back at the closest value which does lie inside the interval.

In certain situations, depending on the specifics of the function surface, we might end up with a lot of weights being either -2 or 2. The intervals constitute a hypercube in the parameter space to which we limit all parameter settings. The corners of this cube have the largest distance to the middle which in this example is the origin. The vector of all weights in this setting is thus longer than it would be to any of the sides of the hypercube. It can therefore cause higher activations in the neuron given the right configuration of input.

It might be a better idea to limit all weights to a hypersphere instead of a hypercube. When the distance of the vector of all weights becomes greater than a certain limit, we rescale the vector, and thus all weights, such that the distance is equal to the limit. We can describe this restriction in formula form as such:

$$\sum_{v \in W} v^2 < l \tag{2.43}$$

where

- l is the hyperparameter for the limit.

We can then use the same transformation as the one from formula 2.41 to formula 2.42 in order to generalize the weight restriction. We can actually view the plots in figure 2.6 as plots of the limit.

Other variations of this kind of weight restriction apply the principles above to all weights concerned with each neuron j. We then limit all weights $w_{i \multimap j}$ and the bias b_j within a hypersphere (or other n-dimensional figure) of a given radius l, for each neuron j.

2.3.5 *Batch Learning*

In the foregoing we've described how the signals of an ANN are determined by feeding forward the signals from the input through the network. We then went on to describe how to update the weights of the ANN in order to accommodate some objective defined on the output layer.

A simple learning mechanism is given by *online learning*. We then update the weights for every data point presented to the network. A single iteration in the learning process consists of first setting the input layer neurons to have the values of the data point, then feeding the signals forward through the network and finally updating the weights according to some update rule. As such we consider one data point at a time, which means we can use the time index τ to indicate the sample as well as the iteration in the learning process. [11]

Another learning mechanism is given by *batch learning*. In such a learning scheme the network is updated after having processed multiple data points. We accumulate statistics and use them for a single weight update step. For gradient ascent, for example, we compute the partial derivatives of the objective function w.r.t. the weights on each data sample, and update the weights proportional to the sum of these derivatives. In fact, we can easily transform any online learning procedure to a batch learning procedure. We simply sum the updates the online procedure would result in and use that to update the weights in a single update per iteration in the batch learning procedure.

We now have to introduce a parameter n in order to differentiate between the different data points, other than τ. Using this parameter, we can redefine the objective function such that its argument consists of the outputs of the network for several data points. The argument of objective function E

11 In the above, the time parameter has largely been left implicit. Instead we relied on the update operator '$\mapsto$', which implicitly has a time component.

implicitly referred to the network output for a single data point n: $E\left(L_{\mathrm{O}}\right)$ was short for $E\left(L_{\mathrm{O}}^{n}\right)$. We can write the objective function for batches as: $E\left(\mathbf{O}^{\tau}\right)$, where $\mathbf{O}^{\tau}$ is a matrix where each row is the vector of outputs $\mathbf{o}^{n}$ of the network in response to data point n at iteration τ.

We can convert any objective function for online learning to one for batch learning:

$$E\left(\mathbf{O}^{\tau}\right)=\sum_{n\in B^{\tau}}E\left(L_{\mathrm{O}}^{n}\right) \tag{2.44}$$

where

- B^{τ} is the set of data point indices for iteration τ.

Note that in the following we leave the time parameter τ implicit again, for readability of the formulas.

However, some batch learning procedures cannot be converted to online learning procedures. A batch learning procedure might compare the output of the network for different data points. We might take the variance of each output neuron as objective, which causes the derivatives of the objective function w.r.t. each output data point to depend on the network output for other data. In this example it might be a good idea to make the batch size equal to the size of the whole data set, so that we process the whole data set before updating the network parameters. Such objectives are considered in chapter 4.

3

UNSUPERVISED LEARNING

In this chapter we consider some techniques which are strongly related to the techniques we introduce in chapter 4. The first section covers Principal Component Analysis (PCA), and analytical techniques for performing the analysis. We show how the technique can be used to determine the weights of an Artificial Neural Network (ANN).

The next section covers the Generalized Hebbian Algorithm (GHA), which is an unsupervised learning technique which is shown to cause the weight vectors of an Multilayer Perceptron (MLP) converge to the principal components of the input data. It is therefore closely related to PCA. In the next chapter we introduce a method for Spread Maximization (SM) which makes use of a technique which can be seen as an extension of the GHA and therefore can be seen as extending PCA.

The last section of this chapter covers Auto-Encoders (AEs). Similar to the application of the extended GHA (which we introduce in the next chapter) to Convolutional Neural Networks (CNNs), a technique has been proposed which applies the AE principle to CNNs in much the same way. We describe a model proposed by Bourlard and Kamp [9], which we call a Pooling Convolutional Auto-Encoder (PCAE).

3.1 PRINCIPAL COMPONENT ANALYSIS

Principal Component Analysis (PCA) a.k.a. the Karhunen-Loève transform is the projection of data onto a space with a new coordinate system, called the *eigenspace*, which might be of a lower dimensionality[30].[1] The new coordinate system forms the basis of what is called the *principal subspace*. The axes of this coordinate system are called the *principal axes*. The positioning of the coordinate system of the eigenspace within the original space is determined by an orthonormal set of vectors called the *loadings* or *loading vectors*. The *principal components*, then, are the data points projected onto the principal axes; the first principal component is the data projected on the first principal axis.[2]

The new coordinate system is one that fits to the data in some way. The data generally determine uniquely what space to project on.[3] The principal

1 Here we mean an orthogonal Euclidean coordinate system.

2 Sometimes the principal axes are referred to in literature as 'principal components', though other times the projected data is denoted by that term.

3 Rotational symmetries of the data are reflected by an underdetermination in the projection.

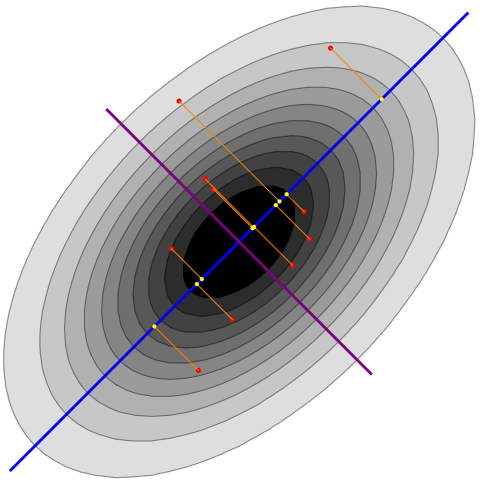


Figure 3.1: A graph containing a contour plot of a multivariate normal distribution in gray ($\mathbf{\Sigma} = \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix}$), its principal axes in blue and purple, points sampled from the distribution in red and the distance of those points from their projections (in yellow) in the principal subspace given by the first loading vector in orange.

axes are ordered on how well they fit the data; the first principal component is such that the corresponding principal axis fits the data best. The coordinate system of the principal subspace can be viewed as fitting to the data in two ways.

First, the transformation is such that the variance of the transformed data is maximized; the variance of the first dimension of the transformed data is maximal, and for every consecutive dimension, the variance is maximal within the subspace of dimension vectors orthogonal to all previous dimension vectors. In figure 3.1 the variance along the blue line is maximal; the direction of the blue line characterizes the direction of the first principal component.[8, Sec. 12.1.1]

Second, when transforming to a space of lesser dimensionality, PCA can be seen as a method to find the subspace such that a projection of the data into that subspace minimizes the total squared distance between the data and the projected data. In figure 3.1 mapping the data onto the blue line minimizes the mapping distance.[8, Sec. 12.1.2]

There are many ways to perform PCA; some are analytical and some are iterative. Here we focus on an analytical method. In section 3.2.3 we consider an iterative procedure which turns out to perform PCA as well. Although we can analytically derive formulas for the loadings, an iterative procedure to find the first N principal components of M-dimensional data can be less time-consuming when $N \ll M$.

3.1.1 *PCA by Eigenanalysis*

In this section we show how to perform PCA by doing an eigenanalysis on the covariance matrix of the data. We show that the loadings are given by

eigenvectors of the covariance matrix and that their variances are given by the corresponding eigenvalues.

Let's first formalize the concepts introduced in the above paragraphs. For principal component λ_k corresponding to eigenvector $\mathbf{W}_{\cdot k}$, the projected data is given by:

$$\mathbf{Z}_{\cdot k} = \mathbf{X}\mathbf{W}_{\cdot k} \tag{3.1}$$

where

- $\mathbf{Z}_{\cdot k}$ is the k^{th} principal component—the data projected on the k^{th} principal axis;
- $\mathbf{W}_{\cdot k}$ is the k^{th} loading;
- $\mathbf{X}$ is the input data, where the rows corresponds to the data points.

The variance of the projected data is given by the projected variance:

$$\begin{aligned}\operatorname{Var}\left[\mathbf{Z}_{\cdot k}\right] &= \mathbb{E}\left[\mathbf{Z}_{\cdot k}^2\right] - \mathbb{E}\left[\mathbf{Z}_{\cdot k}\right]^2 \\ &= \mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k}\end{aligned} \tag{3.2}$$

This result is well known. See appendix C.7 for a full proof.

We want to optimize the variance in the direction given by a principal axis. As we have already said, the principal axes are associated with loading vectors which are defined as unit vectors. For the first principal component, we therefore maximize $\operatorname{Var}\left[\mathbf{Z}_{\cdot 1}\right]$ under the constraint that $\|\mathbf{W}_{\cdot 1}\| = \|\mathbf{W}_{\cdot 1}\|^2 = 1$. [4] We do this by introducing a Lagrange multiplier λ_1, and making an unconstrained maximization of:

$$\mathbf{W}_{\cdot 1}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot 1} + \lambda_1\left(1 - \mathbf{W}_{\cdot 1}^{\mathsf{T}}\mathbf{W}_{\cdot 1}\right) \tag{3.3}$$

We see from appendix C.9 that this results in the fact that λ_1 is an eigenvalue of the covariance matrix with $\mathbf{W}_{\cdot 1}$ as eigenvector:

$$\boldsymbol{\Sigma}\mathbf{W}_{\cdot 1} = \lambda_1\mathbf{W}_{\cdot 1} \tag{3.4}$$

Left-multiplying both sides by $\mathbf{W}_{\cdot 1}^{\mathsf{T}}$, we get:

$$\mathbf{W}_{\cdot 1}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot 1} = \mathbf{W}_{\cdot 1}^{\mathsf{T}}\lambda_1\mathbf{W}_{\cdot 1} = \lambda_1 \tag{3.5}$$

Here we have used the fact that $\|\mathbf{W}_{\cdot 1}\| = 1$. Thus the variance of the projected data is maximized when the loading vector is equal to an eigenvector; the variance then has the size of the eigenvalue corresponding to that loading vector.

4 Instead of constraining the length of the weight vector, we constrain the square length of the weight vector, which is equivalent. That way we reduce the complexity of the formulas to come, since we eliminate the square root in the formula for vector length.

For further loading vectors, we need to constrain the loading vectors to be orthogonal to all previous principal axes; it is a necessary condition of the axes of a coordinate system to be orthogonal. We apply the same techniques as before and maximize:

$$\mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} + \lambda_k\left(1 - \mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot k}\right) + \sum_{j=1}^{K}\lambda_{kj}\left(\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot j}\right) \tag{3.6}$$

This leads us to conclude (see appendix C.10) that also further loading vectors are given by eigenvectors:

$$\mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} = \lambda_k \tag{3.7}$$

Again λ_k is equal to the eigenvalue corresponding to the eigenvector $\mathbf{W}_{\cdot k}$.

In appendix C.8 we show that the covariance of two different principal components is always zero. We therefore say that PCA performs decorrrelation; the transformed data have zero covariance and so also zero correlation. [5] The covariance matrix of the transformed data is therefore given by:

$$\boldsymbol{\Sigma}_{\mathbf{Z}} = \Lambda = \operatorname{diag}(\boldsymbol{\lambda}) \tag{3.8}$$

where

- Λ a diagonal matrix containing the eigenvalues;
- $\operatorname{diag}(\mathbf{v})$ denotes a diagonal matrix of elements of $\mathbf{v}$ in the diagonal;
- $\boldsymbol{\lambda}$ is a vector of the eigenvalues of $\mathbf{Z}$.

3.1.2 *PCA and Neural Networks*

Principal Component Analysis (PCA) can be seen as an unsupervised learning technique for improper neural networks. When we have a single layer ANN with linear transfer function, the network reduces to taking linear combinations of the input—the outputs are just the weighted inputs. This reducibility means it is not a proper ANN. [6] Nevertheless, we can perform PCA in order to train a layer.

It would not make sense to stack PCA layers to form a deep neural network. We have already seen in section 2.1.3 that stacking layers with a linear transfer function causes neurons to be redundant. Moreover, the principal components of the principal components are the principal components themselves; since the variance along the principal axes is already maximal,

5 Here we mean Pearson product-moment correlation coefficient, given by $\rho_{X_jX_k} = \frac{\operatorname{Cov}[X_j,X_k]}{\sigma_{X_j}\sigma_{X_k}}$.

6 Something is proper when it is not reducible to something different. For example, division by one is not proper division.

we don't have to transform the first layer output data in order to maximize the variance in the transformed data.

3.2 HEBBIAN LEARNING

In this section we discuss Hebbian learning. Hebbian learning is a class of unsupervised learning techniques for ANNs. It originated from an early neuroscientific theory—Hebbian Theory—and has since grown as a machine learning technique.[13] It has been formalized in a number of ways[10], though we consider only one mathematical formalization. In its simplest interpretation Hebbian learning gives rise to *Hebb's rule*. However, Hebb's rule doesn't constitute a viable learning rule, as is showed in section 3.2.2. We then move on to a learning rule which rectifies the most important problems of Hebb's rule, the Generalized Hebbian Algorithm (GHA), in section 3.2.3. First we provide some background on Hebbian learning.

3.2.1 *Hebbian Theory*

Humans seem to associate things with each other based on contiguity. This led Hebb [16, p. 62] to formulate what was later to be called *Hebb's Postulate of Learning*:

> When an axon of cell A is near enough to excite cell B or repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

In 1992, this postulate was summarized in the now popular canonical phrase "Cells that fire together, wire together." [7] Though Hebb's postulate doesn't exclude a metabolic change, the mnemonic phrase neatly captures the basic idea.

Although Hebbian learning has been formalized and/or extended in numerous ways, we consider only two well-known learning rules based on Hebbian theory. It's these learning rules which are of importance for defining our own learning mechanisms in chapter 4.

In section 3.2.2 we consider the most basic rule based on the core concepts of Hebbian Theory: *Hebb's rule*. The problems of this update rule are overcome by introducing a specific kind of weight decay, giving rise to the Generalized Hebbian Algorithm (GHA), which is discussed in section 3.2.3.

7 The mnemonic phrase is usually attributed to Carla Shatz at Stanford University, according to Doidge [12, p. 427].

3.2.2 *Hebb's Rule*

Hebb's rule is the most straightforward mathematical instantiation for weight change in a Hebbian synapse. The weight update rule is simply given by:

$$\Delta w_i = \eta x_i y \tag{3.9}$$

where

- x_i is the presynaptic input of the synapse with weight w_i;
- y is the postsynaptic output;
- η is the learning rate hyperparameter.

While the output of a neuron is given by the weighted input:

$$y = \sum_i w_i x_i \tag{3.10}$$

An update rule for the bias is not given. Hebbian learning mechanisms generally disregard the bias. In this chapter we assume the bias always to be zero, while in chapter 4 we consider changing the bias using a non-Hebbian learning mechanism.

Updating an ANN using Hebb's rule alone wouldn't result in any desirable functionality. In fact, the problems associated with Hebb's rule apply to any update rule which is based on Hebbian Theory alone.

When the weight update is defined by a Hebbian synaptic modification *alone*, we call it a purely Hebbian synapse. Below we list the problems encountered when an ANN only contains purely Hebbian synapses.

POSITIVE FEEDBACK LOOP A first problem is that a positive weight update causes greater and/or more future positive weight updates. Since a positive weight update causes the presynaptic neuron to more effectively excite the postsynaptic neuron, the postsynaptic neuron shows more activity when the presynaptic neuron is firing. This constitutes a positive feedback loop; the positive weight update causes the postsynaptic neuron to be more active when the presynaptic neuron is active, which in turn causes greater and/or more positive weight updates, etc. After a brief period of sparse pre- and postsynaptic coactivity, a weight will have grown so much that any presynaptic activity will cause postsynaptic activity. At that stage sporadic presynaptic activity alone already causes indefinite growth.

DUPLICATION Whereas the preceding problems signify that there may be no stable state of the network, a wholly different problem is that we might end up with duplicate features. Supposing the problems above have been overcome and weights do converge (to finite numbers), two neurons might still converge to the same feature—they might have similar weights

on the connections from the same input neurons. There is no mechanism by which neurons which are connected to the same input neurons differentiate. The only case when they do differentiate, is when they have been initialized with entirely different input weights. Note that duplicate features don't constitute a problem per se; although two neurons can show the same activity, they might have different connectivity with higher order neurons, causing them to serve different roles in the network after all. Even so, duplication entails inefficiency, since two neurons are performing the same job.

INERTNESS Another problem which relates to weight initialization, is one of starting up the synaptic modification. The input weights to a neuron can get initialized so low as to be unable to ever cause the neuron to fire.[8] Also some learning rule might decrease weights, causing a neuron to become inert. In that case we will never see postsynaptic activity for any of its input synapses. The neuron then serves no purpose.

Clearly, a purely Hebbian synapse isn't viable. From the above problems we can infer some prerequisites when using Hebbian synapses. Besides the Hebbian modification, a Hebbian synapse needs a mechanism for depression—a mechanism by which its weight can decrease and which prevents a positive feedback loop. In section 3.2.3 we introduce such a depression mechanism, which in fact also alleviates the problem of duplication.

Hebbian Objective Functions

Up until now we have described Hebbian learning mechanisms in terms of the update rules used to change the weights. We have not described any objective function which the network optimizes. Without any clearly defined objective, the weight updates of Hebbian learning might seem arbitrary. In this section we show what objective function the update rule approximates, and thereby what functionality a Hebbian neural network performs.

We can interpret Hebb's rule as the result of simple gradient ascent learning for some objective function. As in the previous section, we suppose the network to be a single layer ANN with linear transfer function and no bias. In the case of Hebb's rule we can view $x_i y$ as the partial derivative of the objective function w.r.t. w_i. We know the partial derivative of the output with respect to the weight: $\frac{\partial y}{\partial w_i} = x_i$, so we need an objective function $E(L_O)$ for which the derivative with respect to the output is y.

It is easy to see that for Hebb's rule we can just take the following objective function:

8 Note that we haven't yet discussed which activation function we use. Especially the use of a threshold function may cause the problem of inertness.

$$E(L_{\mathrm{O}}) = \sum_{y \in L_{\mathrm{O}}} E(y) \tag{3.11}$$

$$E(y) = \frac{1}{2} y^2 \tag{3.12}$$

where

- $E(y)$ is the objective function for a single output.

This objective function has the derivative we required and is therefore equivalent to Hebb's rule if we use simple gradient ascent learning. We call this objective the Hebbian Objective (HO).

3.2.3 *Generalized Hebbian Algorithm*

As we have seen in section 3.2.2, using Hebb's rule alone to determine the weight updates introduces quite some problems. We solve these problems by extending Hebb's rule with decay terms, giving rise to the Generalized Hebbian Algorithm (GHA) a.k.a. *Sanger's rule*[33]. The further decay of a weight connected to neuron n_j is based on the output of the neurons $\{n_i \mid i < j\}$. It causes the neuron to converge to the loading vector with maximal variance orthogonal to the preceding principal axes. The N output neurons therefore converge to the first N loading vectors in descending order. The formula for the GHA is given by:

$$\Delta w_{i \multimap k} = \eta y_k \left(x_i - \sum_{j=1}^{k} w_{i \multimap j} y_j \right) \tag{3.13}$$

The GHA is an approximation to a procedure which involves a process which orthogonalizes a given set of vectors called the *Gram-Schmidt process*. After we have explained this process we show that the orthogonalization of the weight vectors makes the extended learning rule converge to the (different) loading vectors. Finally we show that the GHA can be derived from the Hebbian objective with constraints which are enforced using Lagrange multipliers.

Gram-Schmidt Process

The Gram-Schmidt process is a method to perform orthonormalization. It can be seen to consist of two stages: an orthogonalization stage and a normalization stage. However, the two stages can be merged, leading to less complex formulae. In appendix C.11 the Gram-SChmidt process is explained in more detail, and the merged process is derived.

In the resulting process each vector $\mathbf{w}_k$ is orthonormalized w.r.t. all preceding vectors $\mathbf{w}_j$ by the following formula:

$$\mathbf{w}_k^{\perp} = \mathbf{w}_k - \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}} \mathbf{w}_k\right) \mathbf{w}_j \tag{3.14}$$

where

- $\mathbf{w}_k^{\perp}$ is the orthogonalized vector $\mathbf{w}_k$.

Approximation to Loading Vectors

For the first neuron, the GHA reduces to what is known as *Oja's rule*[29, 28], which is given by:

$$\Delta w_i = \eta y \left(x_i - y w_i\right) \tag{3.15}$$

Oja's rule can be derived from the HO where the weight updates are constrained such that the updated weight vector is of unit length. As such it causes the output of the network to have optimal variance given that the weight vector is of unit length, which is exactly the objective of PCA.

For each further neuron, we can prove the GHA to perform Oja's rule restricted to the subspace in which the weight vectors are orthogonal to all previous weight vectors. We thereby show that these weight vectors also converge to loading vectors.

The full proof is given in appendix C.12. There we also show how Oja's rule is derived using a Taylor series and how the Gram-Schmidt process is involved in the GHA.

Lagrangian Derivation

The standard derivation of the GHA and its approximation to the loading vectors involves Taylor series and the Gram-Schmidt process. In this section we provide an alternative derivation using Lagrangian multipliers. We show how Sanger's rule can be derived from the HO subject to normality constraints and orthogonality constraints. This novel method proves to be useful when deriving a novel kind of Hebbian learning in section 4.5.

It is important to note that we cannot just introduce Lagrange multipliers for all constraints, add all Lagrange terms to the objective function and solve for the λ's. Such a procedure would fail due to the fact that it disregards the inherent asymmetry among the output neurons. While the constraints only declare that the weight vectors should be of unit length and orthogonal *to each other*, the GHA is based on the Gram-Schmidt process, which only makes the weight vectors of subsequent neurons orthonormal to previous weight vectors.

Therefore we turn again to the iterative procedure of the GHA (see section 3.2.3), where each loading vector is approximated in turn. We define a

Lagrange function for each output neuron with the constraints pertaining to that neuron.

Since for the first neuron we know that the GHA reduces to Oja's rule, we have already seen proof that it can be derived using the Lagrangian function. When learning weights on connections to further neurons, we have to add more constraints to the Lagrangian function.

We use the same objective function on each output neuron, namely the HO for a single neuron: $E(y_k) = \frac{1}{2}y_k^2$. We maximize this objective w.r.t. the weights $\mathbf{w}_k$ of that neuron. When learning neuron k we have the constraints which demand the weight vector $\mathbf{w}_k$ to be orthogonal to all preceding weight vectors: $\forall j < k : o_{jk} = \mathbf{w}_k^\mathsf{T}\mathbf{w}_j = 0$ and the constraint that the weight vector we are currently learning is normalized: $n_k = \mathbf{w}_k^\mathsf{T}\mathbf{w}_k - 1 = 0$. We are therefore dealing with multiple Lagrange multipliers. Again we make use of a general technique for determining the Lagrange multipliers, given in appendix C.13.

We then define a Lagrange multiplier λ_k for each normalization constraint n_k and a Lagrange multiplier λ_{jk} for each orthogonalization constraint o_{jk}. We then derive (see appendix C.13.1) a function $\widetilde{L}$ with the same stationary points as the original Lagrangian, which has partial derivatives given by:

$$\frac{\partial \widetilde{L}(\mathbf{w},\boldsymbol{\lambda})}{\partial \lambda_k} = 2y_k^2 + 2\sum_{j<k}\lambda_{jk}\mathbf{w}_k^\mathsf{T}\mathbf{w}_j + 4\lambda_k\mathbf{w}_k^\mathsf{T}\mathbf{w}_k \tag{3.16}$$

and:

$$\frac{\partial \widetilde{L}(\mathbf{w},\boldsymbol{\lambda})}{\partial \lambda_{ak}} = y_k y_a + \lambda_{ak}\mathbf{w}_k^\mathsf{T}\mathbf{w}_k + \sum_{j<k}\lambda_{jk}\sum_i \mathbf{w}_a^\mathsf{T}\mathbf{w}_j + 2\lambda_k\mathbf{w}_a^\mathsf{T}\mathbf{w}_k \tag{3.17}$$

When we are in the feasible space we know that the weight vectors are normalized and orthogonalized. We also know that the partial derivatives w.r.t. all Lagrange multipliers should be equal to zero. This reduces the above formulae to the equations:

$$2y_k^2 + 4\lambda_k = 0 \tag{3.18}$$

$$y_k y_a + \lambda_{ak} = 0 \tag{3.19}$$

from which we derive that

$$\lambda_k = -\frac{1}{2}y_k^2 \tag{3.20}$$

$$\lambda_{ak} = -y_k y_a \tag{3.21}$$

When we substitute the λ's in the derivative of the original Lagrangian (see appendix C.13.1), we get:

$$\frac{\partial L(\mathbf{w},\boldsymbol{\lambda})}{\partial w_i} = y_k\left(x_i - \sum_{j\leq k} y_j w_{i\multimap j}\right) \tag{3.22}$$

which coincides with the update rule for the GHA.

We have therefore proved that using Sanger's update rule is equivalent to doing gradient ascent on the Lagrange functions of the output neurons of the ANN.

3.3 AUTO-ENCODERS

Another unsupervised learning technique is given by AEs. An Auto-Encoder (AE) a.k.a. *auto-associator*, or *Diabolo network*, is an MLP for which the output layer has the same size as the input layer. The goal of an AE is to reconstruct its input. In order to do that the derivatives of some error function of the input and the reconstruction are backpropagated through the network and used by the learning mechanism to update the weights.

In the simple case, an AE consists of two layers: the encoding layer and the decoding layer. Their outputs are computed by equation 3.23 and equation 3.24 respectively.

TIED WEIGHTS The weights in the reconstruction layer may be a function of the weights in the encoding layer. In fact, a common type of AE uses the same weights in both layers, but mirrored by taking $\widetilde{\mathbf{W}} = \mathbf{W}^{\mathsf{T}}$.[9] For each neuron the incoming connections have the same weights as its outgoing edges. This property of having both layers of an AE use the same weights is called *tied weights*.

The use of tied weights downsizes the amount of parameters to be optimized and prevents cases in which the input weights to a neuron are very small, while the output weights are rather large, which causes a sigmoid transfer function to act as a linear transfer function, since the sigmoid functions under consideration (the hyperbolic tangent and the logistic sigmoid) approximate a linear function for small activation values.

ENCODING LAYER SIZE We should be careful in choosing the size of the encoding layer; when the code has the same size as the input, we risk learning the identity function. When the weight matrix is given by the identity matrix $\mathbf{I}$, biases are zero and the linear transfer function is used the code of an input data point is equal to the data point itself, in which case the layer is superfluous. Also when a sigmoid transfer function is used, the network can perform the identity function by having small encoding weights, compensated by large decoding weights, in which case the sigmoid transfer function acts as a linear function[3, p. 46].

However, in practice an AE generally converges to local optima different from the identity function. Experiments reported in [4] show that, when using a larger amount of features than the input dimensionality, gradient

9 Note that there is no such relation between the biases used by both layers; we just use two distinct sets of bias parameters.

$$\mathbf{c} = \sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{3.23}$$

$$\mathbf{r} = \sigma\left(\widetilde{\mathbf{W}}\mathbf{c} + \widetilde{\mathbf{b}}\right) \tag{3.24}$$

where

- $\mathbf{x}$ is the input;
- $\mathbf{c}$ is the output of the encoding layer, a.k.a. the *code*;
- $\mathbf{r}$ is the reconstruction, which has the same shape as the input $\mathbf{x}$;
- $\mathbf{W}$ and $\widetilde{\mathbf{W}}$ are the encoding weights and decoding weights;
- $\mathbf{b}$ and $\widetilde{\mathbf{b}}$ are the encoding biases and decoding biases.

descent results in useful features. This may be due to the fact that the large decoding weights needed in order for the network to perform the identity function are difficult to reach—along the way, gradient descent might encounter another local optimum to which it will converge.

When the dimensionality of the hidden layer is larger than the dimensionality of the input, we call the code *overcomplete*. Overcomplete representations of the input can increase the performance of an ANN; they may increase the robustness to noise and form better representations of the statistical distribution underlying the dataset[26].

Denoising Auto-Encoder That being said, we still run the risk of learning the identity function. We would like to employ some technique which causes the identity function not to be (part of) an optimum in the objective function surface. One way is doing this is by extending AEs into Denoising Auto-Encoders (DAEs); rather than trying only to reproduce the input, a DAE also tries to remove noise in the input image. We therefore train a DAE by presenting distorted input data, while backpropagating the error between the reconstruction and the *undistorted* input data. A simple kind of input distortion is given by adding uncorrelated Gaussian noise to the input values.

Another technique which is commonly used to prevent learning the identity function, is by promoting sparsity among the features[27]. Several techniques can be used to make the encoding layer show sparse activity—for any input the code would then largely consist of small values close to zero, while only a small set of features does show activity. However, we won't pursue such techniques, since such a coding scheme leads to an inefficient representation of the input and leads to redundant features and replications of features[7].

3.3.1 *Learning*

When the inputs are binary, or limited within a range, e.g., $[0, 1]$ or $[-1, 1]$, the output layer uses a sigmoid transfer function, such as the logistic sigmoid or the hyperbolic tangent function. Consider the case where the input is (pre-processed) such that each pixel value lies within the interval $[0, 1]$ and we use the logistic sigmoid transfer function. We can then use cross entropy as error function (see section 2.3.3).

A single value v within the range $[0, 1]$ can be viewed as a *Bernoulli distribution*—a binary probability distribution. The probability of class 1 is then given by v, while the probability of class 0 is given by $1 - v$. The error function used is given by the negated total entropy between the true probability distribution and the reconstructed distribution over all pairs of input pixel and reconstruction pixel. The objective is then given by:

$$O = i \log r + (1 - i) \log(1 - r) \tag{3.25}$$

As was the case for multivariate classification (see section 2.3.3), the derivative of the negated entropy E between input i and reconstruction r w.r.t. the activation a of neuron r can be simplified to $\delta_r = i - r$, which leads to fast and simple backpropagation of errors.

When we use tanh as transfer function on the reconstruction layer and the input data lies within $[-1, 1]$, we use an objective function which has the same formula for the local derivative of at the output neurons as above. Taking the antiderivative of the local error yields the objective function given by:

$$O = \frac{1}{2}(i + 1) \log(r + 1) + \frac{1}{2}(1 - i) \log(1 - r) \tag{3.26}$$

3.3.2 *Relation to PCA*

It is important to note the relation between AEs and PCA. When using a linear transfer function the converged weight vectors of an AE span the principal subspace of the input[9]. Though similar to what the GHA does, it is certainly not equivalent. We've seen in section 3.2.3 that an ANN with zero biases and linear transfer function causes the weight vectors to converge to the loading vectors themselves. The weight vectors of an AE with n hidden neurons, however, converges merely to vectors which lie somewhere within the subspace spanned by the first n principal axes.

The fact that AEs are related to PCA in a similar way as the GHA, provides reasonable cause to think that they have similar functionality and similar performance. The method we have developed—SM—is a method for training CNNs and makes use of a method which is an extension to the GHA. In order to make a fair assessment of its performance, we compare

$$c^k = \sigma\left(\sum_{z<N} i^z * \mathbf{W}_z^k + b^k\right) \tag{3.27}$$

$$r^z = \sigma\left(\sum_{k<M} c^k *^+ \widetilde{\mathbf{W}_z^k} + b^z\right) \tag{3.28}$$

where

- N is the number of input maps;
- M is the number of features, i.e. the number of feature maps in the hidden layer;
- r^k, c^k and $\mathbf{W}_z^k$ are viewed as functions over position parameters $[x_\mathrm{p}, y_\mathrm{p}]$;
- $*^+$ signifies convolution with out-of-map evaluation;
- $\widetilde{\mathbf{W}_z^k}$ is the matrix obtained by flipping $\mathbf{W}_z^k$ both vertically and horizontally.

it to an extension to AEs for learning CNNs, namely the Convolutional Auto-Encoder (CAE), which is the subject of the next section.

3.3.3 *Convolutional Auto-Encoders*

A Convolutional Auto-Encoder (CAE) is an AE which is a CNN; we simply combine the two principles. Again we consider an ANN with only one hidden layer, and an output layer with the same dimensions as the input.

We consider a CAE with tied weights. The weights in the first layer follow the specification of CNNs; in fact, the first layer is just a convolution layer as described in section 2.2.1. Since we employ tied weights, the second layer is just a mirror image of the first layer.[10]

The decoding layer can, perhaps surprisingly, be described as another convolution layer. The weight matrices which we convolute are equal to the weight matrices of the encoding layer, excepts that they are flipped both vertically and horizontally.[11] The convolution employed is one which does consider out-of-map evaluation; points lying outside the map are handled as if they are neurons with output value zero.

We can mathematically formalize the CAE under consideration as equation 3.27 and equation 3.28.

10 Note that we do introduce bias terms for the reconstruction layer.

11 Note that this is not equal to taking the matrix transpose.

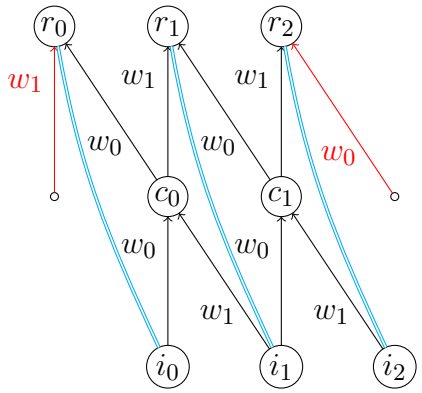


Figure 3.2: A schematic of a CAE on input of size (3×1) from a single input feature, depicting the processing of a feature with convolution field size (2×1). The cyan connections show which reconstructions are associated with which input pixels. The red edges show that the decoding layer makes use of out-of-map evaluation.

The functions r^k, c^k and $\mathbf{W}_z^k$ are defined analogous to the functions f and g in the definition of mathematical convolution in section 2.2.1. In figure 3.2 we show how out-of-map evaluation and the flipping of the weight matrix is equivalent to mirroring the weights of the encoding layer.

3.3.4 *Pooling Convolutional Auto-Encoders*

It is standard to use pooling layers in CNNs. We therefore should consider incorporating the pooling operation when (pre-)training a CNN by a CAE. We can then use a CAE to train each convolution and pooling layer sequentially/layerwise, constituting what is called a *stack of CAEs*, or a *CAE stack*. Without incorporating pooling, CAEs produce features without any structure which do not perform well in a subsequent classification task (Jonathan Masci, personal communication, August 14, 2014).

One might think we can easily define a CAE with pooling by using a convolution and pooling layer for the encoding layer and mirror the design for the decoding layer. However, the pooling layer cannot just be mirrored. The translation invariance the pooling function introduces causes the code[12] to be unable to know where exactly a feature is present in the input. When using a pooling function we cannot reliably reconstruct the input.

This problem is solved by a method proposed by Masci et al. [27]. Their method incorporates the functionality of max-pooling into CAEs. We call their model (and our extension of it) a Pooling Convolutional Auto-Encoder (PCAE). The idea is to set all values in the pool to zero except the maximal value, which is retained. The decoding layer then consists of the mirrored convolution layer alone. Because the pooled value is present only at the location in the pool where the output of the neuron in the pool was maximal,

12 Here 'the code' refers to the output of the encoding layer, consisting of a convolution layer and a pooling layer.

we do have the position information and so the decoding layer can better reconstruct the input.

The step of erasing all non-maximal values greatly increases the performance of PCAEs over CAEs, which can be deduced visually from the features extracted, as presented in their paper. The features extracted by a PCAE generally look like blobs or small line segments[13], while the features extracted by a standard CAE rather look like white noise. However, concrete performance statistics comparing PCAEs to standard CAEs haven't been published.

EXTENSION TO OTHER POOLING FUNCTIONS We extend the ideas of Masci et al. to other types of pooling. We do this by viewing these pooling mechanisms as the result of the inner product of two vectors: the vector of contributions $\mathbf{m}$ and the vector of values in the pool itself $\mathbf{z}$, as described in section 2.2.2. For max-pooling $\mathbf{c}$ contains all zeros excepts for the m_k which corresponds to the maximal value z_k in the pool. For soft arg max-pooling with metaparameter $p = 1$ and $f(x) = x$, the vector m is equal to the output vector of a softmax activation function over the values of the pool. In general the values of $\mathbf{m}$ for soft arg max-pooling are given by $m_k = \frac{e^{pf(z_k)}}{\sum_{z_l \in Z_j} e^{pf(z_l)}}$. Derivatives of the function which converts values to their contributions to the soft arg max-pooled value are given in appendix C.6

The pooling step which causes a CAE to be a PCAE then, consists of multiplying each value z_k in each pool by their contribution m_k. The step makes CAEs suitable for training a CNNs with a convolution and pooling layer. The pooling operation greatly improves the performance, justifying the introduction of methods for extracting pooled convolutional features, rather than just convolutional features.

13 For example a blob of positive weights next to a blob of negative weights, constituting an edge detector, or a line segment of positive negative weights surrounded by positive weights, constituting a detector for a black line segment.

4

Spread Maximization

We have seen in the previous chapter how the Generalized Hebbian Algorithm (GHA) can help in performing Principal Component Analysis (PCA) and how the Auto-Encoder (AE) principle can be applied to Convolutional Neural Networks (CNNs) while considering pooling.

Now we consider a new kind of neural network learning model, which is based on *spread*, a concept which is explained in section 4.1. This concept can be interpreted in two ways, giving rise to two kinds of methods for learning: a method for dichotomization, which is handled in section 4.3 and one for uniformization, which is handled in section 4.4. Several instantiations of these kinds of methods are described and weighed. In section 4.5 we introduce an overarching method which can be used for either dichotomization or uniformization. This method also solves some inherent problems of the methods for dichotomization and uniformization. The ideas are evaluated empirically in chapter 5.

4.1 SPREAD

One of the problems with using (purely) Hebbian learning in a layered feed-forward neural network is one of differentiation. Since neurons then have only local update rules and neurons are generally connected to the same input neurons, they get similar local information, which causes them to converge to similar features. The problem holds especially when the neurons are initialized with the same weight configuration, but even with differently initialized weights, there's nothing keeping neurons from converging to similar features.

It therefore seems like a good idea to change the update rule such that it incorporates some differentiation mechanism. The desirable state is one in which the neurons represent quite different features and so respond quite differently to different input data. We can visualize this by an output space in which the data points are mapped to quite different locations. One might say the outputs for different data points should be *spread out* over the output space. Other terms used for spread are 'dispersion', 'scatter' and 'variability', though these have generally been used to describe a univariate spread measure.

Instead of incorporating some spread term into an existing learning mechanism, we might also use spread as the sole objective of a network. Of course it wouldn't be a good idea to make a model which randomly generates

spread output irrespective of the input. However, when the spread objective is used as the objective function of an Artificial Neural Network (ANN) and the weights are updated so as to maximize this objective function, the spreading is generally based on the input data.

Most learning methods maximize spread by automatically making use of the most differentiating features in the data, because most learning methods make use of the partial derivative of the objective function, which is larger for more common input patterns. To explain this we first need to explicate the notion of spread.

Spread can be viewed in two ways, corresponding to two ways in which we use the word. We could say that the shell of a bomb gets spread out when it explodes, but we can also say that the gas it releases spreads out. In the former case an increasing distance from the origin in all directions is implied, while in the latter an increasing volume in which particles are all around is implied. We can view the former as points on the surface of a growing (hyper)sphere, while we can view the latter as points within the volume of a growing (hyper)sphere. The former kind of spread is maximized by a process which we call *dichotomization*, while the latter is maximized by something which we call *uniformization*. Section 4.3 covers a method to perform dichotomization, namely Eigenvolume Expansion-Dichotomization, while section 4.4 cover a method to perform uniformization, namely Pre-Sigmoid Gaussian-Uniformization. But first, we discuss how we handle CNNs when using any of the methods used for Spread Maximization (SM).

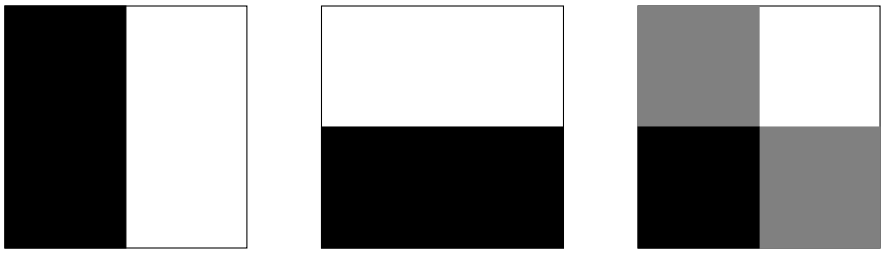
4.2 SM for CNNs

In CNNs pooling layers are used as a way of reducing the dimensionality of the input to the subsequent layer and as a way of making its functionality invariant to small deformations. However, when training a CNN layerwise, using an unsupervised learning method such as SM, the top layer doesn't have any subsequent layer yet; the arguments of dimensionality reduction and deformation invariance do not apply. We may therefore have to treat the top pooling layer differently from lower layer pooling layers.

Instead of doing away with the top pooling layer during learning because of the lack of arguments to use it, we present new arguments to use a pooling layer in the context of an unsupervised objective. These arguments are of a different nature, which leads to different pool sizes than the normal (2×2) pools. The argumentation is twofold, based on the inherent anti-correlational nature of features and on the information theoretic use of features in the next layer.

Let's consider the least complex example of a convolution layer with a pooling layer with pools of size (1×1). The outputs of the pooling layer then cover a visual field the size of which is exactly equal to the size of a feature of the convolution layer itself. This means that in that position

in the output maps of the pooling layer each neuron provides information on in what respect that patch in the input conforms to the feature that neuron represents. Consider a patch in the input which perfectly conforms to a given feature; in order to have it conform more to another feature we've got to have different inputs, which in turn means the input conforms less to the first feature. [1] A given patch can for example never contain a maximal horizontal contrast, while also having a maximal vertical contrast; see figure 4.1.



(a) Horizontal contrast (b) Vertical contrast (c) Both

Figure 4.1: Input patches conforming to a horizontal contrast feature, a vertical one and to a lesser extent both.

This negative correlation is in sharp contrast to our dichotomization objective. If we know the output neurons to be (negatively) correlated, we know that we can't reach the optimal decorrelated output distribution. We should therefore enhance the pool size so that such negative correlations don't necessarily occur. It seems to be a good idea to make the size of the pool close to the size of the visual field of the neurons in the convolutional layer. That way it is possible to have a place in the output map where two mutually exclusive features are fully present because they are present next to (and/or below) each other in the input.

When we've finished training of the current layer and move on to train the next layer, we reduce the pooling size back to the normal (2×2), so that the dimensionality reduction doesn't take away too much spatial accuracy.

4.3 DICHOTOMIZATION BY EIGENVOLUME EXPANSION

As we have indicated above, the process of dichotomization might be seen as spreading all data points away from the origin in the output space. [2] Though ideally this is done by having all points located on the surface of a growing hypersphere, in practice the output data points won't be located

1 We suppose that it is not the case that the two neurons have all zero weights at the places where the other neuron has non-zero weights. There is thus an actual overlap in the input field.

2 The word 'dichotomy' is chosen because of its meaning in natural language as 'contrast on a gradual scale'.

exactly on such a surface. We are therefore in need of some way to identify a surface typical for the given output distribution.

In the univariate case we could simply take the standard deviation as a measure of distance from the origin, which is then identified by the mean. If the data points are then evenly distributed across two points equidistant from the origin and lie exactly on those points, the standard deviation gives us the distance of those points to the origin.[3] In that case the line between the two points is the hypersphere for which all points lie exactly on its surface. Growing the hypersphere then comes down to growing the line between the two points at a distance of one standard deviation from the mean.

The extension of the above method to the multivariate case might not be as straightforward as one might think. First note that the concept of spread intuitively disregards any directionality of the distribution w.r.t. the coordinate system of the output space; whether the data points lie on the line segment OA or on a line segment OB where $O = (0, 0)$, $A = (0, 5)$ and $B = (3, 4)$, shouldn't matter to our measure of spread. Instead of considering the standard deviations in the coordinate system of the output space, we look at the standard deviations in the coordinate system of the *principal subspace* of the distribution. [4]

Two methods come to mind which reduce to the standard deviation in the univariate case. On the one hand we might define dichotomy as the sum of the standard deviations along all axes of the principal subspace, while on the other hand we can use the product.

Recall that the variance in each dimension of the principal subspace is given by an eigenvalue of the covariance matrix (Section 3.1.1). The standard deviations in the principal subspace are therefore given by $\sqrt{\lambda_i}$: the square root of the eigenvalues. We can define dichotomy as equation 4.1.[5]

We can associate the measure of dichotomy D with the volume of a hyperrectangle whose sides have lengths equal to the standard deviation in each eigendimension. We call this hyperrectangle the *eigenvolume*. The method described in this section is therefore called Eigenvolume Expansion-Dichotomization.

When we visualize the eigenvolume alongside the distribution, we see that it neatly captures in which way the output data points are distributed in the space; see figure 4.2. In figure 4.3, you can see an example of a 3-dimensional

3 For an n-dimensional hypervolume, the hypersurface is $(n-1)$-dimensional. So in the univariate case where $n = 1$, the hypersurface is 0-dimensional. That is why we talk here of points instead of surfaces.

4 In the univariate case the principal subspace coincides with the output space, so looking at the standard deviations in the principal subspace provides a valid ground for a generalization of the method described in the above paragraph.

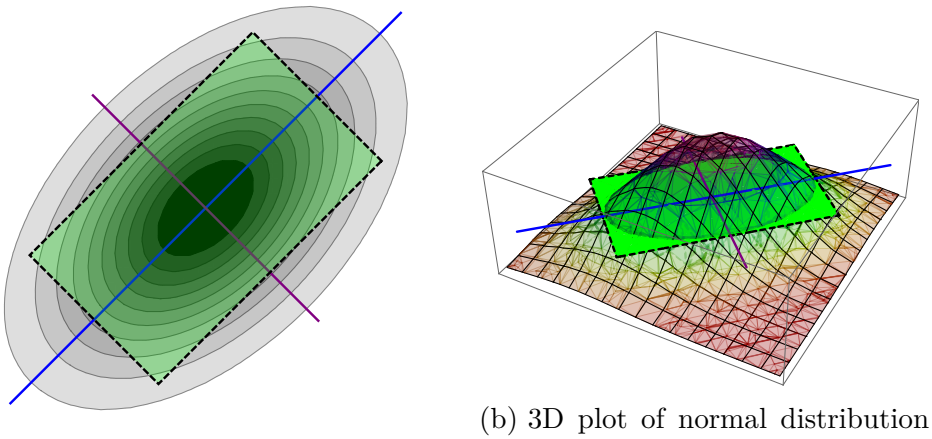
5 The equality used in the equation below has been taken from [31].

$$D = \prod_i \sqrt{\lambda_i} = \sqrt{\prod_i \lambda_i} = \sqrt{\det |\mathbf{\Sigma}|} \tag{4.1}$$

where

- $\mathbf{\Sigma}$ is the covariance matrix of the output data;
- $\det |\mathbf{\Sigma}|$ denotes the determinant of $\mathbf{\Sigma}$.

Gaussian distribution with the corresponding eigenvolume when you stare at the image cross eyed. [6]



(a) Contour plot of normal distribution in gray.

(b) 3D plot of normal distribution in rainbow colours indicating height.

Figure 4.2: Graph containing a plot of a multivariate normal distribution ($\mathbf{\Sigma} = \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix}$), its loading vectors in blue and purple and the eigenvolume in the dashed green box.

6 In literature, formula D is more commonly referred to as a measure of dispersion, in contrast to a measure of 'dichotomy'. However, 'dispersion' is often used as synonymic to 'spread', which we have already used as an abstract term covering both dichotomy and uniformity. We have therefore chosen for 'dichotomy' instead of 'dispersion'.

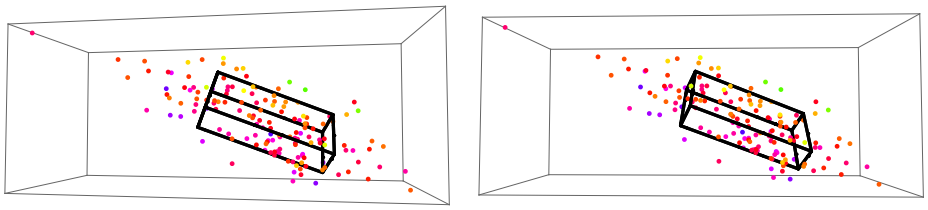


Figure 4.3: Stereogram of points sampled from a multivariate Gauss and the corresponding eigenvolume. Squint to achieve stereopsis.

When each neuron in the output layer is constrained to a certain interval, e.g., $(-1, 1)$ for the hyperbolic tangent transfer function, D is maximized when the eigenvolume coincides with the hypercube of the output space. Since the coordinate system of the principal subspace then aligns with the coordinate system of the output space, we see that the ideal output distribution is decorrelated. The standard deviation in each dimension is then equal to 1, since all points lie on the edge of the output space. We can therefore say that it is part of our objective to decorrelate and standardize. A more direct approach to perform decorrelation and standardization is performed by the Hebbian Objective (HO), which is treated in section 4.5.

4.3.1 *Update Rule*

The derivative of the objective is given by:[7]

$$\frac{\partial D}{\partial p} = \frac{1}{2}\sqrt{\det|\mathbf{\Sigma}|}\mathbf{\Sigma}^{-1}\frac{\partial \mathbf{\Sigma}}{\partial p} \tag{4.2}$$

where

- $\mathbf{\Sigma}$ is the covariance matrix of the output data;
- $\det|\mathbf{\Sigma}|$ denotes the determinant of $\mathbf{\Sigma}$.

However, multiplication by $\frac{1}{2}\sqrt{\det|\mathbf{\Sigma}|}$ only influences the step size, since the eigenvolume is always positive. We can therefore leave it out and use the simplified derivatives given by:

$$\frac{\partial \widetilde{D}}{\partial p} = \mathbf{\Sigma}^{-1}\frac{\partial \mathbf{\Sigma}}{\partial p} \tag{4.3}$$

The derivatives given by the formula above are then used in combination with the derivatives of the entries of the covariance matrix (which are given in appendix C.16) by a learning mechanism such as gradient ascent to determine the weight updates.

7 See appendix C.14 for a full proof.

4.3.2 *Underdetermination*

As we already briefly mentioned in section 4.1, simple learning methods like gradient ascent follow derivatives which are larger for more distinctive features. The partial derivative w.r.t. a weight depends on the magnitude of its input: $\frac{\partial E(\mathbf{w})}{\partial w_{k-\circ n}} = \delta_n k$ (Formula 2.32). Therefore, the inputs which are largest (in an absolute sense) for a given data point tend to dominate the direction of the derivative, such that the weight update accommodates the input pattern.

If we look at a batch of data samples, we see that data points which have similar outputs have similar objective function values and similar partial derivatives w.r.t. activations—their local objectives δ_n are similar. The accumulated partial derivatives over these data points are larger for weights connected to input neurons which consistently showed positive of negative activity over these data points, compared to weights connected to input neurons which showed both positive and negative activity. This means that the most prominent features common to these data points are leading in determining the size and direction of the partial derivatives of the objective function w.r.t. the weights of a neuron.

When we would maximize our objective by making use of learning methods which don't rely so much on the partial derivatives w.r.t the weights as gradient ascent, we cannot guarantee that the features which are extracted represent prominent features in the data. In fact, we can show that there is a vast set of weight configurations for an ANN which would produce the same objective function values but doesn't represent common or prominent features of the input at all. You could say that the objective function is underdetermined; out of a vast set of weight configurations which result in places in the objective surface 'equally close' to a global optimum, we only consider a small number to be intuitively correct solutions for dichotomization. [8]

To see why this is so, consider any set of not perfectly correlated weight configurations, whether we would view them as intuitively optimal or not. Each neuron with one of these weight configurations has a standard deviation which might very well be suboptimal. However, for neurons with a linear transfer function or any other monotonic transfer function, the output of neurons in the output layer can be made bigger by increasing all concerned weights. We can multiply all weights by some large constant, so that the standard deviation is equal to what we consider to be the optimal standard deviation. We can therefore always increase the dichotomy

8 Actually there is no global optimum, as is showed below. However, the reasoning here still holds irrespective of the absence of the optima within finite space. Points are said to be equally close to a global optimum if their objective functions have the same value. For any finite global maximum, any two points on any function surface with the same value are said to have the same distance to the global optimum.

by increasing all weights, while the neurons essentially represent the same features which may have been ones we would view as suboptimal.

Another consequence of the fact that multiplication of the weights always leads to a larger dichotomy is that there is no global optimum. Therefore, it is hard to specify a criterion for when the learning process should stop. However, when a sigmoid function is used as transfer function for the output nodes, the output space is limited to a hypercube and the eigenvolume has an upper limit.

The problem of underdetermination might be overcome by introducing some form of weight control. When we add a simple weight decay regularization term to our objective function, the function surface falls when wandering farther from the origin. This holds when we use sigmoid transfer functions, or when we use a linear transfer function, but after a given distance the evaluation function is dominated by the weight decay regularization term instead of the objective function.

The addition of weight decay causes maxima that were farther away from the origin to disappear irrespective of whether these were maxima which we would view as optimal or suboptimal. It also introduces new maxima at places where a climbing ridge in the objective surface becomes overpowered by the regularization term. The hyperparameter λ of the weight regularization term (formula 2.42) might even be such that the only global maximum is at the origin, corresponding to all zero weights. This shows that the addition of the weight decay term might disregard existing maxima which we would consider as good maxima and introduce new maxima which we would consider as bad, depending on the value of λ.

It is hard to determine the right value of λ, since the objective surface depends on the data at hand. The value at which it is too high, corresponding to neglect of too many (or all) maxima, also depends on the data. On the other hand we might set the value of λ too low, corresponding to a case in which we would end up with too many maxima which we wouldn't intuitively consider optimal. We thus disregard a normal weight decay regularization term, on the grounds that it either leads to unwanted maxima, depending on the data, or needs a too involved procedure to overcome such a problem.

Another form of weight control is to restrict the weights such that the vector of weights connecting to each neuron has a length of 1. This won't change the objective surface, but it restricts the possible solutions to a subspace within that surface, called the *the feasible space*. This *normalization constraint* gives rise to *generalized Hebbian learning*, which is covered in section 4.5.

4.3.3 *Mean Centering Problem*

As mentioned in section 4.3, the eigenvolume is maximized when all output data points are at the edge of the hypercube which represents the possible output space. To be more specific: dichotomy is maximized when each neuron maps exactly half of the data points to -1 and the other half to 1 (in the case of a hyperbolic tangent transfer function) and the neurons are decorrelated. Thus we know that the mean should be at the origin.

However, when at a certain iteration in the learning process we get in a state where a majority of points is mapped to one side of their mean and the rest to the other, we are stuck in a suboptimal region in the objective surface from which gradient ascent cannot escape. Let's give a simple example in the univariate case of a single output neuron, no (or (1×1)) pooling and four data points. Maximizing dichotomy then comes down to maximizing the variance of the output data. Suppose three data points are mapped to a similar value, but the fourth one is quite far from the rest, such that the mean of all points lies between the two groups. In order to get into a symmetric state, one point needs to move over to the other side. However, it then has to cross the mean, while the variance is always lower when points are closer to the mean. The derivative therefore never points toward the mean for any data point and so we are stuck in a suboptimal region of the objective surface.

We could try to solve this issue by introducing some regularization term, or by introducing some constraints on the learning process, but we won't pursue such a line of research in this thesis.

4.3.4 *Binarization*

A principle characteristic of dichotomization in general is that it leads to binarization, in the case a sigmoid transfer function is used. The output data points all tend toward either side of the spectrum. The iterative learning procedure grows toward binarizing the output data. However, it will never reach this end goal. We therefore manually perform the binarization by switching from sigmoid transfer function to the hard transfer function of which it is an approximation. The hyperbolic tangent function will be switched to the signum function and the logistic sigmoid function will be switched to its counterpart, which is known as the *Heaviside step function.*

In a sense binarization is just what we wanted to achieve, but there may also be some downsides to it. In abstract terms it seems to be a reduction in informational value to go from a spectrum of possible outputs to binary alternatives. When a neuron outputs a variety of values on a scale between 0 and 1 it can be seen to provide information as to in what respect a certain feature is present in the input, while a neuron which outputs either 0 or 1 seems to apply a hard threshold to the same information. The binarization

of the outputs discards information on where in the spectrum its activation value was.

We therefore consider another kind of objective: uniformization. Uniformization radically disposes of the binarization that comes with dichotomization by aiming for uniformity instead of dichotomy. Uniformization is the subject of the next section.

4.4 UNIFORMIZATION BY CHANGING THE PRE-SIGMOID OUTPUT DISTRIBUTION

Uniformization is a different interpretation of spread than dichotomization. Instead of pushing all output data points away from the middle, uniformization expands the volume in which data points are scattered throughout. When we use sigmoid transfer functions, which we generally do, dichotomization comes down to pushing all data points toward the edges of the output space, while uniformization comes down to getting the output data uniformly distributed.

Uniformization might be better than dichotomization, because the outputs are on a scale between the edges of the output space instead of only on the edges. A subsequent layer might therefore retrieve more information from the outputs of a uniformized layer.

There are multiple ways in which we can perform uniformization. One approach is to fit a distribution on the output data and update the weights such that the distance between that distribution and the uniform distribution is minimized. This technique is the subject of section 4.4.1 to 4.4.3. Another way of doing uniformization is described in section 4.5. There we describe a method which gives rise to techniques for performing either dichotomization or uniformization, which deal with some of the inherent complications in our first definitions of spread given in sections 4.3 and 4.4.

4.4.1 *Modelling the Output*

Our approach to perform uniformization makes use of a multivariate probability distribution fitted on the output data. The objective is then to minimize the distance of the fitted distribution to the uniform distribution. This distance is measured in terms of the Kullback-Leibler (KL) divergence of the fitted distribution from the ideal distribution.

The main issue here is the question of what kind of model to use to fit to the data. One of the most commonly used multivariate continuous distribution functions is the *multivariate normal distribution*, a.k.a. the *multivariate Gaussian distribution*. However, its support is the entire output space, $\mathbb{R}^N$, while the output of an ANN with sigmoid transfer function is restricted to a hypercube of fixed size, e.g., $[0, 1]^N$.

$$\text{for } \mathbf{x} \sim \mathcal{N}^k(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{4.4}$$

$$f_{\mathbf{x}}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^k \det|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{4.5}$$

where

- $\boldsymbol{\mu}$ is the vector of means;
- $\boldsymbol{\Sigma}$ is the covariance matrix;
- k is the dimensionality.

4.4.2 *Pre-sigmoid Gauss*

Instead of modelling the output with a probability density function which has a support of $[0,1]^k$, we use the multivariate normal distribution to model another distribution which isn't limited to a hypercube in the output space; we look at the output neurons *activations*, i.e. we model the output signals of the network before the sigmoid transfer function is applied. That way we can fit traditional multivariate probability density functions (PDFs), such as the multivariate Gaussian distribution, which is given by equations 4.4 and 4.5. We call this method Pre-Sigmoid Gaussian-Uniformization.

Pooling

Note that usually in CNNs the convolution layer applies a sigmoid transfer function, while a pooling layer only summarizes the information from the previous convolution layer and doesn't use a transfer function (i.e. uses a linear transfer function). Therefore the activations of the pooling layer are still limited to a hypercube, e.g., $[0,1]^N$, which is what we are trying to sidestep.

However, we can move the transfer function used from the convolution layer to the pooling layer, given that the transfer function used in all neurons in a pool are the same sigmoid function and that one of the standard pooling functions is used. The transfer function of neurons in the convolution layer then becomes linear and the transfer function of pooling neurons becomes the one used in the convolution layer.

Because sigmoid functions are monotonically increasing and the max function returns the value of one of its inputs, we know that: $\sigma \circ \max = \max \circ \sigma$. The same reasoning holds for the absolute maximum. The soft approxima-

tions to these functions don't just return one of its input values; it is easy to see that $\sigma \circ \text{soft arg max} \neq \text{soft arg max} \circ \sigma$. However, remember that soft arg max was only a soft approximation to the max function. Therefore soft arg max $\circ$ max is an approximation to max $\circ \sigma$ and hence to $\sigma \circ$ max; at least, under the setting of hyperparameter p (see formula 2.28) which makes it approximate the max function, instead of the average.

When the pooling function consists of taking the average of its inputs, moving the transfer function upward changes the functionality of the pooling layer radically. The average of the sigmoid transformed input is radically different from the sigmoid transformed average. When applying the sigmoid after averaging, an input with a high value is far more influential in determining the output than when we average the sigmoid transformed inputs. If a big positive activation is far larger than a big negative activation, the pooled output is close to 1, and not to the middle, e.g., 0, which could be the case when applying the sigmoid after averaging. This also makes for an interesting pooling scheme; We therefore say that moving the transfer functions from a convolution layer to the soft arg max pooling layer above only changes the functionality of the network negligibly.

4.4.3 *Probability Integral Transform*

Consider a case where as transfer function we take the cumulative density function (CDF) of the univariate normal distribution with zero mean and unit standard deviation, i.e. the *standard normal distribution*:

$$\text{for } x \sim \mathcal{N}^1(x; \mu, \sigma) \tag{4.6}$$

$$f_x(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \tag{4.7}$$

where

- μ is the mean of the normal distribution;
- σ is the standard deviation of the normal distribution;

The transfer function we consider is then its CDF, which is given by:

$$\text{for } x \sim \mathcal{N}^1(x; \mu, \sigma) \tag{4.8}$$

$$F_x(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sqrt{2\sigma^2}}\right)\right) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{4.9}$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_o^x e^{-t^2} \mathrm{d}t \tag{4.10}$$

where

- erf is the Gauss error function.

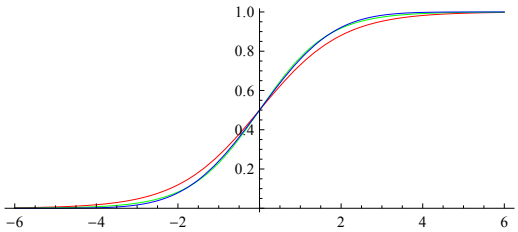


Figure 4.4: In blue the CDF of the standard normal distribution, in red the logistic sigmoid function and in green the logistic sigmoid approximation.

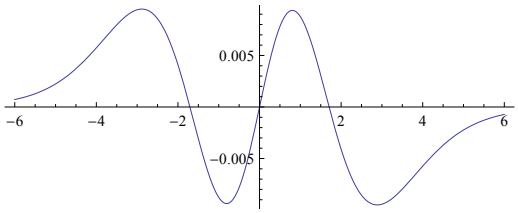


Figure 4.5: The difference between our logistic sigmoid approximation and the CDF of the standard normal distribution.

In figure 4.4 the CDF of the standard normal distribution is depicted in blue. The function has a 'S'-shaped curve and falls within the category of sigmoid functions. It is not that different from the logistic sigmoid function, depicted in red. In fact, because a CDF is by definition a monotone function, the CDF of a continuous probability distribution is always a sigmoid function $\sigma(x)$ such that for $x \to \infty$ we have that $\sigma(x) \to 1$ and that $\sigma(-x) \to 0$.

When the activations of a neuron then follow the standard normal distribution, we know that the output of that neuron conforms to a uniform distribution. This is so because of the fact that the *probability integral transform* of a distribution X results in the uniform distribution (see appendix C.17). The probability integral transform is the application of the cumulative density function to the distribution itself:

$$F_X(X) = U^1_{[0,1]} \tag{4.11}$$

where

- X is any distribution;
- F_X is the CDF of X;
- $U^1_{[0,1]}$ is the univariate uniform distribution on the interval $[0, 1]$.

Therefore, we know that when the transfer function of a neuron is given by the CDF of the standard normal distribution and its activations follow a standard normal distribution, the output is uniformly distributed. We call the standard normal distribution the *ideal distribution* when using the Gaussian CDF as transfer function; it is the ideal for Pre-Sigmoid Gaussian-Uniformization (PSGU), because it gives us a uniform output distribution.[9]

9 The same holds for the logistic distribution and the logistic sigmoid transfer function. However, we won't consider using the logistic distribution to model the pre-sigmoid acti-

However, the CDF of the standard normal distribution is inefficient to use in actual computations, since the integral in the error function is not analytically solvable. We therefore try to approximate the CDF by using the logistic sigmoid function; our approximation has the form of a stretched logistic sigmoid: $\sigma^l(sx)$, where s is the stretch factor. We then minimize the distance between these two functions by minimizing: [10]

$$\int_o^\infty \left(\sigma^l(sx) - F_{\mathcal{N}^1(0,1)}\right)^2 \mathrm{d}x \tag{4.12}$$

We find that at $s \approx 1.203$ this simple distance measure is minimized. [11]

Instead of adjusting the standard logistic sigmoid transfer function by the stretching factor s, we stretch the ideal distribution for the Gaussian CDF transfer function, in order to approximate the ideal distribution for a logistic sigmoid transfer function. Since we suppose that $x \sim \mathcal{N}^1(x; 0, 1)$, by change of variables we get:

$$\begin{aligned} f_Y(y) &= \left|\frac{\partial g^{-1}(y)}{\partial y}\right| f_x(g^{-1}(y)) \\ &= \frac{1}{s} f_x\left(\frac{y}{s}\right) \\ &= \frac{1}{s}\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{\left(\frac{y}{s}\right)^2}{2}\right) \\ &= \frac{1}{s\sqrt{2\pi}} \exp\left(-\frac{y^2}{2s^2}\right) \\ sx &\sim \mathcal{N}^1(0, s) \end{aligned} \tag{4.13}$$

where

- $g(x) = sx$

The above result holds for every neuron in the output layer, so we say that the ideal pre-sigmoid multivariate normal distribution for the logistic sigmoid transfer function, has marginals which follow a univariate normal distribution with zero mean and a standard deviation of s; the optimal standard deviation is given by $\sigma^* = s$. Since for a multivariate normal distribution with means $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ the marginal for the n^{th} activation are given by $\mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_{nn})$, we know that the ideal distribution is

vations, because defining a multivariate extension of the logistic distribution which meets the necessary conditions is not as straightforward as the multivariate normal distribution.

10 Since both the logistic sigmoid and the CDF of the standard normal are rotationally symmetric about $(0, 0.5)$, their squared difference is symmetric about $x = 0$. We therefore can limit the minimization to either half of the function space.

11 More accurately: 1.2027825075620033.

such that the means are given by the zero vector and the diagonal of the covariance matrix only contains $(\sigma^*)^2$. [12] We have yet to provide reasoning to fill in the rest of the covariance matrix, in order to complete the description of the ideal pre-sigmoid distribution.

It should be quite obvious that the (proper) covariances between the activations of different neurons should be zero. When we would allow the ideal pre-sigmoid distribution to have covariances other than zero, the post-sigmoid distribution would also have non-zero covariances and hence it wouldn't be uniform, even though its marginals are uniform. The ideal pre-sigmoid normal distribution is therefore given by:

$$\mathcal{N}^* = \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix}, \begin{bmatrix} (\sigma^*)^2 & 0 & \dots \\ 0 & (\sigma^*)^2 & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}\right) \tag{4.14}$$

Let's take a step back. We've seen how the use of a CDF as transfer function of the distribution we fitted to the pre-sigmoid data leads to a uniform post-sigmoid distribution. We then went on to approximate the CDF of the normal distribution using the logistic sigmoid, since it leads to faster computation. The stretching factor s could then be used to redefine (an approximation to) the ideal pre-sigmoid distribution, so that we can still use the standard logistic sigmoid transfer function. We determined s by approximation of the CDF of the normal distribution, while now it only occurs in the pre-sigmoid ideal distribution; s isn't involved with the transfer function directly. It is therefore better to choose s such that the approximation of the ideal is actually closest to the ideal pre-sigmoid.

This would still be taking a detour, however. Our objective is to make the output distribution more uniform; we want to decrease the distance between the actual output distribution and the uniform distribution. We should therefore estimate s such that the post-sigmoid distribution of the approximation is closest to the uniform distribution.

As distance measure we take the Kullback-Leibler (KL) divergence from the uniform distribution P to the post-sigmoid distribution of the approximation Q:

$$D_{\mathrm{KL}}(P \mid\mid Q) = \int_D f_P(x) \log\left(\frac{f_P(x)}{f_Q(x)}\right) \mathrm{d}x \tag{4.15}$$

where

- D is the domain of the PDFs—their support, e.g., $[0, 1]$.

12 The multivariate normal distribution was defined in terms of (co-)variances, while the univariate normal was defined in terms of the standard deviation. Because we need to square the standard deviation to get the variance, we get s^2 instead of s as parameter to the distribution.

The PDF $f_P(x)$ of the uniform distribution $U^k_{[0,1]}$ is a constant function, always returning 1. [13] We can therefore simplify:

$$\begin{aligned} D_{\mathrm{KL}}(U^k_{[0,1]} \mid\mid Q) &= \int_D 1 \log\left(\frac{1}{f_Q(x)}\right) \mathrm{d}x \\ &= -\int_D \log\left(f_Q(x)\right) \mathrm{d}x \end{aligned} \tag{4.16}$$

The post-sigmoid distribution Q of the approximation to the uniform distribution is harder to characterize. We compute the post-sigmoid PDF by converting the pre-sigmoid distribution R using the logistic sigmoid transfer function. In appendix C.15 we derive that:

$$f_Q(x) = \frac{1}{x(1-x)\sigma^*\sqrt{2\pi}} \exp\left(-\frac{\log\left(\frac{1}{1-x} - 1\right)^2}{2(\sigma^*)^2}\right) \tag{4.17}$$

where

- σ^* is the optimal standard deviation;
- $R = \mathcal{N}(0, \sigma^*)$

The distance is minimized at $\sigma^* \approx 1.814$. [14] In figure 4.6 the difference between the approximations of the ideal output distributions for different settings of s are depicted. Besides the two settings we have discussed, a third option is shown in blue. This is the setting of s when the approximation of the ideal is closest to the ideal pre-sigmoid, which we called a detour above.

Up until now we have discussed what the optimal pre-sigmoid distribution looks like when using the logistic sigmoid as transfer function. We would also like to know what it would look like when we would use the hyperbolic tangent as transfer function. Recall that we had conveniently defined the hyperbolic tangent in terms of the logistic sigmoid (formula 2.6): $\tanh(x) = 2\sigma^l(2x) - 1$. The scaling by 2 and shift by -1 only transform the output space in a way which preserves the uniformity of the output distribution in the output space. We need to account for the pre-sigmoid multiplication by 2. This is simply done by dividing the value we have found for σ^* or the logistic sigmoid function by 2. Thus the optimal standard deviation for each pre-sigmoid output is given by $\sigma^* \approx 0.9069$ when using the hyperbolic tangent transfer function.

OBJECTIVE Now that we have fully specified the ideal pre-sigmoid Gauss which best approximates the ideal pre-sigmoid distribution leading to a uni-

13 When the uniform distribution has a domain different from $[0, 1]^k$, the constant value the PDF returns is different. For a domain of $[a, b]^k$ the values returned are $(b - a)^{-k}$, such that the volume under the function is 1.

14 More precisely: 1.8137993369195464.

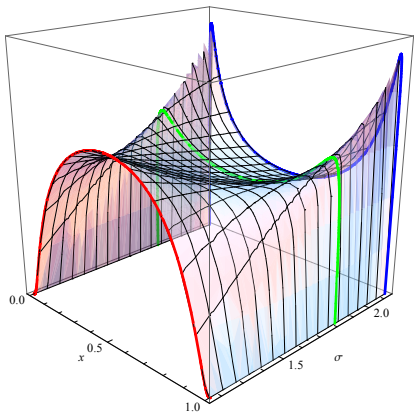


Figure 4.6: Plot of the post-sigmoid distribution for different values of s, the standard deviation of the pre-sigmoid univariate normal distribution. In red the distribution obtained when choosing s such that the sigmoid transfer function approximates the CDF of the standard normal distribution; in blue when s is chosen such that the pre-sigmoid normal distribution best approximates the ideal logistic distribution; in red when s is such that the post-sigmoid distribution best approximates the uniform distribution.

form output distribution, we can define the objective of the network. The objective is to minimize the distance of the approximated ideal distribution from a Gaussian distribution fitted to the actual pre-sigmoid activations of the output layer. This distance is given by the KL divergence from the approximated ideal to the fitted distribution, see formula 4.15.

Learning is performed on the network without the sigmoid function in the output layer, or equivalently, the objective function is applied to the output neurons activations instead of their outputs. The objective function itself is then given by the negated KL divergence:

$$E(L_{\mathrm{O}}) = -D_{\mathrm{KL}}(\mathcal{N}^k(\mathbf{0}, s\mathbf{I}) \,||\, \widehat{\mathcal{N}}) \tag{4.18}$$

where

- k is the dimensionality of the output layer: $k = |L_{\mathrm{O}}|$;
- $\mathbf{I}$ is the identity matrix;
- $\mathbf{0}$ is the zero vector;
- $\widehat{\mathcal{N}}$ is the $|L_{\mathrm{O}}|$-dimensional multivariate normal distribution fitted to the activations $a_1 \dots a_N$;
- N is the number of data points in the sample.

The KL divergence between two multivariate normal distributions actually has a widely know analytical solution.For two distributions $\mathcal{N}_0(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$ and $\mathcal{N}_1(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$, we have:

$$
\begin{aligned}
D_{\mathrm{KL}}(\mathcal{N}_0 \,||\, \mathcal{N}_1) =& \frac{1}{2} \operatorname{tr}\left(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0\right) \\
&+ \frac{1}{2}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^{\mathsf{T}}\boldsymbol{\Sigma}_1^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) \\
&- \frac{1}{2}\left(k + \log\frac{\det|\boldsymbol{\Sigma}_0|}{\det|\boldsymbol{\Sigma}_1|}\right)
\end{aligned} \tag{4.19}
$$

where

- k is the dimensionality of the distributions

In our case this reduces to:

$$
\begin{aligned}
D_{\mathrm{KL}}(\mathcal{N}^k(\mathbf{0}, s\mathbf{I}) \,||\, \widehat{\mathcal{N}}) =& \frac{1}{2} ks \operatorname{tr}\left(\boldsymbol{\Sigma}_1^{-1}\right) \\
&+ \frac{1}{2}\boldsymbol{\mu}_1^{\mathsf{T}}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 \\
&- \frac{1}{2}\left(k + \log\frac{s^k}{\det|\boldsymbol{\Sigma}_1|}\right)
\end{aligned} \tag{4.20}
$$

For which the derivative is given in appendix C.14.

4.4.4 *Underdetermination*

In much the same way as Eigenvolume Expansion-Dichotomization (EED), PSGU has an underdetermined objective. PSGU suffers from the same problems as EED (see section 4.3.2). Whereas with EED we could multiply any weight configuration by a large constant in order to maximize the objective, with PSGU we can multiply the weight configuration of any neuron such that the activations of that neuron have a standard deviation with the optimal value s. For any set of uncorrelated features, we can apply this multiplication method so that the activations fit the approximated ideal distribution exactly. [15] The objective of PSGU allows for weight configurations which are intuitively suboptimal.

Just like the underdetermination we encountered in EED, the underdetermination of PSGU may not be so problematic when using simple learning methods such as gradient ascent. The path of steepest ascent is generally leading to the nearest global maximum from the origin, which is the preferred maximum. The closer the maximum is to the origin, the smaller the weights are; we thus got to a distribution with a given variance with the smallest weights, which means that the largest portion of the contribution to the variance was accomplished by the inputs themselves. This in turn means that the weights convey the information of the most prominently present components in the input.

However, we would like to be able to rephrase our objective such that there is generally only one global maximum. We should limit the learning process

15 Note that PSGU does require these features to be uncorrelated in contrast to EED

such that the objective function excludes intuitively unwanted maxima. In section 4.5 we consider such a learning procedure.

4.5 GENERALIZED HEBBIAN LEARNING

In the previous sections we have seen two approaches to Spread Maximization: Eigenvolume Expansion-Dichotomization and Pre-Sigmoid Gaussian-Uniformization. For both we introduced an objective function which promotes the corresponding form of SM.

However, we've seen that both methods suffer from underdetermination. In this section we introduce a radical new approach in order to solve that problem. The technique explained here works for both variants of SM with different settings of just a single parameter. The approach is based on Hebbian learning, which was the subject of section 3.2.

We first look at the commonalities between dichotomization and uniformization in order to extract a common objective in section 4.5.1. We then transform the objective into an alternative objective, the constrained HO, which can be seen as analogous to it in some sense. In order to apply HO we transform the data and the network before we start the learning process. Afterwards we transform the weights of the network such that the network is optimal for the original objective and the unaltered weights.

We show how the approach works for simple, single layered Multilayer Perceptrons (MLPs) in section 4.5.1 to 4.5.2, which is based on techniques previously discussed in this thesis. After that we move on to derive a concrete technique for the approach for CNNs with a convolution and pooling layer in section 4.5.3 to 4.5.6. This concrete technique is called Convolutional Hebbian Algorithm (CHA)-SM.

In the manifestation of the approach for CNNs, we generalize many concepts previously discussed and apply these general forms to the concrete case of CNNs. We therefore introduce convolutional variants of many concepts and techniques previously discussed.

4.5.1 *Commonalities to Dichotomization and Uniformization*

Let's take a look at what binds the different forms of SM. Here we unify both into a single objective with different settings of parameters.

DECORRELATION We've seen that EED (section 4.3.2) was underdetermined; weight vectors which we would view as intuitively incorrect would result in the same values of the objective function as weight vectors which we would view as optimal. Any weight vector could be stretched such that the variance of the output neuron would be equal to the variance the neuron would have with the weight vector which we *would* view as optimal. Note

that while this optimal variance is infinite the reasoning presented here still holds.

We ran into a similar problem in PSGU (section 4.4.4). The objective function was underdetermined in that its function surface would contain global maxima which we wouldn't view as intuitively optimal. Any weight vector could be stretched such that the variance of the output neuron would be equal to the square of the optimal standard deviation σ^*.

Furthermore we have seen that both methods lead to decorrelated output. EED makes the eigenvolume grow toward the optimal hyperrectangle, which coincides with the output space itself.[16] Since the eigenvolume is defined in terms of the standard deviations in the eigendimensions, we know that at the optimum, the eigendimensions are aligned with the dimensions of the output space. Because data are always decorrelated in its eigenspace, we can conclude that dichotomy performs decorrelation in the output space as well.

PSGU leads to decorrelated output by definition, since the objective is to minimize the distance between the distribution of the current pre-sigmoid output and a Gaussian distribution with independent variables.

We can thus describe both methods as finding a mapping of the input to a decorrelated distribution with optimal variance, where the optimal variance is given by different values for the two methods. We would like this mapping to be based on the most prominent features in the input; we should solve the underdetermination such that the features extracted are most prominently present in the input.

We've seen in section 4.1 that gradient ascent is expected to result in the maximum which is closest to the origin for the specific objective function surfaces at hand. That maximum coincides with what we would intuitively view as the optimum, since the contribution of the input to the (optimal) output variance is greatest when the contribution of the weights is smallest. The maximum closest to the origin is therefore the maximum which is based on the most prominent input features.

In order to solve underdetermination, we therefore try to find a mapping *with the smallest weights* resulting in decorrelated output with a given variance. Note that whereas in PSGU the optimal variance is the variance of the pre-sigmoid output distribution, EED tries to maximize the post-sigmoid variance. However, maximizing the post-sigmoid variance is equivalent to maximizing the pre-sigmoid variance, since the sigmoid functions are monotonic. The same reasoning applies to decorrelation; in both cases the optimal pre-sigmoid output distribution is decorrelated and has a given variance, where the optimal variance depends on the kind of SM we perform.

The mapping of input to pre-sigmoid output is given by a linear combination of the inputs; the activation is just given by the weighted input. Mul-

16 Actually the sides of the eigenvolume is half the sides of the output space. However, the eigenvolume and the eigenspace is aligned and their centers coincide.

tiplying the weights therefore results in a multiplication of the pre-sigmoid output standard deviation. Finding the smallest weights which result in a given pre-sigmoid variance can therefore be performed by finding the largest variance for weight vectors with a given length. After we have found the weight vectors resulting in the largest variances, we can rescale the weight vectors such that their variances coincide with the optimal variance.

The question is how to find the weight vectors of a given length which results in the largest pre-sigmoid variance. For simplicity, let's constrain the weight vectors to a length of one. Our new goal, then, is to find a mapping which results in a decorrelated pre-sigmoid output distribution with largest variance, under the constraint that the weight vectors are of unit length.

CENTERING Another commonality between EED and PSGU is that both have optima where the output distribution is centered at the origin. In a global optimum of EED, the eigenvolume of the output distribution coincides with the output space. Since the middle of this space is the mean of the distribution, we know that the optimal output distribution has zero mean. As for PSGU, we just defined the optimal pre-sigmoid output distribution as having zero mean.

The goal of both methods can therefore be transmuted into a goal of finding weight vectors of unit length which cause the output distribution to be decorrelated and centered at the origin.

4.5.2 *Principal Component Analysis and Standardization*

The above goal is exactly what PCA accomplishes for a single layer MLP with linear transfer function.[17] (See section 3.1.2) PCA finds an orthogonal transformation such that the variance of the transformed data is maximized. This transformation consists of loading vectors which can directly be used as the weight vectors of the PCA.

We've seen that PCA can be performed by eigen decomposition of the covariance matrix of the input distribution. The covariance matrix of the (pre-sigmoid) output distribution is then given by formula 3.8, which states that:

$$\mathbf{\Sigma_Z} = \Lambda = \text{diag}(\boldsymbol{\lambda}) \tag{4.21}$$

where

- Λ a diagonal matrix containing the eigenvalues;
- $\text{diag}(\mathbf{v})$ denotes a diagonal matrix of elements of $\mathbf{v}$ in the diagonal;
- $\boldsymbol{\lambda}$ is a vector of the eigenvalues of $\mathbf{Z}$.

17 Note that centering is not a part of PCA.

The optimal covariance matrix and the optimal mean vector are given by:

$$\boldsymbol{\Sigma}^* = \sigma^* \mathbf{I} \tag{4.22}$$

$$\boldsymbol{\mu}^* = \mathbf{0}^N \tag{4.23}$$

In order to get to our optimal distribution we then transform the weights obtained by PCA. We first perform a standardization step which transforms the (pre-sigmoid) output distribution to a zero-mean, unit variance decorrelated distribution and then multiply by σ^*:

$$\begin{aligned} \widetilde{\mathbf{Z}} &= \left(\mathbf{X} - \mathbf{1}^N \boldsymbol{\mu}^\mathsf{T}\right) \mathbf{W} \sigma^* \boldsymbol{\Lambda}^{-1/2} \\ \widetilde{\mathbf{Z}}_{n\cdot} &= \mathbf{X}_{n\cdot} \left(\mathbf{W} \sigma^* \boldsymbol{\Lambda}^{-1/2}\right) - \boldsymbol{\mu}^\mathsf{T} \mathbf{W} \sigma^* \boldsymbol{\Lambda}^{-1/2} \end{aligned} \tag{4.24}$$

where

- $\widetilde{\mathbf{Z}}$ is the standardized output data;
- $\boldsymbol{\Lambda}^{-1/2}$ denotes the diagonal matrix of inverses of the square roots of the eigenvalues $\left(\boldsymbol{\Lambda}^{-1/2}\right)_{ii} = \frac{1}{\sqrt{\lambda_i}}$

Note that we divide by the square root of the eigenvalues, since the eigenvalues correspond to the output variance, instead of the output standard deviation.

Since for dichotomization the optimal variance is infinite, multiplying by σ^* would be an infeasible solution. Instead of multiplying by infinity, we change the transfer function from sigmoid to its hard counterpart, e.g., we replace the hyperbolic tangent by the signum function.

Formula 4.24 conforms to a single layer MLP with linear transfer function (formula 2.2). We convert the weights resulting from PCA by the above transformation in order to get the MLP which results in the optimal output distribution.

$$\mathbf{W} \mapsto \mathbf{W} \sigma^* \boldsymbol{\Lambda}^{-1/2} \tag{4.25}$$

$$\mathbf{b} \mapsto -\mathbf{W} \boldsymbol{\mu}^\mathsf{T} \sigma^* \boldsymbol{\Lambda}^{-1/2} \tag{4.26}$$

Since we can perform PCA by using the GHA, we have a full procedure to perform SM on single layer MLPs with linear transfer function. First we calculate and subtract the means from the data, which is a necessary preprocessing stage for the GHA. Then we train a single layer ANN without biases and with linear transfer function by using Sanger's rule on the preprocessed data. When the weights have converged we apply the above conversion using the means calculated in the preprocessing stage, which introduces biases. We then change the transfer function from linear to a sigmoid function and end up with the optimal post-sigmoid distribution for the original data.

Note that the introduction of biases in the above way causes the output distribution to have zero mean. We have therefore solved the mean centering problem discussed in section 4.3.3.

HEBBIAN OBJECTIVE We have seen in section 3.2.3 that the GHA can be derived from the Hebbian Objective (HO) with orthonormalization constraints. Now we consider applying the HO with orthonormalization constraints to ANNs which have different structures from the standard single layer MLP without biases or transfer function. Learning by optimizing the constrained HO constitutes Generalized Hebbian Learning (GHL).

Because we maximize the squared output, we are maximizing the variance of the output, which is proportional to it when the output data has zero mean. The constrained objective therefore causes the weight vectors to converge to unit length and orthogonal to each other while achieving the greatest output variance. We can generalize the above method for finding the weights which lead to the optimal output distribution for single layer networks by using the GHA to one for double layer CNNs with a convolution and pooling layer by deriving a concrete update rule from the constrained HO.

Similar to the method for SM which uses the GHA, we zero-mean the data beforehand and remove the transfer functions from the CNN. We then apply a Lagrangian derivation in order to incorporate the orthonormalization constraints in the objective function, with which we train the network. Afterwards, we apply the conversion to the weights which causes the output neurons to have the optimal variance and zero mean to get our final ANN.

4.5.3 *Pooled Convolutional Component Analysis*

We've seen that the constrained HO in the case of simple single layered MLPs reduces to the GHA, which leads to networks performing PCA. Since we now apply the constrained HO to CNNs, we introduce convolutional variants of these concepts: Convolutional Hebbian Algorithm (CHA) and Pooled Convolutional Component Analysis (PCCA). Just as the GHA causes the weights of a network to converge to the loading vectors which form the basis of PCA, the CHA can be said to cause the weights to converge to a configuration which forms the basis of PCCA.

However, PCCA is inherently different from PCA in some respects. While PCA considers the linear (orthogonal) features which result in the greatest variance, PCCA considers the pooled (orthogonal) features which result in the greatest variance, or equivalently, PCCA considers the linear (orthogonal) features which result in the greatest pooled variance. The analysis is of quite a different nature, because the pooling function is 'lossy'; the pooling function is inherently surjective, since its output remains the same under re-

ordering of its inputs. The fact that it is the analysis of *pooled* convolutional components means that this inherent difference is natural to PCCA.

Because the pooling operation is generally not a linear function, we cannot view the output space as a subspace of the input space onto which the input data is projected linearly. While PCA can be seen as a method to find the subspace such that the distance between the input data and the projected data is smallest, PCCA cannot be viewed analogously.

However, since both PCA and PCCA can be viewed as methods for finding the maximal variance of a transformation which uses an orthogonal projection, we find that the methods have enough in common so that we can call PCCA the convolutional variant of PCA.

In the next two sections we show how to perform PCCA. Before we move on to reduce the constrained HO to the CHA in section 4.5.5, we first need to deal with a problem which was dealt with by the CHA trivially, namely the fact that we need to have an output mean of zero if we want the constrained HO to perform variance maximization.

4.5.4 *Zero-mean Problem*

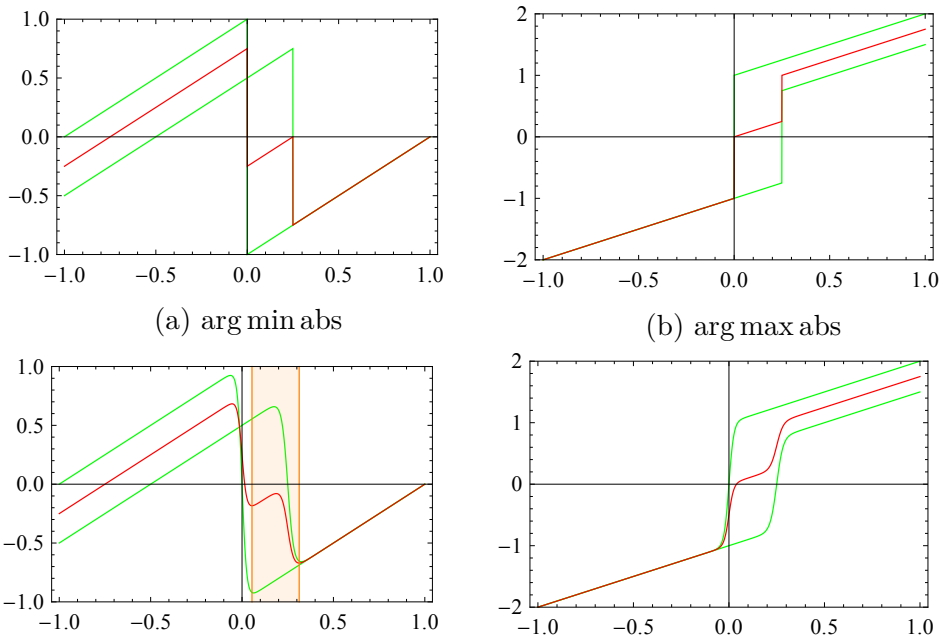
We've seen that when performing PCA by using the GHA, we make the data zero-mean beforehand and then perform the learning algorithm in order to find the components with maximal variance. The GHA can be seen as maximizing the square output, which is proportional to the output variance when the output has zero mean. Because the transformed input has zero mean, we could then ensure our output to have zero mean by keeping the bias term zero.

For PCCA the biases need to be treated differently; we should not simply disregard the biases as was the case when using the GHA for SM. The HO maximizes variance when the output mean is zero. Because PCA uses a linear transformation on the input data, we know that when the input mean is a zero vector, the output mean must also be a zero vector. This is not the case when the transformation that the ANN performs also includes a non-linearity, e.g., the pooling function.

We therefore include a bias b_k for each weight configuration k which causes the pooled outputs to have zero mean. The value of these biases should change during the learning process, since the output distribution changes as well.

The difficulty here, lies in the fact that there is no analytical solution of the bias such that it results in a zero output mean. In fact, it might be impossible to get an output mean of exactly zero. For example when a CNN is such that there is one output neuron, the data consist of a single data point and the convolution map of that data point contains both a positive extreme and a negative extreme, then any bias term would cause the absolute maximum pooling function to output either a positive or a

negative extreme, but in no way could it output zero. However, for larger data sets with some variance, half of the data could be transformed such that they output a positive extreme while the other half outputs a negative extreme, leading to a mean value closer to zero; even though we might be unable to get a mean of exactly zero, we can change the bias in order to get the mean closer to zero.



(a) arg min abs (b) arg max abs

(c) soft arg min abs (d) soft arg max abs

Figure 4.7: Graphs of pooled output and their average for values of the bias within $[-1, 1]$, for different pooling functions. In green the pooled outputs and in red the average output. The soft arg max abs and soft arg min abs pooling functions are given by formula 2.28 with softness parameter $p = 20$ and $p = -20$. The pools of the two data points consist of two outputs in the convolution layer with values $\{-1, \frac{1}{2}\}$ and $\{-1, 1\}$

We add an objective function which minimizes the distance between (the approximation of) the mean and zero, which is then optimized only with respect to the bias terms:

$$E(p_{l_f}^k) = -\left(\widetilde{\mu}_k\right)^2 \tag{4.27}$$

One might be inclined to perform gradient ascent by using backpropagation to get the derivative of this objective w.r.t. the biases for each pooled output $p_{l_f}^k$:

$$\frac{\partial E(p^k_{(x_\mathrm{f},y_\mathrm{f})})}{\partial b_k} = -\widetilde{\mu}_k \frac{1}{N} \sum_{x<\mathrm{w_p}, y<\mathrm{h_p}} \frac{\partial p^k_{(x_\mathrm{f},y_\mathrm{f})}}{\partial a^k_{(x_\mathrm{f}+x,y_\mathrm{f}+y)}} \tag{4.28}$$

where

- $\mathrm{w_p}$ and $\mathrm{h_p}$ are the width and height of the pool;
- N is the number of pooled output across the whole data set.

However, this approach might get stuck in local optima. In figure 4.7 we depict the average output of a neuron in the convolution map of an example CNN as a function of the bias. (See section 2.2.2 for definitions of the pooling functions.) We see that the hard and soft arg max abs pooling functions are monotonically increasing, so there are no local optima. Note that the hard arg max abs function contains discontinuities. Learning the bias terms with gradient ascent might therefore result in an oscillation between two sides of a gap, without converging.

The hard and soft arg min abs function, on the other hand, don't have this monotonicity property. Especially the soft arg min abs pooling function might cause convergence to local optima. Note that any local optimum of the mean as function of the bias corresponds to a local optimum of the squared mean—the objective function by which we change the biases, as given by formula 4.27. The orange coloured region in figure 4.7c is the region in which the gradient is such that gradient ascent converges to the local maximum at $b_k \approx 0.2$, at which point the average output is less than zero.

Instead of using gradient ascent, we provide a different, more simple approach. Note that the derivatives of the hard arg max abs and arg min abs are 1 at every point. When using 1 instead of the gradient of the pooling function, we won't get stuck in local maxima. We therefore take $\widetilde{\mu}_k$ as local objective/local error, instead of the actual partial derivative of the zero-mean objective w.r.t. the bias. These local objectives are then used by a learning mechanism such as gradient ascent when training the network.

CONVERGENCE One might wonder whether the learning mechanism which uses the above local objectives instead of the real partial derivatives converges at all, given that a positive bias update might result in a negative effect on the mean. Nevertheless, the update rule causes the mean to converge to zero, except when we use the hard absolute maximum pooling function, in which case it can keep oscillating. For the other pooling functions it holds that when the gradient and the update rule have a different sign we move away from a zero mean until we have reached a local optimum, after which we move further toward zero mean again. Eventually we end up with zero mean because in the limit where $\operatorname{abs} b_k \to \infty$ the mean as a function of b_k approximates a linear function and so it must intersect

the horizontal axis at which point the bias converges such that the mean is zero.

MEAN APPROXIMATION Since the output distribution changes during the learning process, so the mean changes as well. Instead of recalculating the output mean after each update, we approximate it. It is important that the approximation doesn't depend on the approximation of the mean in previous update steps, since the interaction between the approximation and alteration of the output mean can cause the biases to oscillate around a zero mean, without converging. Instead we use the output mean of the current batch, when using a batch learning mechanism, which in fact is the kind of learning mechanism we employ in the experiments. (See section 5.1.3.)

NON-DETERMINISM As can be seen in figure 4.7 the hard and soft absolute minimum pooling functions can have multiple settings of the bias such that the output mean is zero. This means that we are dealing with non-determinism; multiple bias settings lead to the output distribution with the required properties. Different weight initializations and different orders in which the input data is presented to the network may lead to different maxima and so the GHA might result in different ending states when applied to CNNs.

4.5.5 *The Convolutional Hebbian Algorithm*

Now that we have seen how we can keep the output means of a double layer CNN zero during learning (or at least how the network converges to such a state), we can complete the description of PCCA by showing how the CHA works.

In this section we show what update rule the CHA has and in section 4.5.6 we show how we can use PCCA in order to find the optimal weights for either variant of SM.

CONVOLUTION GHL for a CNN consisting only of a single convolution layer is quite straightforward. We've seen in section 2.2.1 that we can view convolution as the reiterated application of a fully connected MLP to different positions in the input image. The Lagrange multipliers are therefore solved in the same way as for the GHA (see section 3.2.3) and the partial derivatives of the HOs are given by Sanger's rule.

POOLING We've seen in section 4.4.2 that we can move the transfer function used in the convolution layer to the pooling layer, for most of the pooling functions described in section 2.2.2. Though this change doesn't lead to equivalent networks when using soft approximations to hard pooling functions, it still results in approximations to CNNs with hard pooling

functions. We can then change that output layer transfer function into a linear function in the same way as we did when using the GHA for SM.

$$p^k_{(x_\mathrm{f},y_\mathrm{f})} = s\left(Z^k_{(x_\mathrm{f},y_\mathrm{f})}\right) \tag{4.29}$$

$$Z^k_{(x_\mathrm{f},y_\mathrm{f})} = \left\{a^k_{(x_\mathrm{f}+x,y_\mathrm{f}+y)} \mid 0 \leq x < \mathrm{w_p} \wedge 0 \leq y < \mathrm{h_p}\right\} \tag{4.30}$$

$$a^k_{(x_\mathrm{p},y_\mathrm{p})} = \sum_{(x,y,z)<(\mathrm{w}_c,\mathrm{h}_c,N)} i^z_{(x_\mathrm{p}+x,y_\mathrm{p}+y)} w^k_{(x,y,z)} + b_k \tag{4.31}$$

where

- i, a and p are neurons representing input, convoluted activation and pooled output;
- $i^z_{(x,y)}$ is the input on location (x, y) in the map of feature z;
- $a^k_{(x,y)}$ is the activation and output on location (x, y) in the convolution map of neuron k;
- $p^k_{(x_\mathrm{f},y_\mathrm{f})}$ is the output on location (x, y) in the pooling map of feature k;
- N is the number of input maps, i.e. the number of input features;
- x_f and y_f are the coordinates of the final output neuron in the output map;
- $Z^k_{(x_\mathrm{f},y_\mathrm{f})}$ is the pool of neurons connected to the final output neuron;
- b_k is the bias of weight configuration k;
- $\mathrm{w_p}$ and $\mathrm{h_p}$ are the width and height of the pool;
- w_c and h_c are the width and height of the weight configuration k;
- $w^k_{(x,y,z)}$ is the weight on the connection from feature z at a relative location (x, y) for weight configuration k.

OBJECTIVE FUNCTION The output of a neuron in the pooling layer of such a CNN is then given by formula 4.29 to 4.31. The unconstrained objective function is given by formula 3.12. For readability we abbreviate indices (x, y) to a location index l. in that notation for CNNs, the objective function for a single neuron is given by:

$$E(p^k_{l_\mathrm{f}}) = \frac{1}{2}\left(p^k_{l_\mathrm{f}}\right)^2 \tag{4.32}$$

We can view the outputs of neurons at different locations in the output map as the outcomes of multiple evaluations of a CNN which would nor-

mally result in a single output. Just as we can view a convolution layer as performing a reiterated application of a smaller ANN to different locations in the input layer (section 2.2.1), we can view a whole CNN as such. Every neuron at a given location in the pooling map (indirectly) receives information from a subframe of the input image shifted by an amount proportional to the location in the pooling map.

We can therefore view the outputs at a given location in the pooling map as the only outputs of another network applied to a subframe of the original input. The structure of the output layer of such a network then corresponds to the output layer of the network used for the GHA. We therefore constrain the objective function per output map location instead of the total objective function $W(L_\mathrm{O})$.

In much the same way as the derivation of the GHA which used Lagrangian multipliers (section 3.2.3), we introduce constraints and their λ's sequentially for the different features evaluated at a given position in the output map. While the objective function for the neuron employing the first feature at a given location we only have the normalization constraint, while subsequent features also get orthogonalization constraints. We therefore have separate Lagrangians for each output neuron.

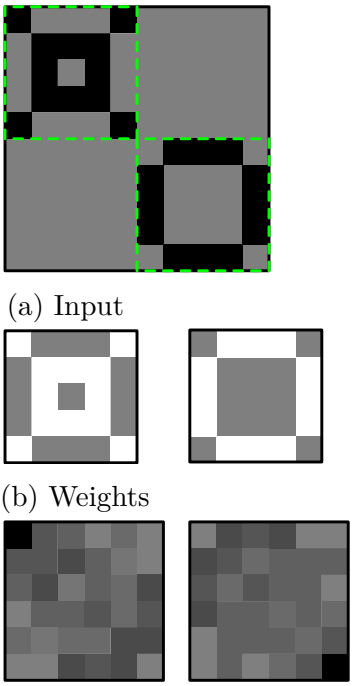
In appendix C.19 we derive that the partial derivative of the Lagrangian of a neuron $p^k_{l_\mathrm{f}}$ w.r.t. a weight on a connection (indirectly) connected to that neuron is given by:

$$\begin{aligned}\frac{\partial L_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial w^k_{(l_\mathrm{i},z)}} =& p^k_{l_\mathrm{f}} \sum_{l_\mathrm{p}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} i^z_{l_\mathrm{f}+l_\mathrm{p}+l_\mathrm{i}} \\ & - p^k_{l_\mathrm{f}} \sum_{j\leq k} w^j_{(l_\mathrm{i},z)} \sum_{l_\mathrm{p}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^j_{l_\mathrm{f}+l_\mathrm{p}}\end{aligned} \tag{4.33}$$

When the CNN is viewed as reiterated application of a CNN with a single output, the above formula gives rise to the weight update when using gradient ascent. Otherwise the weight update consists of more than a single partial derivative, since we sum over all output neurons. Because a weight from a weight configuration k is only connected to a neuron in a pool associated with k and that pool is the only input to an output neuron in pooling map k, we only have partial derivatives w.r.t. a weight from k of Lagrangians for neurons in the same pooling map k. The partial derivatives of the constrained objective function over the whole output layer are therefore given by:

$$\frac{\partial E(L_\mathrm{O})}{\partial w^k_{(l_\mathrm{i},z)}} = \sum_{p^k_{l_\mathrm{f}}\in Z_k} \frac{\partial L_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial w^k_{(l_\mathrm{i},z)}} \tag{4.34}$$

Performing gradient ascent on this objective function then gives rise to the learning mechanism we call the CHA.



(a) Input

(b) Weights

(c) Convolution maps

Figure 4.8: The input, with gray value corresponding to zero, and black corresponding to input of -1. In green we show that the weight vector is evaluated at non-overlapping subframes of the input. The weight vectors, where the light gray corresponds to a value of $\frac{1}{\sqrt{12}}$, such that the length of the weight vectors are 1. The two convolution maps containing the outputs of the convolution layer, where gray values correspond to values between -1 and 0.

REDUCTION TO THE GHA Note that this reduces to the GHA when the pool is reduced to a size of one. When $\forall Z_{l_f}^k \subset Z^k : \left|Z_{l_f}^k\right| = 1$ then $w_p = h_p = 1$ and so $Z_{l_f}^k = \left\{a_{l_f}^k\right\}$. Suppose the pooling function then reduces to the linear function $s(\{a\}) = a$. [18] Formula 4.33 therefore reduces to:

$$\begin{aligned}\frac{\partial L_{(l_f,k)}(\mathbf{w}_k,\Lambda_k)}{\partial w^k_{(l_i,z)}} =& p^k_{l_f} i^z_{l_f+l_i} - p^k_{l_f} \sum_{j\leq k} a^j_{l_f} w^j_{(l_i,z)} \\ =& a^k_{l_f} i^z_{l_f+l_i} - a^k_{l_f} \sum_{j\leq k} a^j_{l_f} w^j_{(l_i,z)}\end{aligned} \tag{4.35}$$

Note that this reduced form conforms to Sanger's update rule (formula 3.13). The reduction reinforces the statement that a convolution layer can be viewed as reiterated application of a simple MLP, because the derivative of the constrained objective function consists of the sum of the derivatives of the constrained objective functions for all locations in the output maps.

Local Maxima

It is important to note that the objective function contains local maxima in which the learning rule of the CHA can get stuck. We give an example in which the CHA is shown to be stuck.

18 For all pooling functions considered in section 2.2.2 this in fact holds.

Consider a CNN consisting of a convolution layer and a pooling layer with one pooled output. The single pool contains at least two neurons which receive input from disjoint sets of input neurons. Lets suppose the input is (10×10) and the weight configuration is (5×5) which makes the pool contain neurons from a frame of size (6×6). Let's use the linear transfer function and the absolute maximum pooling function.

Suppose the input looks like figure 4.8a. We consider two features which look like a crosshair and a circle, given in figure 4.8b. These result for the given input image in the convolution maps given in figure 4.8c.

The input data is similar to the input shown in the figure, except that they are inverted for half of the images in the data set. Another exception is that in some images the upper left corner doesn't have a crosshair; all input values there are zero instead. Note that the variance of the upper left neuron in the convolution map of the crosshair is largest and the variance of the lower right neuron in the convolution map of the circle is largest.

Furthermore, these activations are always the greatest in an absolute sense and so the pooled output is given by the value of the upper left or the lower right neuron of the convolution layer. The output variance is therefore given by the variance of the upper left and lower right pixel in the convolution maps. Since the crosshair doesn't occur in all images, the output variance associated with the circle is greater.

Suppose we have a CNN with only a single weight configuration, corresponding to the crosshair. We would like to see this feature converge to the circle, since that feature results in larger output variance. This is not the case, however.

For the given pooling function, the derivative of the pooled output w.r.t. a neuron in the convolution map is zero at all places except the upper left corner, where it is one. The partial derivative of the constrained objective of the CHA w.r.t. a weight therefore reduces to formula 4.35, which conforms to Sanger's update rule. The weight configuration therefore converges to the first loading vector of the upper left sub-frame of the input image, which is the crosshair. We have thereby shown that the feature doesn't converge to the feature which would result in the largest output variance, namely the circle; the first feature therefore doesn't necessarily converge to the first *convolutional loading*. [19]

We therefore have to train several CNNs with the CHA and see which one extracted the best *convolutional components*. [20]

Pooled Convolutional Component Analysis

Now that we know what update rule the CHA has we can combine it with the zero-meaning method in order to perform PCCA. When we update the

19 A convolutional loading is to PCCA what a loading vector is to PCA.
20 A convolutional component is defined analogously to a principal component.

biases during the learning process, by applying the zero-meaning method after each iteration we ensure that when the weights have converged, the output mean is zero. In that case the CHA has maximized the variance of the pooled convolutional components, since the squared output (HO) is proportional to the variance, when the output mean is zero. The combination of the update rules of zero-meaning and the CHA therefore causes the network to eventually perform PCCA.

PCCA coincides with the reformulated objective of SM. We can thus use the method described above for SM. Next we show how we can use the weights obtained by performing PCCA in order to find the weight configurations which lead to the optimal output distribution in terms of either variant of SM.

4.5.6 *Standardization*

Now that we have described how to perform PCCA, we can show how to use it to perform either variant of SM. We standardize the output data and scale it so that each dimension has the optimal variance, which is different for either method of SM. We incorporate this transformation in the weights and biases of the network. Also the zero-meaning operation which was performed before applying the CHA is incorporated.

RESCALING Whereas the variance of the output neurons of an ANN used by the GHA were equal to the eigenvalues of the covariance matrix, we don't have an analytical way to find the variances of the output neurons of a CNN used by the CHA. We therefore just compute the empirical standard deviation $\widetilde{\sigma}_k$ of each neuron k.

We then transform the pooled outputs $p_{l_\mathrm{f}}^k$ such that their standard deviation is equal to the required standard deviation σ^*:

$$p_{l_\mathrm{f}}^k \mapsto \frac{\sigma^*}{\widetilde{\sigma}_k} p_{l_\mathrm{f}}^k \tag{4.36}$$

For dichotomization the optimal standard deviation was infinite; instead of multiplying by the standard deviation we just replace the sigmoid transfer function by its hard counterpart. For example, we switch the hyperbolic tangent to the signum function.

Note that for all the hard pooling functions discussed it holds that $cs(X) = s(cX)$. We can thus push the rescaling constant $\frac{\sigma^*}{\widetilde{\sigma}_k}$ down the formula for the CNN at hand (formula 4.29 and 4.31) until we reach the bias and weighted

input. The transformation is therefore equivalent to multiplying the weights and bias by the rescaling constant:

$$\begin{aligned} w^k_{x,y,z} &\mapsto \frac{\sigma^*}{\widetilde{\sigma}_k} w^k_{x,y,z} \\ b_k &\mapsto \frac{\sigma^*}{\widetilde{\sigma}_k} b_k \end{aligned} \tag{4.37}$$

The above equivalence of transformations doesn't hold when using a soft pooling function which uses the soft arg max function given by formula 2.28. This is so because for small values the soft approximation to the hard arg max function becomes less accurate. In appendix C.20 we show that by changing the hardness of the approximation while multiplying the input, we can get a similar result as for hard pooling functions: $cs(X) \mapsto s(cX)$ when $p \mapsto \widetilde{p}$.

However, this would lead to different hardness settings for the different pooling functions on the different feature maps. We therefore leave p the way it is when standardizing the output. We expect this not to have a large impact on the output distribution since both settings of p lead to a function which is a soft approximation to the hard arg max function. We therefore just use the transformation given by formula 4.37.

SHIFTING Recall that we had subtracted the mean $\boldsymbol{\mu}_{\mathrm{i}}$ from the input data before we applied the CHA. We therefore have to incorporate this shift in inputs into the weights of the CNN if we want to use it on the unaltered data. The convoluted outputs for the shifted input data can be rewritten to convoluted outputs of the unaltered input data:

$$\begin{aligned} a^k_{(x_{\mathrm{p}},y_{\mathrm{p}})} &= \sum_{(x,y,z)<(\mathrm{w}_c,\mathrm{h}_c,N)} \left(i^z_{(x_{\mathrm{p}}+x,y_{\mathrm{p}}+y)} - \mu^z_{\mathrm{i}}\right) w^k_{(x,y,z)} + b_k \\ &= \sum_{(x,y,z)<(\mathrm{w}_c,\mathrm{h}_c,N)} \left\{i^z_{(x_{\mathrm{p}}+x,y_{\mathrm{p}}+y)} w^k_{(x,y,z)} - \mu^z_{\mathrm{i}} w^k_{(x,y,z)}\right\} + b_k \end{aligned} \tag{4.38}$$

where

- μ^z_{i} is the mean of input feature z;
- other variables are explained in formula 4.29 to 4.31.

We therefore transform

$$b_k \mapsto b_k - \sum_{(x,y,z)<(\mathrm{w}_c,\mathrm{h}_c,N)} \mu^z_{\mathrm{i}} w^k_{(x,y,z)} \tag{4.39}$$

after we have performed the transformation which leads to the optimal standard deviation. Combining these transformations, gives us the transformation which forms the standardization step:

$$\begin{aligned} w^k_{x,y,z} &\mapsto \frac{\sigma^*}{\widetilde{\sigma}_k} w^k_{x,y,z} \\ b_k &\mapsto \frac{\sigma^*}{\widetilde{\sigma}_k} \left(b_k - \sum_{(x,y,z)<(\mathrm{w}_c,\mathrm{h}_c,N)} \mu^z_{\mathrm{i}} w^k_{(x,y,z)} \right) \end{aligned} \tag{4.40}$$

SM

Now that we have shown how to convert the weights obtained from PCCA into weights which result in the optimal output distribution for either variant of SM, we have completed our general method for performing SM. First we make the input data zero mean, just as we did when applying the GHA. Then we use the learning mechanism which consists of both the zero-meaning technique and the CHA. After the learning phase, we perform a standardization step which causes the output distribution to have the required variance for each output neuron. Remember that this variance was the optimal pre-sigmoid variance; as a final step we reintroduce the transfer function in order to obtain our final CNN which maximizes spread.

5

EXPERIMENTATION

Now that we have described and argued for methods to perform Spread Maximization (SM), we evaluate their behavior empirically. This chapter covers empirical experiments performed to assess the performance of the various unsupervised learning methods.

In order to compare the methods *as unsupervised learning techniques* we evaluate them as feature extractors in section 5.3; to this end we assess the performance of a classifier which receives the extracted features as input. The classifier used is a single layered Convolutional Neural Network (CNN), so that the whole model consists of one multi-layered CNN.

In section 5.4 we evaluate the unsupervised learning techniques as *pre-training techniques*. Such evaluation differs from the evaluation of the methods as feature extraction techniques in that the unsupervisedly trained layers are subsequently trained by the supervised learning mechanism.

In the first section we describe which experimental setup was used and in what way the techniques were employed; section 5.2 considers experiments performed on a simple synthetic dataset in order to show the basic workings of the methods considered. Sections 5.3 and 5.4 cover a wide range of experiments to evaluate the unsupervised learning techniques as feature extraction and as pre-training techniques.

5.1 EXPERIMENTAL SETUP

This section covers the specifics of the experiments performed. Section 5.1.1 describes the datasets on which we trained the Artificial Neural Networks (ANNs). In 5.1.2 we describe what hyperparameters we have used in testing the methods, i.e. the parameters of the learning mechanism and network structure. Section 5.1.3 covers method specific hyperparameter settings and implementation details.

5.1.1 *Datasets*

We have tested SM on a data set commonly used in computer vision. The majority of the tests have been performed on the MNIST dataset. As a basic starting point however, we've performed experiments on a synthetic data set.

The data has been scaled such that the values of individual pixels lie within the interval $[-1, 1]$, coinciding with the range of the hyperbolic tan-

gent function. No other preprocessing of the data has taken place. While preprocessing can greatly improve the performance of CNNs[11], we limit our research purely to the proposed methods and comparisons to alternatives such as Pooling Convolutional Auto-Encoders (PCAEs).

Dataset	#Categories	#Training Images	Image Size	Input form
AZ	2	144	10×10	Grayscale
MNIST	10	60000	28×28	Grayscale

Table 5.1: Basic properties of the datasets used.

Figure 5.1: Example images from the synthetic AZ data set. Blue pixels signify values of zero.

Figure 5.2: Example images from the MNIST data set. Near blue pixels signify values near zero.

AZ We've constructed a simple synthetic dataset called AZ which is similar to the one described in section 4.5.5. The dataset consists of images of size (10×10) which contain one of two patterns of size (5×5), which can be viewed as the letter A and the letter Z. The background consists of zero values on which either pattern is inserted positively or negatively at different positions in the image—see figure 5.1. This amounts to 144 images. From these images, 44 were randomly taken as test set, while the rest was used for training.

MNIST The Mixed National Institute of Standards and Technology (MNIST) dataset consists of grayscale images of handwritten digits, which have been size-normalized and centered in a fixed size image [24]—see figure 5.2. The ten classes of digits are quite easily separable, as can be concluded from the fact that CNNs have been reported to produce error rates on the test data between 0.21% and 1.7% [24], [1], depending on the network structure and learning techniques used. The dataset therefore serves as a good benchmark with which we can easily test our unsupervised learning techniques.

5.1.2 *Training Hyperparameters*

For all of the methods we've used in experiments, we computed the derivatives of the objective function associated with the method w.r.t. the parameters of the model, e.g., the weights. We then use either momentum or gradient ascent in order to determine the weight updates. The momentum parameter α was chosen to be 0.9, which is standard[18]. The η parameter (learning rate) of both momentum and gradient ascent depends very much on the dataset and method used. Settings of that parameter which we used in our experiments ranged from 10^{-2} to 10^{-5}.

Instead of calculating the derivative of the objective over the whole dataset, we approximated the true gradient by considering the derivatives of the objective only for a mini-batch of data samples. The size of these mini-batches was chosen to be 24.

The experiments for evaluating the methods as feature extraction methods share the setting of having 20 features. This number is quite low compared to the number of hidden units of ANNs commonly found in literature. Mind that for a convolution field size of (7×7), the input dimensionality is 49; the number of features is roughly half the number of principal components, which is equal to the input dimensionality for convoluted Principal Component Analysis (PCA).[1] Given the intricate relationship between PCA and Convolutional Hebbian Algorithm (CHA) based SM, 20 features seems a good setting.

Weights were initialized by random values sampled from a univariate Gaussian distribution with standard deviation 0.1, though biases were initialized to zero.

No stopping criterion has been used. Instead, a fixed number of iterations were used to train the ANNs. Based on some preliminary testing, we concluded that a number of iterations in the order of 10^3 can already produce fairly good features. We chose settings of the number of iterations around the order of 10^4, so that the learned features have had a chance to converge better. For unsupervised learning we used 1×10^4 iterations, while for supervised learning we chose 2×10^4 iterations

We used the hyperbolic tangent transfer function because of symmetry considerations covered by section 2.1.3. For the pooling function we experimented on two different functions: a hard pooling function and the soft approximation to it; we used the arg max and soft arg max-function with $p = 1$ and $f(x) = \operatorname{abs} x$ so that positive outputs weren't favoured over negative ones. Note that arg max abs-pooling introduces jump discontinuities in the objective function, which might be harmful for the training mechanism used. In the following we indicate the hard pooling function with the code HP and its soft approximation with SP.

1 Section 2.2.1 describes how any method can be used in a convoluted fashion.

We have also experimented with using weight decay in the supervised learning phases. The weight decay regularization term has been added to the cross entropy error function in some of the experiments. Experiments which make use of weight decay are tagged with the code WD, while experiments without are marked with X. The λ setting[2] of the weight decay regularization term was chosen to be 10^{-3}, which seemed to perform well based on some preliminary experiments.

Network Structure

We have tested each method by training the first layer of a CNN unsupervisedly, after which we added a classification layer which is trained by minimizing cross entropy. The first layer was trained with convolution fields of size (7×7) and pool of either size (7×7) or of size (2×2). Recall that in section 4.2 we have argued for using the same convolution field size as pool size when performing SM, while (2×2)-pooling is standard.

After having trained the first convolution layer the pool size was set to (2×2), trading (partial) translation and deformation invariance for spatial accuracy. Instead of training a fully connected classification layer on top of the first layer, we reintroduced some translation invariance by taking a pool size of (2×2) and a convolution field size one pixel smaller than the width and height of the pooling maps of the first layer.[3]

Consider input data samples of size (28×28). A convolution field size of (7×7) leads to convolution maps of size (22×22). When pools have size (7×7) as well, we end up with pooling maps of size (3×3), meaning that we disregard the last pixel in every column and row.[4] When pools have size (2×2), as is the case in the second learning phase, the pooling maps have size (11×11). The classification layer then has convolution field size of (10×10), such that its convolution maps have size (2×2), which leads to outputs of size (1×1) for a pooling field size of (2×2). The output maps of the classification layer should always be of size (1×1), so that we can apply the softmax activation function to all features on the single location.

The network structure of the resulting network can be represented by the following code:

$$20@[7\rangle 2\}]; 10@[10\rangle 2\}] \tag{5.1}$$

The layers are separated by semicolons and each layer $n@[\mathrm{w_c}\rangle \mathrm{w_p}\}]$ is to be read as a (double) layer consisting of n features, a convolution field size of $(\mathrm{w_c} \times \mathrm{w_c})$ and pooling field size of $(\mathrm{w_p} \times \mathrm{w_p})$.

2 See section 2.3.4.

3 Note that a CNN layer with convolution field size equal to the size of its input maps and pool size equal to (1×1) is equivalent to a fully connected Multilayer Perceptron (MLP) layer.

4 Note that for PCAEs the pooling contribution maps always have the same size as the convolution maps.

LENET-1 A more elaborate network structure is used in experiments covered in section 5.4. The structure is the one of a classic CNN example known as *LeNet-1*. Its structure can be represented by the code[25]

$$4@[5\rangle 2\}]; 12@[5\rangle 2\}]; 10@[4\rangle 1\}] \tag{5.2}$$

This structure is considerably more complex than the simple network structure described above.

Note that while the architecture seems more complex the number of parameters (weights) is quite a bit smaller; while LeNet-1 has about 3.000 parameters, the simple network structure has about 20.000 parameters. Looking at the number of parameters in the non-classification layers alone, however, the numbers are given by approximately 1200 for LeNet-1 and 1000 for the simple architecture. The fact that these numbers are comparable means the workload of the unsupervised learning phases of both architectures is comparable. The fact that the number of parameters in the classification layer of the simple network structure is quite large means that training might get stuck in worse local optima more easily which might result in a higher variance of the error rates of the networks.

For unsupervised pre-training of this network structure we again considered using pool sizes different than the eventual (2×2). For the both the first and second layer we used pools of size (4×4) while using convolution field size (5×5). The pools were chosen to be of a slightly different size than the convolution fields so that the network wouldn't suffer from unconsidered edge cases (see section 2.2.1).

HYPERPARAMETER SPACE In the experiments performed we consider different hyperparameter settings as described above. An experiment performed either uses weight decay (WD) or not (X); uses the hard arg max abs-pooling function (HP) or its soft approximation (SP); uses the simple network structure or LeNet-1 and for the simple network structure an experiment performed can use unsupervised training on a layer structure with large pools $(7\rangle 7\})$ or small pools $(7\rangle 2\})$.

Furthermore, we have tested the unsupervised learning techniques as feature extraction techniques and as pre-training technique. These translate into supervised training of the classification layer alone and supervised training of the whole network. We also consider using both; the supervised training of the classification layer alone can be seen as supervised pre-training of the classification layer preceding the supervised training of the whole network. While the former two are indicated with PRE and POST, the latter is indicated with PRE+POST.

In order to obtain reliable statistics on the performance of the ANNs, we tested each setting eight-fold, where each rerun had a different weight initialization.

An exhaustive search through the hyperparameter space proved unfeasible given the computational resources available to us. Some combinations of settings have therefore not been tested.

5.1.3 *Methods*

In this section we describe what settings we have used in experiments for the various methods and which algorithms we used to achieve the objectives of the unsupervised learning techniques.

DICHOTOMIZATION BY EIGENVOLUME EXPANSION For Eigenvolume Expansion-Dichotomization (EED) we used the simplified objective as described in section 4.3. However, at certain points in the objective function surface, the derivatives can be abnormally high, in such a way that it is difficult to choose a setting for the learning rate η. Choose the η too high and training might shoot over a large chunk of the objective function surface. Choose it too low and training might take unreasonably long to get past shallow regions in the objective function surface.

We therefore use a learning mechanism which limits the vectors of gradients for each feature to unit length, which we call *limited gradient ascent*. When the gradient calculated is given by a larger vector, we rescale it so that we know that the largest possible step size is as large as η, which we chose to be 10^{-3}. This procedure is very ad hoc, however; one might think it to be better to limit the vector of all gradients instead of gradient vectors of each feature, for example. Note also that the limit is quite arbitrarily taken to be unit length. Further research could provide better ways of dealing with the irregular objective function surfaces at hand.

The learning mechanism used for these experiments is thus a limited variant of gradient ascent, in accordance with section 4.3.

Since the objective surface of EED does not have any global optimum in finite space we convert the network obtained after we stop the learning phase. Instead of stretching the weight vector to infinity, the hyperbolic tangent transfer function is replaced by the signum function, which is equivalent. The dichotomization is thus finalized by performing binarization manually.

Weights were initialized around the origin randomly sampled from a normal distribution with standard deviation 10^{-2}. Initialization closer about the origin might be more appropriate, given the reasoning in section 4.3 on convergence to a global optimum 'close' to the origin. However, we found that for very low weight initialization, the derivatives start out so low that the learning process takes too long to get started up.

PRE-SIGMOID GAUSSIAN-UNIFORMIZATION For Pre-Sigmoid Gaussian-Uniformization (PSGU) we also used limited gradient ascent, with learning rate $\eta = 10^{-3}$. However, in contrast to EED we have initialized the weights

by sampling from a normal distribution about the origin with standard deviation 10^{-3} instead of 10^{-2}. Though the gradients are very large near the origin, limited gradient ascent only takes steps of size η or smaller, so that we don't have to abstain from low initialization.

CHA BASED UNIFORMIZATION Recall that CHA based SM consists of Generalized Hebbian Learning (GHL) applied to CNNs, with a second objective which causes the mean output of each feature to converge to zero. The objective function is therefore two-fold; one term of the objective function is used to compute the partial derivatives at the proper weights (i.e. not the biases), while the other is used to compute the partial derivatives at the biases. We therefore used two different learning rates η. While the η^{w} of the main objective was chosen to be 10^{-5}, the η^{b} used for updating the biases was chosen to be 10^{-3}, based on some preliminary testing. The higher learning rate for the biases was due to the fact that the partial derivatives w.r.t. the proper weights can be orders of magnitude greater than the partial derivatives w.r.t. the biases.

The learning mechanism used to determine the weight updates was momentum; the momentum parameter α was 0.8.

We observed after some preliminary experiments that while the first couple of weight vectors converge quite quickly, the latter take quite a bit longer to converge. This is due to the fact that latter pooled convolutional components depend on all former ones and so do their derivatives. In order to speed up the process we mapped the weight vectors to the feasible space at the middle iteration, i.e. when the learning phase was at 50% we perform Gram-Schmidt orthonormalization.

When the learning phase has finished, we again performed an orthonormalization step, because numerical instability and the stochastic nature of using mini-batches to approximate the true gradients may cause the learning phase not to converge to solutions in the feasible space.

We then transformed the weights of the network as described in section 4.5.6. The transformation depends on the standard deviation, which we estimate from 10^4 samples.

Because the soft arg max pooling function is non-linear, the resulting features might still have different output variance than the optimal. We therefore apply an iterative process, which minimizes the squared difference between the optimal standard deviation and the actual standard deviation as measured on the current mini-batch. In each iteration the length of the weight vectors was assigned a derivative proportional to $(\sigma^* - \tilde{\sigma})$ and momentum was used in order to compute the weight updates, i.e. the weight vectors were scaled such that the difference in vector length coincided with the $\Delta \|\mathbf{w}\|$ update given by the momentum learning mechanism. For this process we used half the number of iterations used in the main training phase, i.e. 5×10^3 and $\eta = 2 \times 10^{-4}$.

Because of the stochastic nature of this learning phase, caused by using mini-batches to approximate the true standard deviation, we apply another weight transformation based on estimations of the standard deviations again from 10^4 samples.

The above orthonormalization and standardization step may cause the output means of the features to diverge from zero by a rather large amount. We therefore performed a separate learning phase in which we solely trained the biases. This phase consists of half the iterations of the main learning phase, though a stopping criterion is used; when the vector of Exponential Moving Averages (EMAs) of the output features has length smaller than 10^{-4} we stop adjusting the biases.[5]

Finally the signalling function is changed from a linear function to the hyperbolic tangent function.

Pooling Convolutional Auto-Encoder The PCAEs were trained on the same network structure settings as CHA based SM. We tested the denoising variant of PCAEs, with 30% binomial noise, meaning that 30% of pixels is randomly turned fully on (1) or fully off (-1), depending on the outcome of a fair coin toss. 30% noise is an amount of noise commonly used for autoencoders and specifically the amount of noise used by the inventors of PCAEs themselves—Masci et al.[27].

The learning rate η was chosen to be 10^{-5}, since the partial derivatives w.r.t. the weights may be very large at the start of the learning phase.

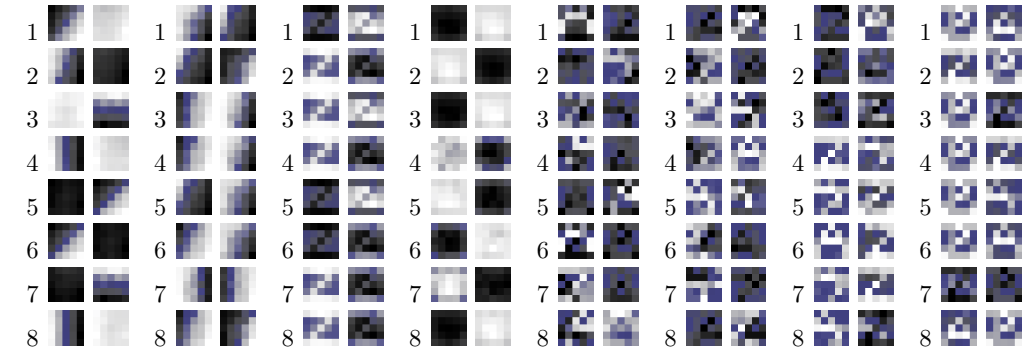
5.2 synthetic data

For the synthetic az data set, we used a layer structure given by $2@[5\rangle 6\}]$. Note that $5\rangle 6\}$ leads to output maps of size (1×1), meaning that a single neuron in the output map is connected to all input image pixels. The hope for any of the unsupervised learning methods considered is to extract the letter A and Z as the two features; hence the number of features.

weight images In the following we depict weight configurations by images of which the pixels' Red Green Blue (RGB) values correspond to the real values of the proper weights (excluding the biases). Each input map in the weight configuration is converted to a single image. Generally we depict values between -1 and 1 on a scale from black to white. Because our eyes can deceive us[6], we have depicted values close to zero as blue, so that it is easier to see what pixels correspond to nearly absent weights. Beyond the interval $[-1, 1]$ the values also get coloured. From $w \to \infty$ the depicted colour gets closer to green, while for $w \to -\infty$ the colour gets redder. However, because it is foremost the relative size of the weights which is of interest, we have

5 The smoothing factor of the EMAs was set to 0.9.

6 Context influences the perception of luminance change as shadow[37].



(a) SP EED (b) SP PSGU (c) SP CHA (d) SP PCAE (e) HP EED (f) HP PSGU (g) HP CHA (h) HP PCAE

Figure 5.3: Weight vector images of the features extracted from the synthetic AZ dataset for eight different runs of each setting.

rescaled the weight vectors such that the maximum weight is transformed to 1 or the minimum is transformed to -1 in such a way that all weight values fall between -1 and 1.

In the remainder of this section we describe the extracted features on a visual basis. Quantitative assessment of the methods is performed in the remainder of this chapter.

5.2.1 *Extracted Features*

We have performed experiments on the synthetic dataset with each of the unsupervised learning techniques and both pooling function settings. In figure 5.3 we present the weight vectors thus extracted. Note that the features extracted by CHA-based uniformization are the same as the features extracted by CHA-based dichotomization; binarization doesn't change the direction of the weight vector. Also the weight decay setting doesn't change features extracted in the unsupervised learning phase, since we only experimented on weight decay in the supervised learning phases. We thus present a total picture of the weight vectors in the figure.

The first thing we note is that the soft pooling function seems to be counterproductive for EED and PSGU (see figures 5.3a and 5.3b). This might be caused by the fact that for small input values, the soft pooling function approximates average pooling. Since weights are initialized close to the origin the outputs of the convolution layer are also small, and thus the inputs to the pooling function are small. Because the derivatives of the objective are backpropagated more homogeneously over the visual field the features become smooth gradients in the initial part of the learning phase. It seems that this initial phase causes EED and PSGU to propel the weight vectors to features which do not capture important aspects of the input data.

PCAEs seem to be unable to make good use of the soft pooling function (see figure 5.3d). Recall that the PCAEs are *denoising* autoencoders. Again this might be related to the fact that the soft pooling function approximates the average pooling function for small input values.

The features extracted by CHA-SM (figure 5.3c) with soft pooling seem quite a bit better. In each run we see that the first feature encodes for a Z, while having zero weights elsewhere. It should be noted that the training data contains more Zs than As and the converse holds for the test data. The second feature seems less helpful, since it also looks like a Z; however, there the pixels belonging to the Z are close to zero while the rest is all positive or negative. These features can be seen as coding for anything-besides-a-Z, which codes for As as well as off-center Zs.

The case is different when using the hard pooling function (see figures 5.3e to 5.3h). Each method seems to be able to capture the two patterns quite well. However, in many cases the patterns are not centered in the visual field of the features. For example, the A in the first run of EED (see figure 5.3e) is shifted one pixel up compared to the optimal pattern coding for an A. This seems to be a recurring problem when using a hard pooling function. Since the soft pooling variant of CHA-SM doesn't seem to suffer from the same problem, we are inclined to conclude that soft pooling might have advantages over hard pooling.

We also see that some features contain information of both an A and a Z. Take for example the weight images from the sixth run of EED; it can be seen as containing a white A shifted up by one pixel, and a black Z which is centered. Though different from what we had intended the features to be, such features do seem to code for patterns in the data efficiently.

5.3 FEATURE EXTRACTION

This section is on the unsupervised learning techniques as feature extraction techniques. It covers experiments performed using the simple network architecture and supervised learning by backpropagating only to weights within the classification layer. The resulting performance statistics then serve as proxy of how well the extracted features work independent of how they are actually used. We discuss the features extracted for each method, before we move on to the performance statistics.

5.3.1 *EED*

The experiments on EED show a great difference between the different settings. While training with a pool size of (2×2) results in highly localized features, training with a pool size of (7×7) results in features which seem to capture properties which cover a larger portion of the visual fields. In figure 5.4 we depict weight images resulting from the setting combinations.

Again soft pooling seems to have an adverse effect on the features extraction, though the features depicted here are not as bad as the features extracted from the synthetic data set (see figure 5.3a). Especially the setting 7⟩7} with SP seems to result in more reasonable features.

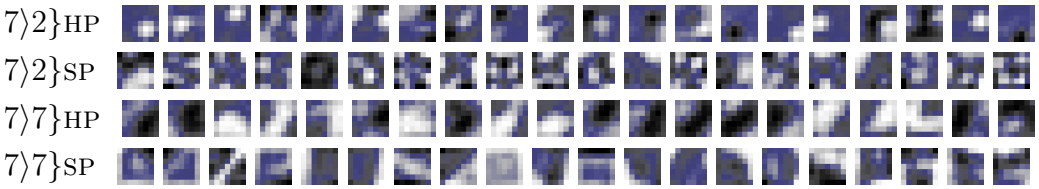


Figure 5.4: Features extracted by EED using various settings.

5.3.2 *PSGU*

The experiments performed with PSGU also show a great difference between the features extracted when using pool size (2×2) and pool size (7×7). In figure 5.5 we depict features extracted under the different setting combinations. We see that all weight images obtained when using the former setting consists of a set of small dots scattered differently across the visual field. These features seem ineffective in capturing any recurring properties of the dataset.

The features extracted under the other setting seem a lot better. However, some features (for example the first and fifth in the bottom row of figure 5.5) code only for a couple of pixels at the edge of the visual field.

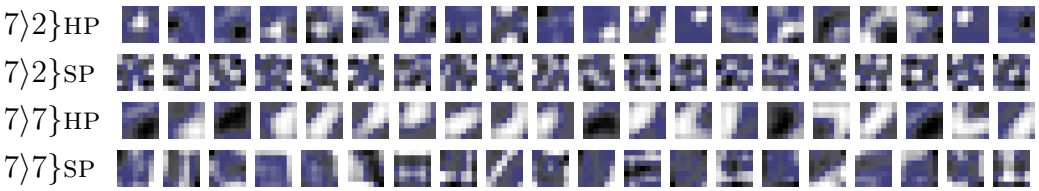


Figure 5.5: Features extracted by PSGU under various settings.

5.3.3 *CHA-based Spread Maximization*

The features extracted by the CHA-based SM techniques have the same topology irrespective of whether we use the dichotomization or uniformization variant. The only difference is a multiplication factor or equivalently a change of transfer function. Features extracted using CHA-SM are depicted in figure 5.6. For weight images of all runs, see appendix D.1.

The features extracted by different runs under different settings are quite similar. The first couple of features are most informative and quite intuitively graspable. The first weight image is often uniformly black or white; the second is a horizontal contrast, though slightly tilted (perhaps because of *italics*); the third is a vertical contrast orthogonal to the preceding contrast and the forth is a thick diagonal line. The first feature acts as a simple downscaling of the image, while the second and third are basic edge detectors.

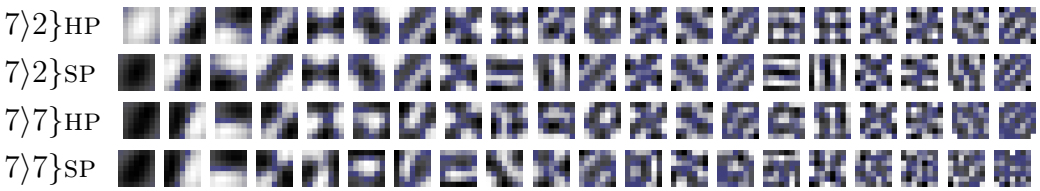


Figure 5.6: Features extracted by CHA-SM under various settings.

FEATURE ORDER From figure 5.7 one can see that preceding weight vectors have smaller lengths while subsequent weight vectors get larger and larger. This is caused by the fact that, after having converged to the the feasible space in which all weight vectors are of unit length, the weight vectors are divided by the standard deviation of their output. We can thus see that the features are neatly ordered by output variance because of the nature of CHA.

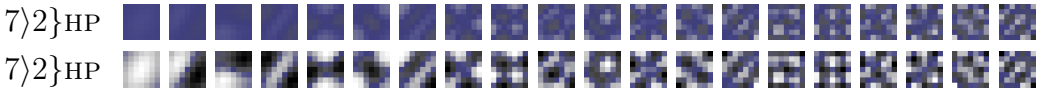


Figure 5.7: Non-scaled (upper) and scaled (lower) weight images of a run of CHA-SM under settings 7⟩2} and HP.

SEMI-DETERMINISM Note that nearly equivalent features are produced by the other runs of CHA-SM with the same setting. This is caused by the fact that for pooling sizes closer to (1×1), CHA produces results closer to the loading vectors of PCA, which is a deterministic procedure. The global optimum at which each pooled convolutional component has maximal variance seems to be reached by many of the runs with pool size (2×2)

Note that the equivalence at hand precludes negation; the weight configurations are nearly equal or each others inverse. We therefore see the same weight images but with inverted gray values. The first weight image of figure 5.7 is equivalent to the first weight image of figure 5.8, for example.

LOCAL OPTIMA While training with pool size (2×2) results in a nearly equivalent set of features each time we run the algorithm, the same cannot be said when using pool size (7×7). In accordance with section 4.5.5, we find that for some initializations the training methods gets stuck in a local optimum where the first feature isn't the feature with highest variance. One can see in figure 5.8 that the bottom run has the first two features switched compared to the run depicted at the top.

The use of large pools also seems to cause later features to act more erratic. The latter weight images in figure 5.8 don't seem to have as much topological structure as the weight images in figure 5.7; they look rather noisy. It might be the case that these features have converged to a suboptimal local optimum, though it might also be the case that these features have more difficulty converging because of the erratic nature of stochastic learning mechanisms when using mini-batches. The gradual orthonormalization to ever changing features might aggregate in the non-convergence of latter features.

DATA DEPENDENCE Note that the features presented in figure 5.7 seem quite rigid; they do not seem to capture inherent features of the MNIST dataset. One might expect similar features when applying the method to another dataset of grayscale images.

One property which does seem specific to the dataset at hand is the tilt at which the features are presented. The contrasts of the second and third feature might have another angle for a different dataset perhaps. Except for that property, the features seem an intuitive systematic enumeration of possible topological arrangements even when taking the actual dataset used not into consideration.

The case is different for the features extracted with pool size (7×7). There we see features encoding for some curvatures which are specifically common in the MNIST dataset (see figure 5.6). From the fourth feature on we see some features which deviate from the rigid pattern we see in the features extracted under the small pool size.

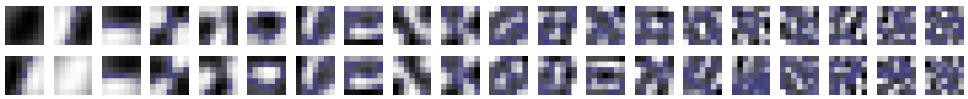


Figure 5.8: Scaled weight images of two arbitrary runs of CHA-SM using settings 7⟩7} and SP.

5.3.4 *Pooling Convolutional Auto-Encoders*

Features extracted using Pooling Convolutional Auto-Encoders are depicted in figure 5.9. The upper rows depicts weight images obtained by training a

PCAEs with pooling operations applied to fields of size (7×7), while the bottom rows depicts weight images for pool size (2×2). For weight images of all runs, see appendix D.1.

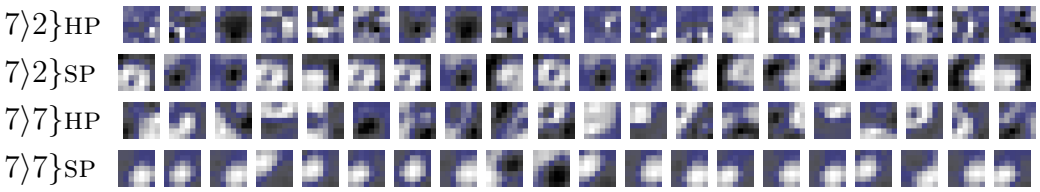


Figure 5.9: Features extracted by PCAEs using various settings.

LOCALITY Note that quite some features obtained by PCAEs trained with a pool field size of (2×2) and hard pooling function are highly local; they only have distinct properties for a small region within the visual field and have small weights elsewhere. Take for example the first couple of features of the bottom row in image 5.9.

Due to the small pool size a large amount of outputs in the coding layer contributes to reconstructing the same patch in the input image. The features can therefore permit to code for only a small patch in the visual field. However, these features do not seem to code for informative or intuitive properties of the input; instead of coding for properties of pen strokes they code for incidental value of some input pixels.

REDUNDANCY Note that other features are not as local, coding for a large white or black spot within the visual field. In spite of the fact that these do seem useful features, we see the PCAEs contain other very similar features. Especially the 7⟩7} SP seems to suffer from this problem, though the 7⟩2} seem to suffer from the problem as well, but to a lesser degree. This near duplication of features makes for an inefficient coding.

Supervised Training

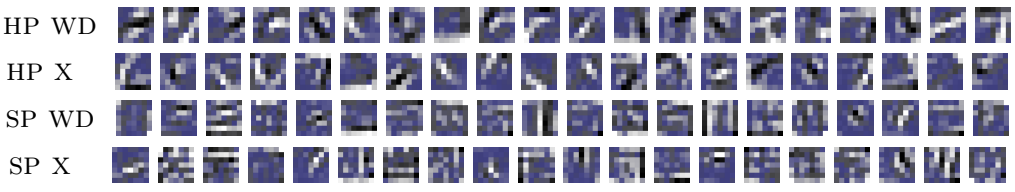


Figure 5.10: Features extracted by supervised training under various settings.

For completeness we also consider the features extracted when only using supervised training for the simple network structure. In figure 5.10 we

depict the weight vectors from the first layer when performing supervised training with several settings. Since these experiments didn't make use of unsupervised pre-training, it doesn't make sense to apply the $7\rangle 7\}$ setting. However, we do consider using weight decay when performing supervised training.

We see that the visually most pleasing features are extracted when using the hard pooling setting. Most features seems to code for a line segment adjacent to pixels of opposite value while the rest of the visual field is assigned small weights. There doesn't seem to be a large difference when using weight decay or not.

The features extracted when using the soft pooling function seem less effective in capturing the patterns in the input data. Quite some features contain a straight white line next to a black one. This is related to the fact that we use pools of size (2×2). When a convolution neuron with such a feature outputs a high positive value, the other convolution neuron in the pool along the direction of the white line probably also outputs a high positive value, while the other two neurons in the pool probably outputs a high negative value, because of the black line next to the white one. Because the output values are then of the same magnitude, the pooling function has partial derivatives w.r.t. each neuron of the same magnitude. The pattern in the feature is thus likely to be reinforced in such a case.

For supervised learning without pre-training, soft pooling doesn't seem to provide an advantage.

5.3.5 *Performance*

Now that we have described the extracted features qualitatively, we assess them quantitatively. We estimate the usefulness of the extracted features by the performance of some classifier which receives input from the unsupervisedly learned layer. The classifier we used is again a CNN, so that the combination of the feature extraction layer and the classification layer becomes one large CNN. However, only the classification layer is trained, since only the classifier is fitted to the output of the feature extraction layer.

The in this section reported error rates of this classifier are a proxy to the performance of any network of which the first layer is initialized using one of the methods considered. Since a network which uses the features merely for pre-training changes the features during supervised learning, we expect the error rates reported in this section to be higher than the error rates of state-of-the-art CNNs which use pre-training.

The difference in error rates for nearly each pair of settings has been shown to be statistically significant by Welch's t-test[36], where $p < 0.05$ implies significance. This is no surprise given that the standard deviation in error rate of nearly each setting is close to 1. Without presenting the results of each t-test, we provide the reader with the rule of thumb that when the

difference in the average error of two settings is larger than one, they have been proved to have significantly different performance. For interesting edge cases deviant of this rule we explicitly inform the reader of the significance.

In appendix D.2 we list statistics of all experiments performed. However, because the large table found there can obscure the data, we present small subsets of the data in small tables in the following. The cells in these tales which are left empty correspond to settings which haven't been experimented on due to insufficient computational resources, while cells marked with '-' correspond to combinations of settings which haven't been experimented on due to principled reasons.

We have depicted the average percentage of misclassification over the eight runs as well as the misclassification rate of the best performing network of any given hyperparameter setting. Both statistics provide valuable information on the performance of a type of ANN. The average error gives insight in how well a single run can be expected to perform, while the error of the best performing network gives insight into what error can be achieved after multiple reruns.

Note that the average error can occasionally be quite misleading. Keep in mind that the results are the end state of a function optimization algorithm. An objective surface can be such that the learning mechanism easily converges to a quite bad local optimum, while other times it finds a far better local optimum. Judging a method by its average performance alone would therefore be unfair.

When using neural networks in practice it is common to do several reruns and take the best performing network based on its performance on some test set or on some other choosing criterion. Although we use the same test set for choosing the best rerun as for evaluating the performance of the network, the statistic denotes the performance of a type of ANN for the optimal choosing criterion.[7]

Results

In table 5.2 we present the error rate statistics for the different methods. We see that for EED the soft pooling function seems to have an adverse effect, increasing the error rate by a staggering 40% approximately. We also note that performing unsupervised learning on a pool size equal to the convolution field size decreases the performance of EED by a couple of percents, contrary to our expectations.

The case is converse for PSGU. The soft pooling function and larger pool size do improve the performance by a couple of percents. Furthermore we

7 While the optimal choosing criterion is purely hypothetical, we suppose that some fairly simple choosing criterion can be found which picks the actual best performing network out of eight reruns or otherwise picks a network with roughly the same classification error.

		7⟩7}		7⟩2}	
		avg	best	avg	best
EED	HP	8.4	7.88	6.89	6.55
	SP	51.57	48.84	48.35	46.94
PSGU	HP	4.7	3.51	7.57	5.17
	SP	3.27	3.1	4.81	4.58
CHA-D	HP	18.05	15.11	19.65	18.5
	SP	80.1	73.26	48.35	46.94
CHA-U	HP	7.79	3.82	7.57	5.17
	SP	3.77	2.72	3.95	3.23
PCAE	HP	19.12	11.37	78.76	74.4
	SP	3.7*	3.7*	2.29*	2.29*

Table 5.2: Average and best error rates associated with methods as feature extraction techniques (PRE) on network architecture **Simple**. (*The experiments on PCAEs with soft pooling function have only been performed once.)

see that PSGU outperforms EED, which is in line with our expectation that the binarization effect of EED is harmful.

The same holds for CHA-based dichotomization; the performance of CHA-D is excessively worse than the performance of CHA-U, for any setting of pool size and pooling function. Soft pooling also has a positive effect on CHA-based uniformization, cutting the error rate roughly in half. Though the 7⟩7} setting seems to outperform the 7⟩2} setting, the difference in performance between the two has not been shown to be statistically significant ($p \approx 0.4$). This is not surprising, given that the features extracted by either setting look quite similar.

Using larger pool sizes during training improves the performance of PCAEs quite much when using the hard pooling function. However, when using the soft pooling function we get the best results: an error rate of 2.29% when using the 7⟩2} setting. Soft pooling seems to greatly increase the performance of a PCAE

	7⟩7}		7⟩2}	
	avg	best	avg	best
BIN	8.4	7.88	6.89	6.55
x	5.14	3.71	5.5	4.38

Table 5.3: Average and best error rates of EED with the hard pooling function setting (HP) as feature extractor (PRE).

Non-binarized EED

Since binarization has proved to have an adverse effect on the CHA-SM, we also tested EED without performing binarization; we performed experiments in which after the iterative training procedure ended the hyperbolic tangent transfer function was not replaced by the signum function. Note that this inevitable means the weights of the networks are not near any global optimum in the objective function surface, since EED doesn't have any global optima within finite space. The ending point of the iterative procedure is therefore at an arbitrary place on an ever climbing slope in the objective function surface, determined by the number of iterations and step size of the learning mechanism.

The statistics on the performance of the networks thus obtained are given in table 5.3. We see that leaving the output non-binarized improves the performance by a couple of percents for both pool size settings. This shows that there may be a level of dichotomy less than fully binarized which constitutes a better unsupervised learning criterion; perhaps this lesser level of dichotomy coincides with uniformity, or it may be in between.

Comparison

Our best feature extractor seems to be CHA-based uniformization on a CNN layer with the soft pooling function and 7⟩7} setting, based on the fact that this method produced a network which achieved an error of 2.72, which is the lowest error rate among all settings of our methods. PSGU under the same settings comes in a close second. Note however, that the average performance of PSGU is greater than CHA-U. This discrepancy is explained by the fact that the standard deviation of the performance of PSGU is rather small (0.15) compared to that of CHA-U (1.02). A choosing criterion over the multiple reruns of a given method may therefore prove CHA-U to perform better. However, we haven't found any significant difference in performance between the sets of reruns in themselves ($p = 0.11$).

PCAEs achieved even a lower error rate of 2.29%, based merely on a single run (due to insufficient computational resources). Based on these preliminary findings one would like to conclude that our unsupervised learning techniques do not outperform conventional unsupervised learning techniques for convolution and pooling layers. However, a more sophisticated view emerges when assessing the performance of the unsupervised learning techniques as pre-training methods.

5.4 PRE-TRAINING CNNS

In this section we take a look at the performance of the various methods as pre-training methods. The features extracted by unsupervised earning techniques were subsequently used merely as initialization of the first layer

of a larger CNN. We have performed some such experiments on the simple network architecture, and others on the LeNet-1 network architecture, which is described in section 5.1.2.

KNOWN RESULTS The LeNet-1 network architecture is a fairly simple multi-layered CNN which has already been experimented on in literature[25]. It has been reported to achieve an error rate of 1.7%, which is quite low. The performance of the methods we experimented on should be tested against this statistic. Note that LeCun et al. did perform some preprocessing on the data.

Note that several other network architectures have been proved to work better for MNIST. For example LeNet-5, which is similar to LeNet-1 except that two fully connected non-convolutional layers have been added on top; this network architecture is reported to achieve an even lower error rate of 0.95%. Other techniques left unconsidered by this thesis are preprocessing techniques, which destort the input data in some ways; such techniques may further cut the error rate roughly in half[34]. However, such techniques are beyond the scope of this thesis.

BINARIZATION The experiments covered by this section disregard binarization. Binarization is the replacement of the transfer function of a CNN pre-trained by either EED or CHA by the signum transfer function. However, the derivatives of the signum function are always zero, which means that the weight updates in the corresponding layer are zero as well.[8] The performance statistics of these methods reported in the previous section can therefore not be improved by using the extracted features for initialization. This section therefore only covers the uniformization variant of CHA-SM and the variant of EED which doesn't use binarization.

		Simple		LeNet1	
		avg	best	avg	best
HP	WD	8.03	6.95	6.31	5.98
	x	4.76	4.45	5.67	4.88
SP	WD	12	9.26	1.77	1.44
	x	10.91	7.56	6.39	4.87

Table 5.4: Average and best error rates of CNNs trained without unsupervised learning (none).

8 Except in case we use weight decay, which causes all weights to converge to zero.

SUPERVISED LEARNING Before we examine the performance of the unsupervised learning techniques we discuss the baseline which results from supervised training alone. The performance statistics are listed in table 5.4.

For the simple network architecture the best performing network (4.45%) is obtained when using the hard pooling function and not using weight decay. When using the hard pooling function weight decay significantly impoverishes the performance.

For the LeNet-1 network architecture the best setting is the one which used the soft pooling function and weight decay resulting in error rates as low as 1.44%, which is even lower than known result of 1.7%. This might be due to the fact that in our experiments the hyperbolic tangent function was used as transfer function or that the hard pooling function was the symmetric arg max abs function instead of the asymmetric max function.

Given the fact that for the simple network architecture the hard pooling function and no weight decay performs significantly better and the fact that for hard pooling the simple network architecture performs significantly better it might come as a surprise that the overall best setting is given by the LeNet-1 architecture with soft pooling and weight decay, but the converse holds as well. This shows that there are local optima in hyperparameter space.

5.4.1 *Unsupervised Pre-training Performance*

Here we review the performances of CNNs which have been pre-trained using the unsupervised learning techniques. Due to the large number of possible setting combinations we first review some settings which seem disadvantageous to all unsupervised learning techniques. After that we can downsize the number of settings in order to produce smaller and more comprehensible tables of error rates.

			avg	best
CHA-U	HP	WD	6.16	4.9
		x	5.67	4.88
	SP	WD	9.01	6.48
		x	6.39	4.87
PCAE	HP	WD	17.1	8.69
		x	10.66	9.05

Table 5.5: Average and best error rates of CHA-based uniformization and PCAEs as pre-training techniques (POST) on **LeNet-1** showing the influence of weight decay.

WEIGHT DECAY In table 5.5 we list some error rate statistics related to using weight decay when weight vectors have been pre-trained using a supervised learning technique. Note again that in appendix D.2 the results of all experiments performed are listed. While purely supervised learning clearly benefits from using weight decay we can conclude that weight decay significantly harms the performance of pre-trained CNNs.

We suppose this result holds as well for the other unsupervised learning methods and have not performed experiments using weight decay on EED and PSGU.

		Simple				**LeNet-1**	
		7⟩7}		7⟩2}		5⟩4}	
		avg	best	avg	best	avg	best
EED	HP	4.31	2.98	5.94	5.71	7.1	5.57
	SP	1.94	1.57	1.47	1.31	2.15	1.66
PSGU	HP	4.87	3.54	5.89	5.52	7.4	5.91
	SP					2.31	2.02
CHA-U	HP	5.19	3.65	8.75	5.4	5.67	4.88
	SP	3.78	2.95	4.43	3.86	6.39	4.87
PCAE	HP					10.66	9.05
	SP	4.3*	4.3*	3.23*	3.23*	6.67*	6.42*

Table 5.6: Average and best error rates methods as pre-training techniques (POST) without using weight decay (X). (*The experiments on PCAEs with soft pooling function have only been performed once or twice.)

PRE-TRAINING WITHOUT WEIGHT DECAY In table 5.6 we list the error statistics of various hyperparameter settings of the unsupervised learning techniques as pre-training techniques without weight decay. Note that not all hyperparameter settings have been tested due to insufficient computational resources—especially for the simple network architecture.

The method performing best on MNIST turns out to be EED for the simple network architecture with soft pooling function and pre-trained on pools of size (2×2), resulting in a CNN which achieves an error rate of 1.31%. However, experiments on PSGU with the soft pooling function have not been performed. Given that we cannot prove PSGU and EED to perform significantly different on LeNet-1 with soft pooling, we cannot prove that EED outperforms PSGU.

Note that the 7⟩2} setting significantly outperforms the 7⟩7} setting for EED ($p = 0.003$). We conclude that using a larger pool size has a negative effect on this method.

The lowest error rate which EED achieves is significantly lower than the lowest error rate which is achieved by supervised training alone ($p = 0.007$). We can thus conclude that unsupervised pre-training by Spread Maximization (SM) can improve the performance of CNNs.

Furthermore each of our unsupervised learning techniques with any setting outperforms PCAE when using the hard pooling function. However, we have also enriched PCAEs with the introduction of a soft pooling function, which decreased their error rate down to 3.23%. Still, each of our unsupervised training techniques has some setting performing better than the best setting of PCAEs.[9] We can therefore conclude that we have invented valuable unsupervised learning techniques which may outperform existing ones.

		Simple PRE				**LeNet1**			
		7⟩7}		7⟩2}		PRE		POST	
		avg	best	avg	best	avg	best	avg	best
EED	SP	1.6	1.56	1.97	1.59	4.9	4.52	2.15	1.66
PSGU	SP	3.27	3.1	4.81	4.58	7.68	6.94	2.31	2.02
PCAE	SP	3.7*	3.7*	2.29*	2.29*	6.34*	5.48*	6.67*	6.42*
CHA-SM	SP	3.77	2.72	3.95	3.23	11.88	9.02	6.39	4.87
PSGU	HP	4.7	3.51	7.57	5.17	19.42	17.61	7.4	5.91
EED	HP	5.14	3.71	5.5	4.38	23.37	20.36	7.1	5.57
CHA-SM	HP	7.79	3.82	7.57	5.17	8.66	8.21	5.67	4.88
PCAE	HP	19.12	11.37	78.76	74.4	15.76	13.67	10.66	9.05

Table 5.7: Average and best error rates sorted on 7⟩7} PRE, showing the appropriateness of using PRE as a proxy to POST when not using weight decay (X). (*The experiments on PCAEs with soft pooling function have only been performed once or twice.)

FEATURE EXTRACTION PERFORMANCE AS PROXY TO PRE-TRAINING PERFORMANCE While the previous section assessed the unsupervisedly extracted features as features in themselves (unsupervised learning as feature extraction), in this section we view the extracted features merely as initial approximation to the final features (unsupervised learning as pre-training). Table 5.7 compares the error rate statistics obtained when the unsupervised learning techniques are viewed as feature extraction techniques and when viewed as pre-training techniques. When considering the order of

9 Note that this comparison might slightly favor our unsupervised learning techniques, since the experiments on PCAEs with soft pooling function consist only of one or two reruns, depending on the network structure.

the first two and last two columns of this table, we see that the PRE performance works quite well as a proxy of how well the type of extracted features works in a POST setting; the order of the column of average error rates for LeNet-1 POST loosely corresponds to the order of the column of average error rates for the simple network architecture with setting 7⟩7} PRE.[10] We can conclude that the valuation of the extracted features in themselves is a fine indication of how well the final features perform.

SUPERVISED PRE-TRAINING Recall that the CNN classifier used to assess the performance of the unsupervised learning techniques as feature extraction techniques forms one large CNN together with the unsupervisedly trained layer. The supervised training of the classification layer alone can be seen as a supervised pre-training technique. A CNN then gets initialized with the features extracted by the unsupervised learning techniques and the weight vectors resulting from the classifier. Another way of looking at it is to see it as a three step process:

1. Use unsupervised learning to determine the features in the first layers of the CNN;

2. Perform supervised learning while keeping the unsupervisedly trained layers locked;

3. Train the whole CNN supervisedly, unlocking all layers.

We expect CNNs to benefit from such unsupervised pre-training, because the final supervised learning phase might change the features extracted by unsupervised learning; when this learning phase would take of from randomly initialized weights in the classification layers the unsupervisedly learned feature might change quite much before the classification layer starts to settle into a weight configuration. This might change the features in such respect that some of the work performed by the unsupervised learning might get undone. Supervised pre-training causes the weights of the classification layer to be tuned to the extracted features, so that the final learning phase doesn't change the first layers' features based on the random configuration of the classification layer.

Table 5.8 shows error rate statistics related to such supervised pre-training. The experiments using supervised pre-training are tagged PRE+POST. Supervised pre-training has not been tested on all methods due to insufficient computational resources.

One can tell from the table that supervised pre-training can significantly increase the performance of a CNN (the difference between the average error when performing CHA-U with hard pooling function is not statistically

10 The largest deviation from the order is CHA-based uniformization, which performs rather better than what would be predicted based on the simple PRE performance.

			PRE+POST		POST	
			avg	best	avg	best
CHA-U	HP	WD	6.53	5.55	6.16	4.9
		x	5.56	4.93	5.67	4.88
	SP	WD	4.35	3.87	9.01	6.48
		x	4.35	3.87	6.39	4.87
PCAE	HP	WD	6.16	5.35	17.1	8.69
		x	8.74	8.04	10.66	9.05
	SP	WD				
		x	4.86*	4.39*	6.67*	6.42*

Table 5.8: Average and best error rates on network architecture **LeNet-1** showing the influence of supervised pre-training. (*The experiments on PCAEs with soft pooling function have only been performed twice.)

significant). We conclude that supervised pre-training is a great way to supplement unsupervised pre-training.

	Simple					**LeNet-1**			
	settings				error	settings			error
EED	non-BIN	SP	7⟩7}	PRE	1.3	non-BIN	SP	POST	1.66
PSGU		SP	7⟩7}	PRE	3.1		SP	POST	2.02
CHA-SM	U	SP	7⟩7}	PRE	2.72	U	SP	POST	4.87
PCAE		SP	7⟩2}	PRE	2.29		SP	PRE	5.48
none	X	HP			4.45	WD	SP		1.44

Table 5.9: Settings and error rate of the best performing network per method per network architecture when considering only PRE and POST.

OVERVIEW OF RESULTS In table 5.9 we list the best performing run out of all settings for each method when not considering supervised pre-training. Figure 5.11 shows the same error rates in a visual way. The reason we do not take into consideration experiments which did use supervised pre-training is that these settings haven't been experimented on for all unsupervised learning techniques due to insufficient computational resources.

Note that EED has produced the lowest error rate outperforming purely supervised training. Unsupervised pre-training does not seem to increase the performance when using the LeNet-1 network architecture. However, the converse holds for the simple network architecture. Spread maximization does outperform PCAEs on LeNet-1, but the same does not hold for the simple network architecture.

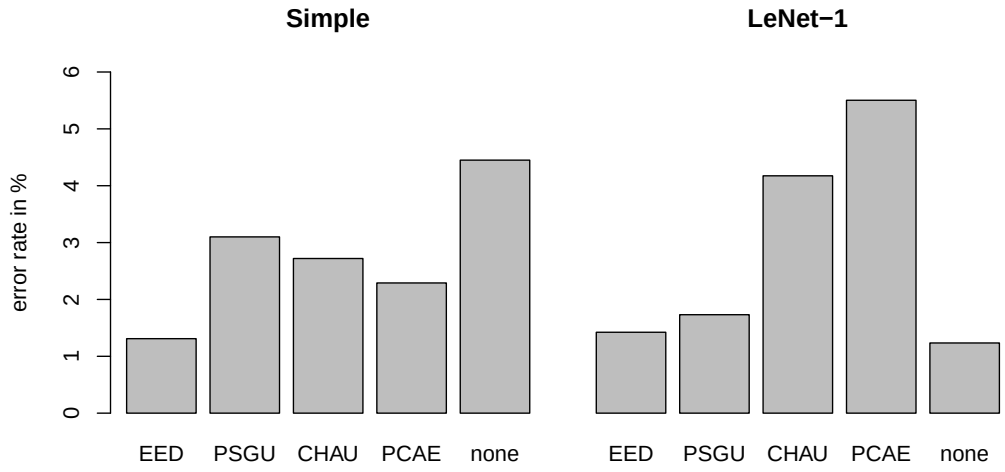


Figure 5.11: Error rate of the best performing network per method per network architecture when not using weight decay (x) and considering PRE and POST.

The non-binarized version of EED and the uniformization variant of CHA-SM outperform their counterparts, confirming our hypothesis that the binarizing effect of dichotomization is disadvantageous. Furthermore we see that nearly all of the best performing setting combinations in the table contain the soft pooling setting, leading us to conclude that soft pooling forms a valuable alternative to the standard hard pooling functions. using larger pool sizes during supervised training seems to benefit all of the SM techniques, but not PCAEs, which is in line with our predictions.

6

CONCLUSION

We now come to the conclusion of our thesis. We first present an overview of the material presented in the previous chapters. Section 6.1.1 discusses the results and what they mean for the relevance of our methods for the field of unsupervised learning of Artificial Neural Networks (ANNs). The last section (6.2) discusses the limitations of our research and experiments and present possible future research.

6.1 OVERVIEW

In this thesis we've described the workings of standard Multilayer Perceptrons (MLPs) and of Convolutional Neural Networks (CNNs) specifically in chapter 2 and supplemented these with new types of pooling function. We've reviewed some established methods for the unsupervised training of ANNs in chapter 3. The Generalized Hebbian Algorithm (GHA) forms an important learning mechanism for ANNs and its optimal weight configuration for a simple type of ANN forms the basis of Principal Component Analysis (PCA).

In chapter 4 we have introduced a new unsupervised learning paradigm, based solely on the output distribution: Spread Maximization (SM). In order to solve the underdetermination the objective introduces, we augment SM with the goal of keeping the weight vectors low. We have also introduced the idea of performing unsupervised learning on CNN layers with greater pool sizes than the CNN layer in which the extracted features are eventually used.

SM has been interpreted in two ways: dichotomization and uniformization. While dichotomization causes a network to eventually perform binarization, meaning that the output data points lie within the corners of the output space, uniformization causes the output data to be scattered across the output space uniformly.

We've introduced three methods for performing SM: one specifically for dichotomization, one specifically for uniformization and one which can be used for both. The first method, Eigenvolume Expansion-Dichotomization (EED), performs dichotomization by maximizing the eigenvolume, the size of which is equal to the determinant of the covariance matrix of the output distribution. This objective is maximized by replacing the sigmoid transfer function with a hard transfer function, which causes the output to be binarized. The second method, Pre-Sigmoid Gaussian-Uniformization

(PSGU), performs uniformization by minimizing the distance between the pre-sigmoid output distribution and an optimal uncorrelated Gaussian distribution with the same variance in each dimension.

The final method, Convolutional Hebbian Algorithm (CHA)-SM for performing either kind of SM makes use of the Hebbian Objective (HO). Applied to a simple form of ANN the HO reduces to the GHA, which performs PCA. Applied to CNNs it reduces to the CHA, which performs Pooled Convolutional Component Analysis (PCCA). When the GHA training phase is finished, we transform the network so that it performs SM. This transformation differs depending on whether we perform dichotomization or uniformization.

In chapter 5, we have compared our methods to Pooling Convolutional Auto-Encoders (PCAEs), which is similar in spirit to SM applied to CNNs with pooling in that both incorporate the pooling function in learning the weights of the convolution layer. We have compared the features resulting from each of the methods qualitatively and quantitatively; furthermore we have given an indication into how well these features perform on themselves and how well CNNs perform which have been initialized with the features. These two types of assessment of the features can be combined, giving rise to supervised pre-training of the classification layer.

6.1.1 *Discussion*

We've seen in chapter 5 that a soft and symmetric pooling function can greatly increase the performance of CNNs which don't make use of unsupervised learning and it can greatly increase the performance of the features extracted by unsupervised learning techniques; we have improved on PCAEs with the use of a soft pooling function.

Performing unsupervised learning on pools of greater size than the ones eventually used has shown to improve performance for all SM techniques. It also has the further benefit of reducing the output dimensionality, leading to faster computations. This technique can be seen as supplement to the proposed SM techniques and not as much to unsupervised learning in general, since PCAEs did not seem to benefit from it.

Supervised pre-training has proved to be a valuable technique to complement unsupervised pre-training when pre-training deep CNNs. The same cannot be concluded for the simple network architecture, since the necessary experiments have not all been performed due to a lack of computational resources. However, because without supervised pre-training the final supervised learning phase would start of changing the supervisedly learned features influenced by the random initialization of the classification layers, we suspect supervised pre-training to also be of benefit for more simple network architectures.

The inherent binarizing nature of dichotomization has proved to be disadvantageous; binarizing the output decreases the performance. We therefore considered a variant of EED where the transfer function is not replaced by the hard signum function. Though performing such EED means there is no principled stopping criterion (since there is no global optimum in finite space), the given settings with which we have tested it result in CNNs having the best performance of all experiments. This form of EED applied to a simple network architecture outperforms purely supervised training of the same network architecture, as well as the more complex LeNet-1 architecture. It therefore seems that this form of SM can greatly improve CNNs. However, we have experimented on a very limited set of hyperparameter settings, compared to the vast set of possible hyperparameter settings.

We've seen that the structure of the CNN has great influence on the performance of the unsupervised learning techniques. While performance of purely supervised training increased with the complexity of the network architecture, some unsupervised learning techniques performed worse on the LeNet-1 architecture than on the simple architecture and other performed better.

We've seen that for the simple network architecture all unsupervised learning mechanisms can increase performance, while for the LeNet-1 architecture each unsupervised learning technique merely decreases performance. Note that while the LeNet-1 architecture contains more layers, the individual layers have less features. Perhaps with more features per layer we would see unsupervised pre-training to cause CNNs to outperform purely supervisedly trained CNNs by a significant amount.

Note that while the SM techniques not all outperform PCAEs on the simple network architecture, they do for the LeNet-1 architecture. It therefore seems that Spread Maximization (SM) forms a valuable alternative to PCAEs when used for deep learning.

We can conclude that it is possible and fruitful to define unsupervised learning techniques based solely on some primary desirable attributes of the output data themselves, such as spread, rather than on a reconstruction of the input. We have seen that the proposed techniques for SM produce intuitively understandable features, which have been shown to work well, achieving lower error rates than conventional unsupervised learning techniques.

The achievements of our techniques for performing Spread Maximization justify further research into novel unsupervised learning criteria which are based solely on desirable properties of ANN in themselves. They show that an unsupervised learning paradigm radically disposing with input reconstruction can be competitive to conventional unsupervised learning techniques which *are* based on input reconstruction.

6.2 FUTURE RESEARCH

This section covers in what ways the performed research is limited and in what ways the ideas and techniques presented can be extended, purified, enhanced, etc.

6.2.1 *Limitations of Performed Experiments*

First we discuss some limitations and pitfalls concerning the experiments performed.

INSTABILITY AND NON-CONVERGENCE In the method for SM by CHA, we optimize two objectives at the same time: zero-meaning and the constrained HO. The two might interact in such a way that the goal might never be reached. For example, a weight vector might grow faster than the zero-meaning objective can keep up with, causing the weight vector to always have a length greater than one. One might therefore change the zero-meaning technique such that the above situation would never occur. Alternatively, a technique might be proposed which uses a single objective function for which the optima occur only at points where the mean is zero.

Another instability problem is caused by the stochastic nature of using mini-batches to estimate the gradient of the objective over the whole dataset. This causes the weight vectors to fluctuate around the optimum, rather than converging to it. This might be quite problematic for CHA based SM, since the weight vectors of latter features depend on former ones. Since latter features converge to weight vectors orthogonal to all former weight vectors, and all former weight vectors fluctuate, we might end up in a state where some weight vectors cannot converge to the feasible space or even stay at near the origin.

One way in which one might solve this issue is by performing CHA sequentially. One would then train each weight vector separately, while keeping all preceding weight vectors constant. Alternatively, we might use our parallel method with the addition of locking the weights sequentially as soon as they have converged.

NUMBER OF FEATURES Note that we have tested the unsupervised learning techniques on CNNs with 20 features, while in literature it is quite common to experiment with ANNs with an order of magnitude more hidden neurons. Recall from section 5.1.3 that the number of features one should like to use is related to the number of principal components, which is equal to the dimensionality of the convolution field for convoluted PCA. We should expect the performance of CHA based SM to benefit less from having more features than the convolution field dimensionality than the PCAE does, which doesn't have such a relation to PCA. When experiment-

ing with a larger number of features we might find that PCAEs outperform our methods by some amount.

DEEP LEARNING A big shortcoming of the experiments performed is the shallow architecture of the networks used. While Convolutional Auto-Encoders (CAEs) and PCAEs have widely been used to pre-train deep ANNs with over four layers, we didn't have the computational resources to perform such experiments. Deep architectures are currently the models with leading performance on Mixed National Institute of Standards and Technology (MNIST)[24]. As such, we haven't had the possibility to fairly compare our techniques and the CNNs they produce with the state-of-the-art in digit recognition.

RESTRICTED BOLTZMANN MACHINES We have weighed our method against a type of auto-encoder which we have called the PCAE. Another commonly used unsupervised learning technique if given by Restricted Boltzmann Machines (RBMs). However, in our experiments we have not compared our methods to RBMs. This was in part due to insufficient computational resources, but we also have principled reasons for not having compared to RBMs. In order to make a fair comparison, we have chosen to compare our methods to an unsupervised learning technique which also incorporates the pooling function into the objective. We haven't found any such modification of RBMs, however, and therefore felt that weighing our methods against RBMs wouldn't constitute a fair comparison.

It would be interesting to find or invent a form of RBM which does incorporate the pooling function used in the CNN for which the RBM is pre-training. Such method can then be weighed against our methods of SM, to see which one would work better.

6.2.2 *Extensions*

SM is an unsupervised learning objective in itself and not specifically for CNNs. One could therefore extend the proposed methods to other types of ANN. While dichotomization by eigenvolume expansion and uniformization by Kullback-Leibler (KL)-divergence minimization are objectives which are defined in terms of the outputs of a network, and hence applicable to any kind of ANN, the same doesn't hold for CHA based SM. One should derive a different formula for the constrained HO applied to the specific ANN.

The application of CHA based SM to simple ANNs has already be shown to be equivalent to performing PCA and using a whitening step to transform the loading vectors obtained. It would be interesting to see how well such a method would perform compared to denoising Auto-Encoders (AEs).

6.2.3 *Theoretic Reservations*

We conclude this thesis by discussing some limitations of the theory underlying spread maximization and what research can be performed to further crystallize and solidify the practices in SM.

GLOBAL OPTIMUM LOCALIZATION In section 4.1, 4.3.2 and 4.4.4 we've argued that for the particular objective function surfaces of EED and PSGU are such that the path of steepest ascent is in the direction of the global optimum closest to the origin. While it does seem feasible given the reasoning presented in those sections, we have not provided proof of any kind. Finding the proof or its invalidation is thus possible future research.

SOFT POOLING The soft pooling function we have provided is by definition an approximation to the hard arg max-pooling function. However, different approximations to the hard pooling function may be considered. Any soft approximation should have cases for which the output is less proximate to the output of the hard arg max-pooling function, since the hard arg max-function may have jump discontinuities (depending on the inner function).

In our case the soft approximation is less proximate when the absolute distance between the mapped input values is small (mapped by the inner function of the arg max function). Our soft approximation is given in formula 2.28. Since we initialize weights near the origin the input values to the pooling function are also small and so is their absolute difference. This means the soft pooling function starts out as performing average-pooling approximately, which causes the features to have no definite structure after an initial part of the learning phase. Thus can cause the features to converge to suboptimal solutions.

One would instead like the approximation to break when the *relative* distance between the mapped input values is small. This can be accomplished by scaling the mapped input values by the inverse of the length of the vector of mapped inputs. The soft arg max would then be given by:

$$\begin{aligned}\underset{z\in Z_j}{\text{soft}\arg\max}\, f(z) &= \mathbf{z}_j^\top \mathbf{m} \\ m_k &= \frac{c_k}{\sum_l c_l} \\ c_n &= \exp\left(p\frac{f(z_n)}{\|f(\mathbf{z}))\|}\right)\end{aligned} \tag{6.1}$$

where

- p is a hyperparameter controlling the hardness of the approximation;
- f is the inner function.

Further research would have to show how well such a pooling function performs and whether it outperforms the soft pooling function we have used in our experiments.

CHA'S CONVERGENCE TO THE FEASIBLE SPACE In section 4.5.5 and appendix C.19 we've derived the formula for the constrained HO, when applied to CNNs (the CHA). However, the derivation uses the assumption that the weight vectors converge to the feasible space much like the derivation of the formula for the constrained HO when applied to simple ANNs (the GHA). While we have shown how the update rule of the GHA converges to the feasible space, we haven't done the same for the CHA.

However, one might expect there to be such a proof, since the empirical experiments verify the conjecture. The proof might very well look a lot like the proof of convergence of the GHA, since the convolution layer can be seen as the application of the network to different sub-images, and pooling might be seen as selecting some of those sub-images, while disregarding others.

TRANSMUTATION BETWEEN DICHOTOMIZATION AND UNIFORMIZATION In section 4.5.1 we show how EED and PSGU can both be seen as performing decorrelation and variance optimization. While the optimal standard deviation of dichotomization is infinite, the optimal standard deviation of uniformization is estimated to be approximately 0.9 when using the hyperbolic tangent transfer function. We then showed how the network obtained by CHA can be converted in order to achieve either variant of SM.

It would be interesting to look at the performance of a network with weights initialized by EED but converted in order to achieve uniformization. Conversely, we could convert the transfer function of a network pre-trained with PSGU and see how well it performs as dichotomization method. Such experiments could be future research.

OPTIMAL PRE-SIGMOID VARIANCE FOR UNIFORMIZATION Both PSGU and CHA-based uniformization make use of a (finite) optimal pre-

sigmoid standard deviation σ^*. The value of the optimal pre-sigmoid standard deviation was chosen by analytical considerations. The only way in which the output can be perfectly uniformly distributed within the interval $[0, 1]$ is for the pre-sigmoid data to follow a distribution for which the cumulative distribution function is given by the transfer function. Given that we use the logistic sigmoid transfer function (or the hyperbolic tangent transfer function, which can be seen as a stretched version of it) the optimal pre-sigmoid distribution would be conform to the logistic distribution. However, a multivariate form of the logistic distribution is hard to construct; moreover, a formula for the derivative of the KL-divergence between two such multivariate distributions would be especially hard to deduce. We therefore quite pragmatically chose to use the multivariate normal distribution.

Of course a line of further research might be to find a good multivariate logistic distribution and try to derive a formula for the KL-divergence between the fitted pre-sigmoid distribution and the optimal one. However, there is no guarantee at all that the pre-sigmoid data in fact conforms to either a multivariate normal or a multivariate logistic distribution. Following the line of this argument, one would have to find a formula for the KL-divergence between the optimal pre-sigmoid logistic distribution and some distribution appropriate for modelling the actual pre-sigmoid output data.

However, we provide the reader with a more pragmatic approach; one can view the optimal standard deviation σ^* as an optimizable hyperparameter. Future research might even do away with the whole foundation of spread maximization as having two interpretations. Instead one may try to find the σ^* which results in the best performing ANNs, which could lead to output distributions which are neither nearly uniform nor nearly dichotomized, but somewhere in between.

Supposing the optimal value of σ^* lies somewhere between a value which would result in a uniform distribution and a dichotomized distribution, it would make sense that the non-binarized variant of EED outperforms both the binarized version as other uniformization techniques. Optimizing σ^* therefore clears the way for explaining why the non-binarized version of EED performs so well.

7

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Rodrigo Benensen. What is the class of this image? URL http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html.

[2] Y. Bengio. Learning deep architecures for AI. Technical Report 1312, Universite de Montreal, dept. IRO, 2007.

[3] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[4] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[5] Yoshua Bengio, Aaron C Courville, and James S Bergstra. Unsupervised models of images by spike-and-slab rbms. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1145–1152, 2011.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, 2013.

[7] James Bergstra, Aaron Courville, and Yoshua Bengio. The statistical inefficiency of sparse coding for images (or, one gabor to rule them all). *arXiv preprint arXiv:1109.6638*, 2011.

[8] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[9] Hervé Bourlard and Yves Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59 (4-5):291–294, 1988.

[10] Thomas H Brown, Edward W Kairiss, and Claude L Keenan. Hebbian synapses: biophysical mechanisms and algorithms. *Annual review of neuroscience*, 13(1):475–511, 1990.

[11] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[12] Norman Doidge. *The brain that changes itself: Stories of personal triumph from the frontiers of brain science*. Penguin, 2007.

[13] Colin Fyfe. *Hebbian learning and negative feedback networks*. Springer, 2007.

[14] Trevor Hastie, Robert Tibshirani, Jerome Friedman, T Hastie, J Friedman, and R Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[15] Simon Haykin and Neural Network. A comprehensive foundation. *Neural Networks*, 2(2004), 2004.

[16] Donald Olding Hebb. *The organization of behavior : a neuropsychological theory*. New York [etc.] : Wiley [etc.], 1949.

[17] John Hertz. *Introduction to the theory of neural computation*, volume 1. Basic Books, 1991.

[18] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[19] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[20] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[22] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. San Mateo, CA: Morgan Kaufmann, 1990.

[23] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361, 1995.

[24] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits. 2009. URL http://yann.lecun.com/exdb/mnist/.

[25] Yann LeCun, LD Jackel, L Bottou, A Brunot, C Cortes, JS Denker, H Drucker, I Guyon, UA Muller, E Sackinger, et al. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, volume 60, 1995.

[26] Michael Lewicki and Terrence Sejnowski. Learning overcomplete representations. *Neural computation*, 12(2):337–365, 2000.

[27] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59. Springer, 2011.

[28] E. Oja. Neural networks, principal components, and subspaces. *International Journal of Neural Systems*, 1(1):61–68, 1989.

[29] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[30] Karl Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[31] Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook. *Technical University of Denmark*, pages 7–15, 2012.

[32] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document, 1961.

[33] Terence D Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural networks*, 2(6):459–473, 1989.

[34] Patrice Y Simard, Dave Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *2013 12th International Conference on Document Analysis and Recognition*, volume 2, pages 958–958. IEEE Computer Society, 2003.

[35] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

[36] Bernard L Welch. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, pages 28–35, 1947.

[37] J.M. Wolfe, K.R. Kluender, D.M. Levi, L.M. Bartoshuk, and R.S. Herz. *Sensation & Perception*. Sinauer Associates Incorporated, 2008. ISBN 9780878939565. URL http://books.google.nl/books?id=ovYsnwEACAAJ.

Appendices

A

LIST OF ACRONYMS

AE Auto-Encoder

ANN Artificial Neural Network

CAE Convolutional Auto-Encoder

CDF cumulative density function

CHA Convolutional Hebbian Algorithm

CNN Convolutional Neural Network

DAE Denoising Auto-Encoder

EED Eigenvolume Expansion-Dichotomization

EMA Exponential Moving Average

GHA Generalized Hebbian Algorithm

GHL Generalized Hebbian Learning

HO Hebbian Objective

KL Kullback-Leibler

MLP Multilayer Perceptron

MNIST Mixed National Institute of Standards and Technology

PCA Principal Component Analysis

PCAE Pooling Convolutional Auto-Encoder

PCCA Pooled Convolutional Component Analysis

PDF probability density function

PSGU Pre-Sigmoid Gaussian-Uniformization

RBM Restricted Boltzmann Machine

RGB Red Green Blue

SM Spread Maximization

B

NOTATIONAL CONVENTIONS

B.1 GENERAL

σ	activation function	Function applied by a neuron to its local activation
y	output	Output(s) of a simple neural network
x	input	Input(s) of a simple neural network
σ^l	logistic sigmoid	The logistic sigmoid function
tanh	hyperbolic tangent	The hyperbolic tangent function
η	learning rate	Learning rate hyperparameter for an update rule
τ	time	Time parameter or iteration parameter for the τ^{th} update step in the learning process
a_k	local activation	The local activation of neuron k: the weighted sum of its inputs
λ	regularization cost	Hyperparameter governing by what amount to weigh a regularization term such as weight decay
$S(x)$	softmax activation function	Function which applied to the outputs of neurons in one layer, causes their outputs to sum to one
$\operatorname{soft\,max}(S)$	soft max	Function approximating the max function over a set S
$\operatorname*{arg\,max}_{x \in S} f(x)$	arg max	Function returning the x for which $f(x)$ is largest for all $x \in S$
$\operatorname{soft\,arg\,max}$	soft arg max	Soft approximation of the arg max

B.2 ANNs

Neuron

$i \multimap j$	synapse	Connection between from neuron i to neuron j
$w_{i \multimap j}$	weight	Weight on the connection from neuron i to neuron j
b_j	bias	Bias variable of neuron j
i	input	Presynaptic neuron: the variable representing its output
j	output	Postsynaptic neuron: the variable representing its output

Neural Network

L	layer	Set of neurons within one layer
L_{O}	output layer	The set of neurons within the output layer
$E(L_{\mathrm{O}})$	objective function	Function giving the value of the objective of the ANN which is to be maximized
δ_x	local objective	The local objective at a_x,
$S(A, a_m)$	softmax activation function	The softmax activation function: an output function
O	objective function	The objective function consisting of the objective function $E(L_{\mathrm{O}})$ and perhaps a regularization term R
$j \prec k$	neuron order	Some total order on the neurons within a layer

Convolutional Neural Networks

Z	feature map	Set of neurons with the same weight configuration
Z_i	pool	Set of neurons which constitute a pool in some feature map which are inputs to pooling neuron j
$\mathbf{z}_i$	pool	vector representation of Z_i
$\mathbf{m}_i$	pool contributions	weight vector used by the pooling function
$i^z_{x,y}$	input pixel	Input neuron at the coordinates (x, y) from the feature map z
$s(\mathbf{z})$	pooling function	Summarization function, or pooling function applied to a vector of neuron outputs from the corresponding pool Z
$*$	mathematical convolution	Convolution operation, convolution two functions resulting in a new function
$p^k_{(x,y)}$	pool value	Value at location (x, y) in the pooling map of feature k

Simple Neural Networks

$\mathbf{y}$	outputs	Vector of outputs of the last layer in a layered neural network
$y_j(\mathbf{x}, \mathbf{W})$	output function	Function computing the output of neuron j for given input $\mathbf{x}$ and weights $\mathbf{W}$
$\mathbf{x}$	inputs	Vector of inputs of the first layer of a layered neural network
$\mathbf{W}$	weights	Matrix containing all weights between two layers
$\mathbf{b}$	biases	Vector of all biases of neurons within one layer

Training

τ	time	Iteration number
η	learning rate	Hyperparameter scaling the update steps
α	momentum	Hyperparameter of the momentum learning mechanism
λ	weight decay	Amount of weight decay

B.3 MATH AND STATISTICS

$\mathrm{Var}\left[\cdot\right]$	variance
$\mathrm{Cov}\left[\cdot,\cdot\right]$	covariance
$\boldsymbol{\Sigma}$	covariance matrix
$\mathbb{E}[\cdot]$	expectation
λ_i	principal axis i
$\boldsymbol{\lambda}$	principal axes vector
Λ	set of Lagrange multipliers
λ	Lagrange multiplier
$\mathcal{N}$	Normal distribution
erf	Gauss error function
$U^N_{[a,b]}$	N-dimensional uniform distribution on the interval $[a, b]$
σ	standard deviation
μ	mean
$\boldsymbol{\mu}$	mean vector
$f_x(x)$	probability density function
$F_x(x)$	cumulative density function
$D_{\mathrm{KL}}(P\|\|Q)$	KL divergence from P to Q
x^*	optimal x
$\widetilde{x}$	redefinition or alteration of x
$\widehat{x}$	x fitted to data
$\overline{x}$	average x

B.4 LINEAR ALGEBRA

$\mathbf{I}$	identity matrix
$\mathbf{1}$	all-ones vector
$\mathbf{1}^N$	all-ones vector of length N
$\mathbf{M}^\mathsf{T}$	matrix transpose
$\mathbf{M}_{i\cdot}$	row i of $\mathbf{M}$
$\mathbf{M}_{\cdot i}$	column i of $\mathbf{M}$
λ_i	eigenvalue i
$\boldsymbol{\lambda}$	eigenvalue vector
$\boldsymbol{\Lambda}$	diagonal eigenvalue matrix
$\mathbf{W}$	matrix of eigenvectors (columns)
$\operatorname{diag}\mathbf{v}$	diagonal matrix with values of vector $\mathbf{v}$ on the diagonal
$\det\lvert\mathbf{M}\rvert$	determinant of matrix $\mathbf{M}$
$\operatorname{tr}\mathbf{M}$	trace of matrix $\mathbf{M}$
$\perp$	orthogonal

C

PROOFS

C.1 REDUNDANCY OF LINEAR TRANSFER FUNCTION

Suppose that $\forall i : i \multimap j : \sigma_i(x) = x$, then

$$
\begin{aligned}
j =& \sigma_j \left(\sum_{i:i \multimap j} \{i \cdot w_{i \multimap j}\} + b_j \right) \\
=& \sigma_j \left(\sum_{i:i \multimap j} \left\{ \left(\sum_{h:h \multimap i} \{h \cdot w_{h \multimap i}\} + b_i \right) \cdot w_{i \multimap j} \right\} + b_j \right) \\
=& \sigma_j \left(\sum_{i:i \multimap j} \left\{ \sum_{h:h \multimap i} \{h \cdot w_{h \multimap i} \cdot w_{i \multimap j}\} + b_i \cdot w_{i \multimap j} \right\} + b_j \right) \\
=& \sigma_j \left(\sum_{i:i \multimap j} \left\{ \sum_{h:h \multimap i} \{h \cdot w_{h \multimap i} \cdot w_{i \multimap j}\} \right\} \right. \\
& \left. + \sum_{i:i \multimap j} \{b_i \cdot w_{i \multimap j}\} + b_j \right) \\
=& \sigma_j \left(\sum_{h:\exists i:h \multimap i \wedge i \multimap j} \left\{ \sum_{i:h \multimap i \wedge i \multimap j} \{h \cdot w_{h \multimap i} \cdot w_{i \multimap j}\} \right\} \right. \\
& \left. + \sum_{i:i \multimap j} \{b_i \cdot w_{i \multimap j}\} + b_j \right) \\
=& \sigma_j \left(\sum_{h:h \multimap^{\mathrm{D}} j} \left\{ h \cdot w_{h \multimap^{\mathrm{D}} j} \right\} + b_j^{\mathrm{D}} \right) \qquad \text{(C.1)}
\end{aligned}
$$

where

- $h \multimap^{\mathrm{D}} j \iff \exists i : h \multimap i \wedge i \multimap j$;
- $w_{h \multimap^{\mathrm{D}} j} = \sum_{i:h \multimap i \wedge i \multimap j} w_{h \multimap i} \cdot w_{i \multimap j}$;
- $b_j^{\mathrm{D}} = \sum_{i:i \multimap j} \{b_i \cdot w_{i \multimap j}\} + b_j$.

So instead of the intermediate neurons i, we can have an equivalent network with direct connections $\multimap^{\mathrm{D}}$.

C.2 EQUIVALENCE OF NEURAL NETWORK WITH TANH AND WITH LOGISTIC SIGMOID TRANSFER FUNCTION

$$
\begin{aligned}
a_k &= \tanh\left(\sum_{i:i\multimap j}\{i\cdot w_{i\multimap j}\}+b_j\right)\cdot w_{j\multimap k}+b_k \\
&= \left(2\sigma^l\left(2\left(\sum_{i:i\multimap j}\{i\cdot w_{i\multimap j}\}+b_j\right)\right)-1\right)\cdot w_{j\multimap k}+b_k \\
&= \left(2\sigma^l\left(\sum_{i:i\multimap j}\{i\cdot 2w_{i\multimap j}\}+2b_j\right)-1\right)\cdot w_{j\multimap k}+b_k \\
&= 2\sigma^l\left(\sum_{i:i\multimap j}\{i\cdot 2w_{i\multimap j}\}+2b_j\right)\cdot w_{j\multimap k}-w_{j\multimap k}+b_k \\
&= \sigma^l\left(\sum_{i:i\multimap j}\{i\cdot 2w_{i\multimap j}\}+2b_j\right)\cdot 2w_{j\multimap k}-w_{j\multimap k}+b_k \\
&= \sigma^l\left(\sum_{i:i\multimap j}\left\{i\cdot w^{\mathrm{S}}_{i\multimap j}\right\}+b^{\mathrm{S}}_j\right)\cdot w^{\mathrm{S}}_{j\multimap k}+b^{\mathrm{S}}_k \qquad \text{(C.2)}
\end{aligned}
$$

where

- $w^{\mathrm{S}}_{i\multimap j}=2w_{i\multimap j}$;
- $b^{\mathrm{S}}_j=2b_j$;
- $b^{\mathrm{S}}_k=b_k-w_{j\multimap k}$;
- $w^{\mathrm{S}}_{j\multimap k}=2w_{j\multimap k}$.

C.3 DERIVATIVE OF THE SOFTMAX ACTIVATION FUNCTION

For simplicity of the equations to come, we denote the output of the softmax activation function as $o_k = S(\{a_n | n \in L_O\}, a_k)$

$$o_k = \frac{e^{a_k}}{\sum_l e^{a_l}} \tag{C.3}$$

$$\begin{aligned}
\frac{\partial o_k}{\partial a_n} &= \frac{\partial}{\partial a_n} \frac{e^{a_k}}{\sum_l e^{a_l}} \\
&= \frac{\partial e^{a}k}{\partial a_n} \frac{1}{\sum_l e^{a_l}} + e^{a_k} \frac{\partial}{\partial a_n} \frac{1}{\sum_l e^{a_l}} \\
&= \frac{\partial a_k}{\partial a_n} e^{a_k} \frac{1}{\sum_l e^{a_l}} + e^{a_k} \frac{\partial}{\partial a_n} \frac{1}{\sum_l e^{a_l}}
\end{aligned} \tag{C.4}$$

$$\begin{aligned}
\frac{\partial}{\partial a_n} \frac{1}{\sum_l e^{a_l}} &= -\frac{\frac{\partial}{\partial a_n} \sum_l e^{a_l}}{(\sum_l e^{a_l})^2} = -\frac{\sum_l \frac{\partial}{\partial a_n} e^{a_l}}{(\sum_l e^{a_l})^2} \\
&= -\frac{\sum_l \frac{\partial}{\partial a_n} e^{a_l}}{(\sum_l e^{a_l})^2} = -\frac{\frac{\partial}{\partial a_n} e^{a_n}}{(\sum_l e^{a_l})^2} = -\frac{e^{a_n}}{(\sum_l e^{a_l})^2}
\end{aligned} \tag{C.5}$$

$$\begin{aligned}
\frac{\partial o_k}{\partial a_n} &= \frac{\partial a_k}{\partial a_n} e^{a_k} \frac{1}{\sum_l e^{a_l}} - e^{a_k} \frac{e^{a_n}}{(\sum_l e^{a_l})^2} \\
&= \frac{\partial a_k}{\partial a_n} \frac{e^{a_k}}{\sum_l e^{a_l}} - e^{a_k} \frac{e^{a_n}}{(\sum_l e^{a_l})^2} \\
&= \frac{\partial a_k}{\partial a_n} \frac{e^{a_k}}{\sum_l e^{a_l}} - \frac{e^{a_k}}{\sum_l e^{a_l}} \frac{e^{a_n}}{\sum_l e^{a_l}} \\
&= \frac{\partial a_k}{\partial a_n} o_k - o_k o_n \\
&= o_k \left(\frac{\partial a_k}{\partial a_n} - o_n \right) \\
&= o_k \left((n \equiv k) ? * - o_n \right) \\
&= o_k \left((n \equiv k) ? - o_n \right)
\end{aligned} \tag{C.6}$$

For any function e which ranges over all outputs of the softmax activation function, we have that

$$\begin{aligned}
\frac{\partial e}{\partial a_n} &= \sum_k \frac{\partial e_k}{\partial o_k}\frac{\partial o_k}{\partial a_n} \\
&= \sum_k \left\{ \frac{\partial e_k}{\partial o_k} o_k \left((n \equiv k)? - o_n \right) \right\} \\
&= \sum_k \left\{ \frac{\partial e_k}{\partial o_k} o_k (n \equiv k)? - \frac{\partial e_k}{\partial o_k} o_k o_n \right\} \\
&= \frac{\partial e_n}{\partial o_n} o_n - \sum_k \left\{ \frac{\partial e_k}{\partial o_k} o_k o_n \right\} \\
&= o_n \left(\frac{\partial e_n}{\partial o_n} - \sum_k \frac{\partial e_k}{\partial o_k} o_k \right)
\end{aligned} \tag{C.7}$$

C.4 DERIVATIVE CROSS ENTROPY

Given a dataset of N data points and K output neurons, corresponding to K classes, cross entropy is defined by:

$$H(\mathbf{o}, \mathbf{t}) = -\sum_n^N \sum_k^K t_{kn} \log o_{kn} \tag{C.8}$$

where we apply 1-of-K coding to represent the true distribution t over the classes K, so for a given sample n having true class c (such that $t_{xn} = 1$ if $x = c$ and $t_{xn} = 0$ otherwise) we have

$$\begin{aligned}
H(\mathbf{o}, \mathbf{t}) &= -\sum_k^K t_{kn} \log o_{kn} \\
&= -\sum_k^K (k \equiv c)? \log o_{kn} \\
&= -\log o_{cn}
\end{aligned} \tag{C.9}$$

for which the derivative is given by (omitting the subscript n)

$$\begin{aligned}
\frac{\partial H(\mathbf{o},\mathbf{t})}{\partial o_k} &= \frac{\partial}{\partial o_k} - \log o_c \\
&= (k \equiv c)? \cdot -\frac{t_k}{o_k} \\
&= -(k \equiv c)? \frac{t_c}{o_c} \\
&= -(k \equiv c)? \frac{1}{o_c}
\end{aligned} \tag{C.10}$$

$$
\begin{aligned}
\frac{\partial H(\mathbf{o},\mathbf{t})}{\partial a_n} &= \sum_k^K \frac{\partial H(\mathbf{o},\mathbf{t})}{\partial o_k} \cdot \frac{\partial o_k}{\partial a_n} \\
&= \sum_k^K o_k \left((n \equiv k)? - o_n\right) \cdot -(k \equiv c)? \frac{1}{o_c} \\
&= o_c \left((n \equiv c)? - o_n\right) \cdot -\frac{1}{o_c} \\
&= -\left((n \equiv c)? - o_n\right) \\
&= (o_n - (n \equiv c)?)
\end{aligned} \tag{C.11}
$$

C.5 soft arg max DERIVATIVE

$$
\begin{aligned}
o &= \sum_k \left\{ o_k \frac{e^{f(o_k)}}{\sum\limits_l e^{f(o_l)}} \right\} \\
&= \frac{1}{\sum\limits_k e^{f(o_k)}} \sum_k o_k e^{f(o_k)}
\end{aligned} \tag{C.12}
$$

$$
\begin{aligned}
\frac{\partial o}{\partial o_n} =& \frac{\partial}{\partial o_n} \left\{ \frac{1}{\sum\limits_k e^{f(o_k)}} \sum_k o_k e^{f(o_k)} \right\} \\
=& \frac{\partial}{\partial o_n} \left\{ \frac{1}{\sum\limits_k e^{f(o_k)}} \right\} \sum_k o_k e^{f(o_k)} + \frac{1}{\sum\limits_k e^{f(o_k)}} \frac{\partial}{\partial o_n} \left\{ \sum_k o_k e^{f(o_k)} \right\}
\end{aligned} \tag{C.13}
$$

$$
\begin{aligned}
\frac{\partial}{\partial o_n} \sum_k o_k e^{f(o_k)} &= \frac{\partial}{\partial o_n} o_n e^{f(o_n)} \\
= e^{f(o_n)} + o_n f'(o_n) e^{f(o_n)} &= \left(o_n f'(o_n) + 1 \right) e^{f(o_n)}
\end{aligned} \tag{C.14}
$$

$$
\frac{\partial}{\partial o_n} \frac{1}{\sum\limits_l e^{f(o_l)}} = -\frac{\frac{\partial}{\partial o_n} \sum\limits_l e^{f(o_l)}}{\left(\sum\limits_l e^{f(o_l)} \right)^2} = -\frac{\frac{\partial}{\partial o_n} e^{f(o_n)}}{\left(\sum\limits_l e^{f(o_l)} \right)^2} = -\frac{f'(o_n) e^{f(o_n)}}{\left(\sum\limits_l e^{f(o_l)} \right)^2} \tag{C.15}
$$

$$
\begin{aligned}
\frac{\partial o}{\partial o_n} =& \frac{\partial}{\partial o_n}\left\{\frac{1}{\sum_k e^{f(o_k)}}\right\}\sum_k o_k e^{f(o_k)} + \frac{1}{\sum_k e^{f(o_k)}}\frac{\partial}{\partial o_n}\left\{\sum_k o_k e^{f(o_k)}\right\} \\
=& -\frac{f'(o_n)e^{f(o_n)}}{(\sum_l e^{f(o_l)})^2}\sum_k o_k e^{f(o_k)} + \frac{1}{\sum_k e^{f(o_k)}}\left(o_n f'(o_n)+1\right)e^{f(o_n)} \\
=& -\frac{f'(o_n)e^{f(o_n)}}{\sum_l e^{f(o_l)}}o + \frac{1}{\sum_k e^{f(o_k)}}\left(o_n f'(o_n)+1\right)e^{f(o_n)} \\
=& \frac{e^{f(o_n)}}{\sum_k e^{f(o_k)}}\left(o_n f'(o_n) + 1 - f'(o_n)o\right) \\
=& \frac{e^{f(o_n)}}{\sum_k e^{f(o_k)}}\left(1 + f'(o_n)\left(o_n - o\right)\right) \\
=& \, c_n\left(1 + f'(o_n)(o_n - o)\right) \qquad \text{(C.16)}
\end{aligned}
$$

C.6 soft arg max CONTRIBUTION POOL DERIVATIVE

Each input o_n is accompanied by an output, which we denote o'_n:

$$o'_n = o_n c_n = o_n \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} \tag{C.17}$$

First note that

$$\begin{aligned}\frac{\partial}{\partial o_n}\frac{1}{\sum_l e^{f(o_l)}} &= -\frac{\frac{\partial}{\partial o_n}\sum_l e^{f(o_l)}}{(\sum_l e^{f(o_l)})^2} = -\frac{\sum_l \frac{\partial}{\partial o_n} e^{f(o_l)}}{(\sum_l e^{f(o_l)})^2} \\ &= -\frac{\frac{\partial}{\partial o_n} e^{f(o_n)}}{(\sum_l e^{f(o_l)})^2} = -\frac{f'(o_n)e^{f(o_n)}}{(\sum_l e^{f(o_l)})^2}\end{aligned} \tag{C.18}$$

When $n \neq k$:

$$\begin{aligned}\frac{\partial o'_k}{\partial o_n} &= \frac{\partial}{\partial o_n} o_k \frac{e^{f(o_k)}}{\sum_l e^{f(o_l)}} \\ &= o_k \frac{\partial}{\partial o_n}\frac{e^{f(o_k)}}{\sum_l e^{f(o_l)}} \\ &= o_k e^{f(o_k)} \frac{\partial}{\partial o_n}\frac{1}{\sum_l e^{f(o_l)}} \\ &= -o_k e^{f(o_k)} \frac{f'(o_n)e^{f(o_n)}}{(\sum_l e^{f(o_l)})^2} \\ &= -f'(o_n) o_k \frac{e^{f(o_k)}}{\sum_l e^{f(o_l)}} \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} \\ &= -f'(o_n) o'_k c_n\end{aligned} \tag{C.19}$$

When $k = n$

$$\begin{aligned}
\frac{\partial o'_n}{\partial o_n} &= \frac{\partial}{\partial o_n} o_n \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} \\
&= \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} + o_n \frac{\partial}{\partial o_n} \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} \\
&= \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} + o_n \left(f'(o_n) e^{f(o_n)} \frac{1}{\sum_l e^{f(o_l)}} + e^{f(o_n)} \frac{\partial}{\partial o_n} \frac{1}{\sum_l e^{f(o_l)}} \right) \\
&= \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} + o_n \left(f'(o_n) \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} - e^{f(o_n)} \frac{f'(o_n) e^{f(o_n)}}{(\sum_l e^{f(o_l)})^2} \right) \\
&= \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} + f'(o_n) o_n \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} - f'(o_n) o_n \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} \frac{e^{f(o_n)}}{\sum_l e^{f(o_l)}} \\
&= c_n + f'(o_n) o'_n - f'(o_n) o'_n c_n
\end{aligned} \tag{C.20}$$

For any E, function of the whole pool, we have:

$$\begin{aligned}
\frac{\partial E}{\partial o_n} &= \sum_l \frac{\partial E}{\partial o'_l} \frac{\partial o'_l}{\partial o_l} \\
&= \frac{\partial E}{\partial o'_n} \frac{\partial o'_n}{\partial o_n} + \sum_{l \neq n} \frac{\partial E}{\partial o'_l} \frac{\partial o'_l}{\partial o_n} \\
&= \frac{\partial E}{\partial o'_n} \left(c_n + f'(o_n) o'_n - f'(o_n) o'_n c_n \right) + \sum_{l \neq n} \frac{\partial E}{\partial o'_l} \left(-f'(o_n) o'_l c_n \right) \\
&= \frac{\partial E}{\partial o'_n} \left(c_n + f'(o_n) o'_n \right) - \frac{\partial E}{\partial o'_n} f'(o_n) o'_n c_n - \sum_{l \neq n} \frac{\partial E}{\partial o'_l} f'(o_n) o'_l c_n \\
&= \frac{\partial E}{\partial o'_n} \left(c_n + f'(o_n) o'_n \right) - \sum_l \frac{\partial E}{\partial o'_l} f'(o_n) o'_l c_n \\
&= \frac{\partial E}{\partial o'_n} \left(c_n + f'(o_n) o'_n \right) - f'(o_n) c_n \sum_l \frac{\partial E}{\partial o'_l} o'_l
\end{aligned} \tag{C.21}$$

C.7 VARIANCE OF PRINCIPAL COMPONENTS

$$
\begin{aligned}
&\operatorname{Var}\left[\mathbf{Z}_{\cdot k}\right] \\
&= \mathbb{E}\left[\mathbf{Z}_{\cdot k}^2\right] - \mathbb{E}\left[\mathbf{Z}_{\cdot k}\right]^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\mathbf{X}_{n\cdot}\mathbf{W}_{\cdot k}\right)^2 - \left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{X}_{n\cdot}\mathbf{W}_{\cdot k}\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbf{X}_{ni}\right)^2 - \left(\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbf{X}_{ni}\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbf{X}_{ni}\right)\left(\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbf{X}_{ni}\right) - \left(\sum_{i=1}^{K}\mathbf{W}_{ik}\frac{1}{N}\sum_{n=1}^{N}\mathbf{X}_{ni}\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K}\sum_{j=1}^{K}\mathbf{W}_{ik}\mathbf{X}_{ni}\mathbf{W}_{jk}\mathbf{X}_{nj} - \left(\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbb{E}\left[\mathbf{X}_{\cdot i}\right]\right)^2 \\
&= \sum_{i=1}^{K}\sum_{j=1}^{K}\mathbf{W}_{ik}\mathbf{W}_{jk}\frac{1}{N}\sum_{n=1}^{N}\mathbf{X}_{ni}\mathbf{X}_{nj} - \left(\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbb{E}\left[\mathbf{X}_{\cdot i}\right]\right)\left(\sum_{i=1}^{K}\mathbf{W}_{ik}\mathbb{E}\left[\mathbf{X}_{\cdot i}\right]\right) \\
&= \sum_{i=1}^{K}\sum_{j=1}^{K}\mathbf{W}_{ik}\mathbf{W}_{jk}\mathbb{E}\left[\mathbf{X}_{\cdot i}\mathbf{X}_{\cdot j}\right] - \sum_{i=1}^{K}\sum_{j=1}^{K}\mathbf{W}_{ik}\mathbb{E}\left[\mathbf{X}_{\cdot i}\right]\mathbf{W}_{jk}\mathbb{E}\left[\mathbf{X}_{\cdot j}\right] \\
&= \sum_{i=1}^{K}\sum_{j=1}^{K}\mathbf{W}_{ik}\mathbf{W}_{jk}\left(\mathbb{E}\left[\mathbf{X}_{\cdot i}\mathbf{X}_{\cdot j}\right] - \mathbb{E}\left[\mathbf{X}_{\cdot i}\right]\mathbb{E}\left[\mathbf{X}_{\cdot j}\right]\right) \\
&= \sum_{i,j=1}^{K}\mathbf{W}_{ik}\mathbf{W}_{jk}\sigma_{\mathbf{X}_{\cdot i}\mathbf{X}_{\cdot j}} \\
&= \mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} \qquad \text{(C.22)}
\end{aligned}
$$

where

- K is the dimensionality of the data;
- N is the size of the data set;

Thanks to Jolliffe[20].

C.8 COVARIANCE OF PRINCIPAL COMPONENTS

$$\begin{aligned}\mathbb{E}\left[\mathbf{Z}_{\cdot k}\right] &= \mathbb{E}\left[\mathbf{X}\mathbf{W}_{\cdot k}\right] \\ &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{X}_{n\cdot}\mathbf{W}_{\cdot k} \\ &= \mathbf{W}_{\cdot k}\frac{1}{N}\sum_{n=1}^{N}\mathbf{X}_{n\cdot} \\ &= \mathbf{W}_{\cdot k}\mathbb{E}\left[\mathbf{X}_{n\cdot}\right] \end{aligned} \tag{C.23}$$

Covariance is invariant under translations of the data:

$$\operatorname{Cov}\left[\mathbf{X}_{\cdot l}, \mathbf{X}_{\cdot k}\right] = \operatorname{Cov}\left[\mathbf{Y}_{\cdot l}, \mathbf{Y}_{\cdot k}\right] \tag{C.24}$$

where

- $\mathbf{Y} = \mathbf{X} + \mathbf{1}^N\mathbf{v}^\mathsf{T}$;
- $\mathbf{1}^N$ is an N-dimensional column vector of only ones;
- $\mathbf{v}$ is any column vector with the same dimension as the width of $\mathbf{X}$.

Let's subtract the mean from the data:

$$\mathbf{X} \mapsto \mathbf{X} + \mathbf{1}^N\boldsymbol{\mu}^\mathsf{T} \tag{C.25}$$

We then have:

$$\begin{aligned}\operatorname{Cov}\left[\mathbf{X}_{\cdot l}, \mathbf{X}_{\cdot k}\right] &= \mathbb{E}\left[\mathbf{X}_{\cdot l}\mathbf{X}_{\cdot k}\right] - \mathbb{E}\left[\mathbf{X}_{\cdot l}\right]\mathbb{E}\left[\mathbf{X}_{\cdot k}\right] \\ &= \mathbb{E}\left[\mathbf{X}_{\cdot l}\mathbf{X}_{\cdot k}\right] \end{aligned} \tag{C.26}$$

$$\boldsymbol{\Sigma} = \frac{1}{N}\mathbf{X}^\mathsf{T}\mathbf{X} \tag{C.27}$$

$$\begin{aligned}\mathbb{E}\left[\mathbf{Z}_{\cdot l}\mathbf{Z}_{\cdot k}\right] &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{W}_{\cdot l}\mathbf{X}_{n\cdot}\mathbf{W}_{\cdot k}\mathbf{X}_{n\cdot} \\ &= \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K}\mathbf{W}_{il}\mathbf{X}_{ni}\mathbf{W}_{ik}\mathbf{X}_{ni} \\ &= \sum_{i=1}^{K}\mathbf{W}_{il}\mathbf{W}_{ik}\frac{1}{N}\sum_{n=1}^{N}\mathbf{X}_{ni}\mathbf{X}_{ni} \\ &= \sum_{i=1}^{K}\mathbf{W}_{il}\mathbf{W}_{ik}\frac{1}{N}\mathbf{X}_{\cdot i}^\mathsf{T}\mathbf{X}_{\cdot i} \\ &= \mathbf{W}_{\cdot l}\frac{1}{N}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{W}_{\cdot k} \\ &= \mathbf{W}_{\cdot l}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} \\ &= \mathbf{W}_{\cdot l}\lambda_k\mathbf{W}_{\cdot k} \\ &= \lambda_k\mathbf{W}_{\cdot l}\mathbf{W}_{\cdot k} = 0 \end{aligned} \tag{C.28}$$

C.9 FIRST PRINCIPAL COMPONENT

$$\mathbf{W}_{\cdot 1}^{\mathsf{T}} \boldsymbol{\Sigma} \mathbf{W}_{\cdot 1} + \lambda_1 \left(1 - \mathbf{W}_{\cdot 1}^{\mathsf{T}} \mathbf{W}_{\cdot 1}\right) \tag{C.29}$$

is optimal when its partial derivatives are zero.

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{W}_{\cdot 1}} \left\{ \mathbf{W}_{\cdot 1}^{\mathsf{T}} \boldsymbol{\Sigma} \mathbf{W}_{\cdot 1} + \lambda_1 \left(1 - \mathbf{W}_{\cdot 1}^{\mathsf{T}} \mathbf{W}_{\cdot 1}\right) \right\} &= 0 \\
\frac{\partial}{\partial \mathbf{W}_{\cdot 1}} \left\{ \mathbf{W}_{\cdot 1}^{\mathsf{T}} \boldsymbol{\Sigma} \mathbf{W}_{\cdot 1} \right\} + \frac{\partial}{\partial \mathbf{W}_{\cdot 1}} \left\{ \lambda_1 \left(1 - \mathbf{W}_{\cdot 1}^{\mathsf{T}} \mathbf{W}_{\cdot 1}\right) \right\} &= 0 \\
\left(\boldsymbol{\Sigma} + \boldsymbol{\Sigma}^{\mathsf{T}}\right) \mathbf{W}_{\cdot 1} - \lambda_1 \frac{\partial}{\partial \mathbf{W}_{\cdot 1}} \left\{ \mathbf{W}_{\cdot 1}^{\mathsf{T}} \mathbf{W}_{\cdot 1} \right\} &= 0 \\
2\boldsymbol{\Sigma} \mathbf{W}_{\cdot 1} - \lambda_1 2 \mathbf{W}_{\cdot 1} &= 0 \\
\boldsymbol{\Sigma} \mathbf{W}_{\cdot 1} &= \lambda_1 \mathbf{W}_{\cdot 1} \qquad \text{(C.30)}
\end{aligned}$$

C.10 FURTHER PRINCIPAL COMPONENTS

$$\mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} + \lambda_k\left(1 - \mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot k}\right) + \sum_{j=1}^{K}\lambda_{kj}\left(\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot j}\right) \tag{C.31}$$

is optimal when its partial derivatives are zero.

We will first consider the partial derivatives of the formula w.r.t. λ_{kl} for each l:

$$\begin{aligned}
\frac{\partial}{\partial\lambda_{kl}}\Big\{\mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} + \lambda_k\left(1 - \mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot k}\right) \\
+ \sum_{j=1}^{K}\lambda_{kl}\left(\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot j}\right)\Big\} &= 0 \\
\frac{\partial}{\partial\lambda_{kj}}\left\{\sum_{j=1}^{K}\lambda_{kj}\left(\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot j}\right)\right\} &= 0 \\
\frac{\partial}{\partial\lambda_{kj}}\left\{\lambda_{kl}\left(\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot l}\right)\right\} &= 0 \\
\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot l} &= 0
\end{aligned} \tag{C.32}$$

When we substitute 0 for $\mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot l}$ for each l, the formula reduces to:

$$\mathbf{W}_{\cdot k}^{\mathsf{T}}\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} + \lambda_k\left(1 - \mathbf{W}_{\cdot k}^{\mathsf{T}}\mathbf{W}_{\cdot k}\right) \tag{C.33}$$

The above formula and the derivation of the variance of the projected data are analogous to appendix C.9. We can conclude that

$$\boldsymbol{\Sigma}\mathbf{W}_{\cdot k} = \lambda_k\mathbf{W}_{\cdot k} \tag{C.34}$$

C.11 GRAM-SCHMIDT PROCESS

The Gram-Schmidt process is a method to perform orthonormalization. It can be seen to consist of two stages: an orthogonalization stage and a normalization stage. However, we will show below that merging the two stages leads to less complex formulas. We will first explain the orthogonalization stage and then move on to explain how the inclusion of normalization simplifies the process.

C.11.1 *Gram-Schmidt Orthogonalization*

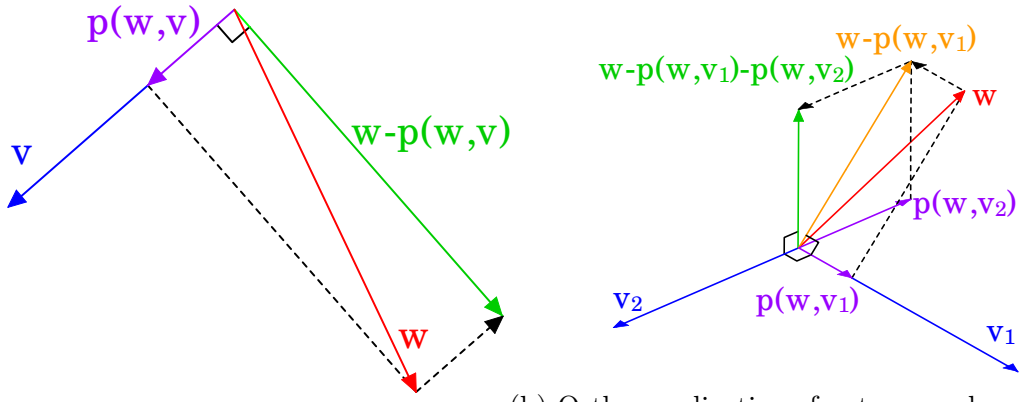
Gram-Schmidt orthogonalization is a simple procedure to make a given list of vectors orthogonal to each other. The first vector remains the same as the original, but every subsequent vector is made to be orthogonal to all preceding orthogonalized vectors. Orthogonalization of a vector $\mathbf{w}_m$ to a given vector $\mathbf{v}_n$ is performed by subtracting the component of $\mathbf{w}_m$ in the direction of $\mathbf{v}_n$; the orthogonalized vector $\mathbf{o}_m$ is obtained by subtraction of the projection of $\mathbf{w}_m$ onto $\mathbf{v}_n$:

$$\mathbf{o}_m = \mathbf{w}_m - p(\mathbf{w}_m, \mathbf{v}_n) \tag{C.35}$$

$$p(\mathbf{w}_m, \mathbf{v}_n) = \frac{\mathbf{v}_n^\mathsf{T}\mathbf{w}_m}{\mathbf{v}_n^\mathsf{T}\mathbf{v}_n}\mathbf{v}_n \tag{C.36}$$

where

- $p(\mathbf{w}, \mathbf{v})$ is the projection of vector $\mathbf{w}$ onto vector $\mathbf{v}$;
- $p(\mathbf{w}, \mathbf{0}) = 0$.



(a) Orthogonalization of $\mathbf{w}$ to $\mathbf{v}$.

(b) Orthogonalization of $\mathbf{w}$ to $\mathbf{v}_1$ and $\mathbf{v}_2$.

Figure C.1: Visual representation of Gram-Schmidt Orthogonalization.

Gram-Schmidt orthogonalization then consists of sequentially orthogonalizing each vector to each preceding vector:

$$\mathbf{w}_k^{\perp} = \mathbf{w}_k - \sum_{j=1}^{k-1} p(\mathbf{w}_k, \mathbf{w}_j^{\perp}) \tag{C.37}$$

Note that we first compute $\mathbf{o}_j$ before we use its value in any further $\mathbf{o}_k$ for $k > j$. See figure C.1 for visual aid to this explanation.

C.11.2 *Orthonormalization*

As we have said in section 3.1, we want the principal axes to be of unit length. We will therefore perform normalization after orthogonalization, constituting orthonormalization. When we perform Gram-Schmidt orthogonalization with normalization, we perform what is called the *Gram-Schmidt process*.

We could do the normalization step after orthogonalizing all vectors ($\mathbf{w}_k \mapsto \mathbf{o}_k / \|\mathbf{w}_k\|$), but we might just as well normalize during Gram-Schmidt orthogonalization. Instead of first iteratively orthogonalizing and then normalizing, we could iteratively do orthogonalization and normalization steps. The two are equivalent because the projection of a vector onto vector $\mathbf{v}$ results in the same vector as projection onto the normalized $\mathbf{v}$:

$$p(l\mathbf{w}, \mathbf{v}) = \frac{(l\mathbf{v})^{\mathsf{T}}\mathbf{w}}{(l\mathbf{v})^{\mathsf{T}}(l\mathbf{v})}(l\mathbf{v}) = l\frac{l\mathbf{v}^{\mathsf{T}}\mathbf{w}}{l^2\mathbf{v}^{\mathsf{T}}\mathbf{v}}\mathbf{v} = \frac{\mathbf{v}^{\mathsf{T}}\mathbf{w}}{\mathbf{v}^{\mathsf{T}}\mathbf{v}}\mathbf{v} = p(\mathbf{w}, \mathbf{v}) \tag{C.38}$$

where

- l could be any scalar value.

The resulting process can then be simplified, because we know that the previous vectors already have unit length:

$$\mathbf{w}_k \mapsto \frac{\mathbf{w}_k^{\perp}}{\left\|\mathbf{w}_k^{\perp}\right\|} \tag{C.39}$$

$$\begin{aligned}\mathbf{w}_k^{\perp} &= \mathbf{w}_k - \sum_{j=1}^{k-1} \frac{\mathbf{w}_j^{\mathsf{T}}\mathbf{w}_k}{\mathbf{w}_j^{\mathsf{T}}\mathbf{w}_j}\mathbf{w}_j \\ &= \mathbf{w}_k - \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}}\mathbf{w}_k\right)\mathbf{w}_j \end{aligned} \tag{C.40}$$

where

- $\mathbf{w}_k^{\perp}$ is the orthogonalized vector $\mathbf{w}_k$.

C.12 PROOF THAT THE GHA PERFORMS PCA

We will here show how the update rule of the GHA will let the weight vectors converge to principal components. For simplicity we suppose our data has zero mean, so that the covariance of x_1 and x_2 is proportional to $\mathbb{E}[x_1 x_2]$ for any two different input variables x_1 and x_2. When the data at hand does not have zero mean, we simply apply a preprocessing stage in which we subtract the mean, so that we guarantee the data presented to be centered around the origin.

The proof is performed by mathematical induction on the output neurons. The base case consists of the proof that the weight vector of the first neuron will converge to the first principal component. Provided that all previous weight vectors have converged to principal components, the inductive step then consists of the proof that the consecutive weight vector will converge to the next principal component.

C.12.1 *Oja's Rule*

Remember that the update rule of the GHA is given by:

$$\Delta w_{i \multimap k} = \eta y_k \left(x_i - \sum_{j=1}^{k} w_{i \multimap j} y_j \right) \tag{C.41}$$

For the first neuron the GHA reduces to what is known as *Oja's Rule*:

$$\Delta w_i = \eta y \left(x_i - y w_i \right) \tag{C.42}$$

Here we have left out the index k because Oja's rule is only concerned with a single output neuron. In this section we leave k implicit, since all weights are weights on connections to the same neuron y.

Oja's Rule is derived from Hebb's rule constrained within a weight space where the weight vectors are of unit length. Because Hebb's rule can be viewed as an optimization technique for the HO, which is proportional to variance of the output when the output data has zero mean, Oja's rule causes the weight vector to converge to the first principal component; the first principal component is defined by the weight vector which maximizes the variance of the converted data, under the restriction that the weight vector has unit length.

Note that when the input has zero mean, the output of an ANN with linear activation function also has zero mean, since the network forms a linear combination of the input values. Given that the input does have zero mean, we can conclude that the HO is indeed proportional to the output variance.

The constrained update rule consists of two steps; first it updates the weights according to Hebb's rule and then it normalizes the weight vector:

$$
\begin{aligned}
w_i &\mapsto w_i + \eta y x_i \\
w_i &\mapsto \frac{w_i}{\sqrt{\sum_{j=1}^{K} w_j^2}}
\end{aligned}
\tag{C.43}
$$

where

- K is the number of input neurons.

We can merge these into the single stage and rewrite it by doing a first order Taylor expansion about the point $\eta = 0$, constituting a Maclaurin series:

$$
\begin{aligned}
w_i^{\tau+1} &= \frac{w_i + \eta y x_i}{\sqrt{\sum_{j=1}^{K} (w_i + \eta y x_i)^2}} \\
&= \frac{1}{l} w_i + \eta y \frac{1}{l} \left(x_i - w_i \frac{1}{l^2} \sum_j w_j x_j \right) + O(\eta^2)
\end{aligned}
\tag{C.44}
$$

where

- l is the length of the weight vector at iteration τ, i.e. $l = \sqrt{\left(\sum_j w_j^p \right)}$
- τ indicates the time or iteration; it's omitted where possible.
- $O(\eta^2)$ are terms with at most an order of magnitude proportional to η^2

Generally we choose η to be quite small, $\eta \ll 1$, so that $\eta^2 \lll \eta$ and so the effect of $O(\eta^2)$ on the value of $w_i^{\tau+1}$ is negligible. We can therefore safely omit all higher order terms from the formula to approximate the weight update.

When we further suppose that the weight vector from the previous step was already of unit length, i.e. $l = 1$ and that the linear transfer function is used (as is usual when using Hebbian learning rules), we end up with Oja's rule:

$$
\begin{aligned}
w_i^{\tau+1} &= \frac{1}{l} w_i + \eta y \frac{1}{l} \left(x_i - w_i \frac{1}{l^2} \sum_j w_j x_j \right) \\
&= w_i + \eta y \left(x_i - w_i \sum_j w_j x_j \right) \\
&= w_i + \eta y \left(x_i - w_i y \right)
\end{aligned}
\tag{C.45}
$$

Approximation to the Feasible Space

We have now proved that the unit length constraint is preserved under application of Oja's rule; when the weight vector at some point has unit length the update rule will let the weight vector converge to the first principal component. We still have to prove that Oja's rule will in fact let the weight vector converge to any weight vector of unit length.

The hypersurface within the weight space where the weight vector is of unit length is called the *the feasible space*, since in that subspace the unit length constraint is met. Under the current constraint the the feasible space is a hypersphere centered at the origin. In this section we prove that Oja's rule causes the weights to converge to the the feasible space.
After a weight update the length of a weight vector is given by:

$$\sqrt{\sum_{i=1}^{K}\left(w_i^{\tau+1}\right)^2} \tag{C.46}$$

For simplicity of the formulas, let's consider the squared length of the weight vector:

$$\begin{aligned}
\sum_{i=1}^{K}\left(w_i^{\tau+1}\right)^2 &= \sum_{i=1}^{K}\left(w_i+\eta y\left(x_i-w_i y\right)\right)^2 \\
&= \sum_{i=1}^{K}\left(w_i^2+w_i\eta y\left(x_i-w_i y\right)+\eta^2 y^2\left(x_i-w_i y\right)^2\right) \\
&= \sum_{i=1}^{K} w_i^2+\sum_{i=1}^{K} w_i\eta y\left(x_i-w_i y\right)+\sum_{i=1}^{K}\eta^2 y^2\left(x_i-w_i y\right)^2 \\
&= \sum_{i=1}^{K} w_i^2+\eta y\sum_{i=1}^{K} w_i\left(x_i-w_i y\right)+\eta^2\sum_{i=1}^{K} y^2\left(x_i-w_i y\right)^2
\end{aligned} \tag{C.47}$$

Because $\eta^2 \lll \eta$ we can omit the third term:

$$\begin{aligned}
\sum_{i=1}^{K}\left(w_i^{\tau+1}\right)^2 &\approx \sum_{i=1}^{K} w_i^2+\eta y\sum_{i=1}^{K} w_i\left(x_i-w_i y\right) \\
&= \sum_{i=1}^{K} w_i^2+\eta y\sum_{i=1}^{K}\left\{w_i x_i-w_i^2 y\right\} \\
&= \sum_{i=1}^{K} w_i^2+\eta y\sum_{i=1}^{K} w_i x_i-\eta y\sum_{i=1}^{K} w_i^2 y \\
&= \sum_{i=1}^{K} w_i^2+\eta y^2-\eta y^2\sum_{i=1}^{K} w_i^2 \\
&= \sum_{i=1}^{K} w_i^2+\eta y^2\left(1-\sum_{i=1}^{K} w_i^2\right)
\end{aligned} \tag{C.48}$$

Since $\eta > 0$ and $y^2 > 0$ we know that when the squared weight vector length was less than 1 in the previous time step, the square weight vector will become larger and when the square weight vector length was greater than 1 the square weight vector will become zero. The same must hold for the weight vector itself, since $\sqrt{1} = 1$.[1] In effect we home in on a unit vector length.

Thus we have proved that Oja's rule will cause the weight vector to converge to weight vector of unit length which maximizes the output variance, which is the first principal component.

C.12.2 *Further Principal Components*

Above we have proved the base case of the inductive proof that GHA will cause the weight vectors of a neural network layer to converge to the principal components. Now we deduce the inductive step; provided that we have already proved the first $k-1$ weight vectors to be converged to principal components, we have to prove that $\mathbf{w}_k$ converges to the next principal component.

We do this by showing that the GHA can be expected to perform a Gram-Schmidt orthogonalization. When using the GHA when a random input sample (sampled from the uniform distribution over the whole data set) we compute the expected value of the update. For simplicity of the derivation we divide both sides by η.

1 Also the square root function is monotonically increasing.

$$
\begin{aligned}
&\mathbb{E}\left[\Delta w_{i\multimap k}\right]/\eta \\
=&\mathbb{E}\left[y_k\left(x_i-\sum_{j=1}^{k}w_{i\multimap j}y_j\right)\right] \\
=&\mathbb{E}\left[y_k\left(x_i-\sum_{j=1}^{k-1}w_{i\multimap j}y_j\right)-\left(y_k\right)^2 w_{i\multimap k}\right] \\
=&\mathbb{E}\left[y_k\left(x_i-\sum_{j=1}^{k-1}w_{i\multimap j}y_j\right)\right]-\mathbb{E}\left[\left(y_k\right)^2 w_{i\multimap k}\right] \\
=&\mathbb{E}\left[\sum_{h=1}^{D}\left\{x_h w_{h\multimap k}\right\}\left(x_i-\sum_{j=1}^{k-1}\left\{w_{i\multimap j}\sum_{h=1}^{D}\left\{x_h w_{h\multimap j}\right\}\right\}\right)\right] \\
&-\mathbb{E}\left[\left(y_k\right)^2 w_{i\multimap k}\right] \\
=&\mathbb{E}\left[\sum_{h=1}^{D}x_h w_{h\multimap k}x_i\right] \\
&-\mathbb{E}\left[\sum_{j=1}^{k-1}w_{i\multimap j}\sum_{h=1}^{D}\sum_{g=1}^{D}x_h w_{h\multimap k}x_g w_{g\multimap j}\right]-\mathbb{E}\left[\left(y_k\right)^2 w_{i\multimap k}\right] \\
=&\sum_{h=1}^{D}\mathbb{E}\left[x_h w_{h\multimap k}x_i\right]-\sum_{j=1}^{k-1}w_{i\multimap j}\sum_{h=1}^{D}\sum_{g=1}^{D}\mathbb{E}\left[x_h w_{h\multimap k}x_g w_{g\multimap j}\right] \\
&-\mathbb{E}\left[\left(y_k\right)^2 w_{i\multimap k}\right] \\
=&\sum_{h=1}^{D}w_{h\multimap k}\mathbf{\Sigma}_{hi}-\sum_{j=1}^{k-1}w_{i\multimap j}\sum_{g,h=1}^{D}w_{h\multimap k}\mathbf{\Sigma}_{gh}w_{g\multimap j} \\
&-\mathbb{E}\left[\left(y_k\right)^2 w_{i\multimap k}\right] \qquad \text{(C.49)}
\end{aligned}
$$

$$
\mathbb{E}\left[\Delta\mathbf{w}_k\right]/\eta=\mathbf{\Sigma}\mathbf{w}_k-\sum_{j=1}^{k}\left(\mathbf{w}_j^{\mathsf{T}}\mathbf{\Sigma}\mathbf{w}_k\right)\mathbf{w}_j-\mathbb{E}\left[\operatorname{diag}(\mathbf{y})\mathbf{y}\operatorname{diag}(\mathbf{w}_k)\right] \qquad \text{(C.50)}
$$

where

- D is the input dimensionality;
- $\mathbf{w}_k$ is the vector of weights on connections to the neuron corresponding to y_k;
- $\mathbf{\Sigma}$ is the covariance matrix of the input.

Here the first two terms are equivalent to the orthonormalization of $\mathbf{\Sigma}\mathbf{w}_k$ to the first $k-1$ weight vectors. (See formula 3.14.) We therefore rewrite it to:

$$
\mathbb{E}\left[\Delta\mathbf{w}_k\right]/\eta=\left(\mathbf{\Sigma}\mathbf{w}_k\right)^{\perp}-\mathbb{E}\left[\operatorname{diag}(\mathbf{y})\mathbf{y}\operatorname{diag}(\mathbf{w}_k)\right] \qquad \text{(C.51)}
$$

We can prove the distributivity of perpendicularity, i.e. we can prove that $(\mathbf{\Sigma w}_k)^{\perp} = \mathbf{\Sigma w}_k^{\perp}$:

$$\begin{aligned}
(\mathbf{\Sigma w}_k)^{\perp} &= \mathbf{\Sigma w}_k - \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}} \mathbf{\Sigma w}_k\right) \mathbf{w}_j \\
&= \mathbf{\Sigma w}_k - \sum_{j=1}^{k-1} \left(\left(\mathbf{\Sigma w}_j\right)^{\mathsf{T}} \mathbf{w}_k\right) \mathbf{w}_j \\
&= \mathbf{\Sigma w}_k - \sum_{j=1}^{k-1} \left(\left(\lambda_j \mathbf{w}_j\right)^{\mathsf{T}} \mathbf{w}_k\right) \mathbf{w}_j \\
&= \mathbf{\Sigma w}_k - \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}} \mathbf{w}_k\right) \lambda_j \mathbf{w}_j \\
&= \mathbf{\Sigma w}_k - \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}} \mathbf{w}_k\right) \mathbf{\Sigma w}_j \\
&= \mathbf{\Sigma w}_k - \mathbf{\Sigma} \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}} \mathbf{w}_k\right) \mathbf{w}_j \\
&= \mathbf{\Sigma} \left(\mathbf{w}_k - \sum_{j=1}^{k-1} \left(\mathbf{w}_j^{\mathsf{T}} \mathbf{w}_k\right) \mathbf{w}_j\right) \\
&= \mathbf{\Sigma} \left(\mathbf{w}_k\right)^{\perp}
\end{aligned} \tag{C.52}$$

The left term of formula C.51 thus causes the weight update to be in the direction orthogonal to all previous principal components, while the right term constitutes a weight decay term. [2] Any component of the weight not orthogonal to the previous weight vectors will in effect decay toward zero, while the decay of a component which *is* orthogonal might be countered by the left term. The weight vector is therefore bound to end up in a subspace which is orthogonal to all preceding principal components.

We can further derive:

2 Since $\mathbf{\Sigma}$ is positive-definite, $\mathbf{w}_k^{\mathsf{T}} \mathbf{\Sigma w}_k$ is always positive, so that the right term of formula C.51 will constitute a weight decay term.

$$\mathbb{E}\left[\Delta \mathbf{w}_k\right] / \eta = \mathbf{\Sigma} \mathbf{w}_k^{\perp} - \mathbb{E}\left[\operatorname{diag}(\mathbf{y}) \mathbf{y} \operatorname{diag}(\mathbf{w}_k)\right] \tag{C.53}$$

$$\begin{aligned} \mathbb{E}\left[\Delta w_{i \multimap k}\right] / \eta &= \sum_{h=1}^{D} w_{h \multimap k}^{\perp} \mathbf{\Sigma}_{hi} - \mathbb{E}\left[(y_k)^2 w_{i \multimap k}\right] \\ &= \sum_{h=1}^{D} \mathbb{E}\left[x_h w_{h \multimap k}^{\perp} x_i\right] - \mathbb{E}\left[(y_k)^2 w_{i \multimap k}\right] \\ &= \mathbb{E}\left[y_k^{\perp} x_i - (y_k)^2 w_{i \multimap k}\right] \end{aligned} \tag{C.54}$$

where

- $w_{h \multimap k}^{\perp}$ is element h from the orthogonalized weight vector $\mathbf{w}_k^{\perp}$;
- $y_k^{\perp}$ is the component of output k within the orthogonal subspace.

When restricted to the orthogonal subspace the above formula reduces to Oja's rule restricted to the orthogonal subspace. Sanger's rule will thus behave as Oja's rule within the subspace orthogonal to all preceding principal components.[17, p. 209] When converged, the weight vector will therefor maximize the variance, subject to the constraint of being orthonormal to all previous weight vectors, which is to say that the weight vector converges to the next principal component.

Note that even though the inductive proof applies to a sequential procedure, where we let each neuron converge after we start learning the next, the result must be the same for a concurrent procedure. When we learn all weight vectors simultaneously, they will still converge to the principal components, albeit consecutively. Any weight vector can only be converged when all weight vectors of preceding neurons have converged and so the conditions for the proof of the iterative procedure apply.

C.13 SOLVING MULTIPLE LAGRANGIAN CONSTRAINTS GENERICALLY

The *Lagrangian function* and its partial derivatives are given by:

$$L(\mathbf{w}, \Lambda) = E(\mathbf{w}) + \sum_{c \in C} \lambda_c c \tag{C.55}$$

$$\frac{\partial L(\mathbf{w},\Lambda)}{\partial w_i} = \frac{\partial E(\mathbf{w})}{\partial w_i} + \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_i} \tag{C.56}$$

$$\frac{\partial L(\mathbf{w},\Lambda)}{\partial \lambda_{c_n}} = c_n \tag{C.57}$$

where

- $E(\mathbf{w})$ is the function to be optimized;
- C is a set of constraints c which are met when $c = 0$;
- c_n is a constraint in C;
- λ_c is the Lagrange multiplier belonging to constraint c;
- Λ is the set of all Lagrange multipliers.

However, setting the derivatives of the Lagrangian to zero doesn't always help in solving the Lagrange multipliers. We therefore define a new function $\widetilde{L}$, which is zero exactly when all partial derivatives of the original Lagrangian function are zero. Moreover, it has a stationary point where the original Lagrangian has one. Our modified function will have minima when the original Lagrangian function has saddle points, maxima or minima.

$$\begin{aligned}\widetilde{L}(\mathbf{w}, \Lambda) &= \sum_j \left(\frac{\partial L(\mathbf{w},\Lambda)}{\partial w_j} \right)^2 + \sum_{c \in C} \left(\frac{\partial L(\mathbf{w},\Lambda)}{\partial \lambda_c} \right)^2 \\ &= \sum_j \left(\frac{\partial E(\mathbf{w})}{\partial w_j} + \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_j} \right)^2 + \sum_{c \in C} c^2 \end{aligned} \tag{C.58}$$

$$\begin{aligned}\frac{\partial \widetilde{L}(\mathbf{w},\Lambda)}{\partial w_i} &= 2 \sum_j \left(\frac{\partial E(\mathbf{w})}{\partial w_j} + \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_j} \right) \left(\frac{\partial^2 E(\mathbf{w})}{\partial w_j \partial w_i} + \sum_{c \in C} \lambda_c \frac{\partial^2 c}{\partial w_j \partial w_i} \right) \\ &\quad + 2 \sum_{c \in C} c \frac{\partial c}{\partial w_i} \end{aligned} \tag{C.59}$$

$$\begin{aligned}
\frac{\partial \widetilde{L}(\mathbf{w},\Lambda)}{\partial \lambda_{c_n}} &= \frac{\partial}{\partial \lambda_{c_n}} \left\{ \sum_j \left(\frac{\partial E(\mathbf{w})}{\partial w_j} + \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_j} \right)^2 + \sum_{c \in C} c^2 \right\} \\
&= \sum_j \left\{ 2 \left(\frac{\partial E(\mathbf{w})}{\partial w_j} + \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_j} \right) \frac{\partial c_n}{\partial w_j} \right\} \\
&= \sum_j \left\{ 2 \frac{\partial c_n}{\partial w_j} \frac{\partial E(\mathbf{w})}{\partial w_j} + 2 \frac{\partial c_n}{\partial w_j} \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_j} \right\} \\
&= 2 \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial E(\mathbf{w})}{\partial w_j} + 2 \sum_j \frac{\partial c_n}{\partial w_j} \sum_{c \in C} \lambda_c \frac{\partial c}{\partial w_j} \\
&= 2 \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial E(\mathbf{w})}{\partial w_j} + 2 \sum_{c \in C} \lambda_c \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial c}{\partial w_j} \qquad \text{(C.60)}
\end{aligned}$$

When $\widetilde{L}$ is minimal we have that:

$$\begin{aligned}
\frac{\partial \widetilde{L}(\mathbf{w},\Lambda)}{\partial \lambda_{c_n}} &= 0 \\
2 \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial E(\mathbf{w})}{\partial w_j} + 2 \sum_{c \in C} \lambda_c \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial c}{\partial w_j} &= 0 \\
\sum_{c \in C} \lambda_c \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial c}{\partial w_j} &= - \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial E(\mathbf{w})}{\partial w_j} \qquad \text{(C.61)}
\end{aligned}$$

This forms a system of linear equations; we want to know $\boldsymbol{\lambda}$ such that

$$\mathbf{K}\boldsymbol{\lambda} = \mathbf{o} \qquad \text{(C.62)}$$

where

- $\boldsymbol{\lambda}$ is a vector of the elements of Λ;
- $\mathbf{K}_{nm} = \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial c}{\partial w_j}$
- $\mathbf{o}_n = - \sum_j \frac{\partial c_n}{\partial w_j} \frac{\partial E(\mathbf{w})}{\partial w_j}$

C.13.1 *Generalized Hebbian Algorithm*

For Generalized Hebbian Algorithm we have:

$$E(y_k) = \frac{1}{2} y_k^2 \tag{C.63}$$

$$C_k = \Big\{ o_{jk} \mid j < k \Big\} \cup \{ n_k \} \tag{C.64}$$

$$o_{jk} = \sum_i w_{ij} w_{ik} = \mathbf{w}_j^\mathsf{T} \mathbf{w}_k \tag{C.65}$$

$$n_j = 1 - \sum_i w_{ij}^2 = 1 - \mathbf{w}_j^\mathsf{T} \mathbf{w}_j \tag{C.66}$$

$$\Lambda_k = \Big\{ \lambda_{jk} \mid j < k \Big\} \cup \{ \lambda_k \} \tag{C.67}$$

where

- $E(y_k)$ is the objective function of output neuron k;
- C_k is the set of constraints for output neuron k;
- o_{jk} is an orthogonalization constraint between the weight vectors of neuron j and k;
- n_j is a normalization constraint on the weight vector of neuron j;
- Λ_k is the set of Lagrange multipliers for output neuron k;
- λ_{jk} is the Lagrange multiplier for orthogonalization constraint o_{jk}, i.e. $\lambda_{jk} = \lambda_{o_{jk}}$;
- λ_j is the Lagrange multiplier for normalization constraint n_j, i.e. $\lambda_j = \lambda_{n_j}$.

Note that we define objective functions for each output neuron k. We want to maximize such an objective function with respect to the weights on connections to that neuron alone, i.e. $\mathbf{w}_k$. We therefore only use the weights $\mathbf{w}_k$ when substituting in the Lagrangian functions given above.

For each orthogonalization constraint o_{jk} we have a corresponding Lagrange multiplier given by λ_{jk} and for each normalization constraint n_j we have a corresponding Lagrange multiplier given by λ_j.

Partial derivatives with respect to the weights are given by:

$$\frac{\partial E(y_k)}{\partial w_{ij}} = \begin{cases} y_k x_i & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{C.68}$$

$$\frac{\partial o_{lk}}{\partial w_{ij}} = \begin{cases} w_{il} & \text{if } j = k \\ w_{ik} & \text{if } j = l \\ 0 & \text{otherwise} \end{cases} \tag{C.69}$$

$$\frac{\partial n_k}{\partial w_{ij}} = \begin{cases} -2w_{ik} & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{C.70}$$

Since we maximize the objective only w.r.t. the weights of neuron k, we can view the Lagrangian functions L and $\tilde{L}$ as functions over $\mathbf{w}_k$ and view the other weights as constants. We can then fill in the partial derivatives of the orthogonalization Lagrange multipliers in formula C.60 and simplify the resulting modified Lagrangian:

$$\begin{aligned} \frac{\partial \tilde{L}_k(\mathbf{w}_k, \Lambda_k)}{\partial \lambda_{ak}} =& \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial E(y_k)}{\partial w_{ik}} + \sum_{c \in C} \lambda_c \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial c_m}{\partial w_{ik}} \\ =& \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial E(y_k)}{\partial w_{ik}} \\ &+ \sum_{j<k} \lambda_{jk} \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial o_{jk}}{\partial w_{ik}} \\ &+ \lambda_k \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial n_k}{\partial w_{ik}} \\ =& \sum_i w_{ia} y_k x_i \\ &+ \lambda_{ak} \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial o_{ak}}{\partial w_{ik}} + \sum_{j<k \wedge j \neq a} \lambda_{jk} \sum_i \frac{\partial o_{ak}}{\partial w_{ik}} \frac{\partial o_{jk}}{\partial w_{ik}} \\ &+ \lambda_k \sum_i w_{ia} \left(-2w_{ik}\right) \\ =& y_k y_a + \lambda_{ak} \mathbf{w}_a^\mathsf{T} \mathbf{w}_a + \sum_{j<k, j \neq a} \lambda_{jk} \mathbf{w}_a^\mathsf{T} \mathbf{w}_j - 2\lambda_k \mathbf{w}_a^\mathsf{T} \mathbf{w}_k \end{aligned} \tag{C.71}$$

At the optimum the constraints are met; since we are trying to find the optimum we will suppose the parameters are in the feasible space. We also know that at the optimum, the partial derivative of the modified Lagrangian

w.r.t. any Lagrange multiplier must be zero. We can therefore simplify formula C.71 and derive an expression for λ_{ak}:

$$\begin{aligned}\frac{\partial \tilde{L}_k(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_{ak}} &= y_k y_a + \lambda_{ak} = 0 \\ \lambda_{ak} &= -y_k y_a \end{aligned} \tag{C.72}$$

Now we do the same for the normalization constraints; we fill in the partial derivatives of the normalization multipliers in formula C.60 and simplify:

$$\begin{aligned}\frac{\partial \tilde{L}_k(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_k} &= \sum_i \frac{\partial n_k}{\partial w_{ik}} \frac{\partial E(y_k)}{\partial w_{ik}} + \sum_{c\in C} \lambda_c \sum_i \frac{\partial n_k}{\partial w_{ik}} \frac{\partial c_m}{\partial w_{ik}} \\ &= \sum_i -2 w_{ik} y_k x_i + \sum_{j<k} \lambda_{jk} \sum_i \frac{\partial n_k}{\partial w_{ik}} \frac{\partial o_{jk}}{\partial w_{ik}} + \lambda_k \sum_i \frac{\partial n_k}{\partial w_{ik}} \frac{\partial n_k}{\partial w_{ik}} \\ &= -2y_k^2 + \sum_{j<k} \lambda_{jk} \sum_i -2 w_{ik} w_{ij} + \lambda_k \sum_i 4 w_{ik}^2 \\ &= -2y_k^2 - 2 \sum_{j<k} \lambda_{jk} \mathbf{w}_k^\mathsf{T} \mathbf{w}_j + 4\lambda_k \mathbf{w}_k^\mathsf{T} \mathbf{w}_k \end{aligned} \tag{C.73}$$

At the optimum, similar to the derivation of λ_{ak}, we then derive that

$$\begin{aligned}\frac{\partial \tilde{L}_k(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_k} &= -2y_k^2 + 4\lambda_k = 0 \\ \lambda_k &= \frac{1}{2} y_k^2 \end{aligned} \tag{C.74}$$

We substitute the Lagrange multipliers in the definition of the original Lagrangian of formula C.55 for the expressions derived above and simplify:

$$\begin{aligned}\frac{\partial L_k(\mathbf{w}_k,\Lambda_k)}{\partial w_i} &= \frac{\partial E(y_k)}{\partial w_{ik}} + \sum_{m=1}^{C} \lambda_m \frac{\partial c_m}{\partial w_i} \\ &= x_i y_k - \sum_{j<k} y_j y_k \frac{\partial o_{jk}}{\partial w_{ik}} + \frac{1}{2} y_k^2 \frac{\partial n_k}{\partial w_{ik}} \\ &= x_i y_k - \sum_{j<k} y_j y_k w_{ij} - \frac{1}{2} y_k^2 2 w_{ik} \\ &= x_i y_k - \sum_{j<k} y_j y_k w_{ij} - y_k^2 w_{ik} \\ &= y_k \left(x_i - \sum_{j<k} y_j w_{ij} - y_k w_{ik} \right) \\ &= y_k \left(x_i - \sum_{j\le k} y_j w_{ij} \right) \end{aligned} \tag{C.75}$$

C.14 DERIVATIVE OF EED

Recall that

$$D = \sqrt{\det |\mathbf{\Sigma}|} \tag{C.76}$$

where

- $\mathbf{\Sigma}$ is the covariance matrix of the output data;
- $\det |\mathbf{\Sigma}|$ denotes the determinant of $\mathbf{\Sigma}$.

The derivative is then given by:

$$\begin{aligned}
\frac{\partial D}{\partial p} &= \frac{\partial \sqrt{\det|\mathbf{\Sigma}|}}{\partial p} \\
&= \frac{\partial \sqrt{\det|\mathbf{\Sigma}|}}{\partial \mathbf{\Sigma}} \frac{\partial \mathbf{\Sigma}}{\partial p} \\
&= \frac{1}{2\sqrt{\det |\mathbf{\Sigma}|}} \det |\mathbf{\Sigma}| \left(\mathbf{\Sigma}^{-1}\right)^{\mathsf{T}} \frac{\partial \mathbf{\Sigma}}{\partial p} \\
&= \frac{1}{2} \sqrt{\det |\mathbf{\Sigma}|} \left(\mathbf{\Sigma}^{-1}\right)^{\mathsf{T}} \frac{\partial \mathbf{\Sigma}}{\partial p} \\
&= \frac{1}{2} \sqrt{\det |\mathbf{\Sigma}|} \left(\mathbf{\Sigma}^{\mathsf{T}}\right)^{-1} \frac{\partial \mathbf{\Sigma}}{\partial p} \qquad = \frac{1}{2} \sqrt{\det |\mathbf{\Sigma}|} \mathbf{\Sigma}^{-1} \frac{\partial \mathbf{\Sigma}}{\partial p}
\end{aligned} \tag{C.77}$$

In the above proof we made use of the well known derivative of the determinant of a matrix[31] and the fact that the covariance matrix is symmetric.

C.15 PRE-SIGMOID NORMAL DISTRIBUTION

By change of variables we derive that the post-sigmoid distribution of a pre-sigmoid normal distribution (for the logistic sigmoid) is given by:

$$\begin{aligned} f_Q(y) &= \left|\frac{\partial}{\partial y}\sigma^{-1}(y)\right| \cdot f_R(\sigma^{-1}(y)) \\ &= \frac{1}{y(1-y)} \cdot f(\sigma^{-1}(y); \mu, \sigma) \\ &= \frac{1}{y(1-y)} \cdot f\left(\log\left(\frac{1}{1-y} - 1\right); \mu, \sigma\right) \\ &= \frac{1}{y(1-y)} \cdot (\sigma\sqrt{2\pi})^{-1} \exp\left(-\frac{\left(\log\left(\frac{1}{1-y} - 1\right) - \mu\right)^2}{2\sigma^2}\right) \end{aligned}$$

where

- $R = \mathcal{N}(0, \sigma^*)$
- σ^* is the standard deviation of the pre-sigmoid distribution;
- $\sigma(\cdot)$ is the logistic sigmoid σ^l.

C.16 DERIVATIVE OF MEAN AND COVARIANCE

$$\frac{\partial \overline{o}}{\partial o_r} = \frac{\partial}{\partial o_r}\frac{1}{N}\sum_i o_i = \frac{1}{N} \tag{C.78}$$

$$\begin{aligned}
\frac{\partial \operatorname{var} o}{\partial o_r} &= \frac{\partial}{\partial o_r}\frac{1}{N}\sum_i (o_i - \overline{o})^2 = \frac{1}{N}\frac{\partial}{\partial o_r}\sum_i (o_i - \overline{o})^2 \\
&= \frac{1}{N}\sum_i \frac{\partial}{\partial o_r}(o_i - \overline{o})^2 = \frac{1}{N}\sum_i 2(o_i - \overline{o})\frac{\partial}{\partial o_r}(o_i - \overline{o}) \\
&= \frac{1}{N}\sum_i 2(o_i - \overline{o})(\frac{\partial o_i}{\partial o_r} - \frac{1}{N}) = \frac{1}{N}\sum_i 2(o_i - \overline{o})(o_i' - \frac{1}{N}) \\
&= \frac{1}{N}\sum_i \left\{2(o_i - \overline{o})o_i' - 2(o_i - \overline{o})\frac{1}{N}\right\} \\
&= \frac{1}{N}\left(\sum_i 2(o_i - \overline{o})o_i' - \sum_i 2(o_i - \overline{o})\frac{1}{N}\right) \\
&= \frac{1}{N}\left(\sum_i 2(o_i - \overline{o})o_i' - \frac{1}{N}\sum_i 2(o_i - \overline{o})\right) \\
&= \frac{1}{N}\left(\sum_i 2(o_i - \overline{o})o_i' - 2\frac{1}{N}\sum_i (o_i - \overline{o})\right) \\
&= \frac{1}{N}\left(\sum_i 2(o_i - \overline{o})o_i' - 2\frac{1}{N}\sum_i o_i - -2\frac{1}{N}N\overline{o}\right) \\
&= \frac{1}{N}\left(\sum_i 2(o_i - \overline{o})o_i' - 2\overline{o} + 2\overline{o}\right) \\
&= \frac{1}{N}\sum_i 2(o_i - \overline{o})o_i' \\
&= \frac{1}{N}2(o_r - \overline{o}) \\
&= \frac{2}{N}(o_r - \overline{o})
\end{aligned} \tag{C.79}$$

$$
\begin{aligned}
\frac{\partial \mathrm{Cov}[o,u]}{\partial o_r} &= \frac{\partial}{\partial o_r}\frac{1}{N}\sum_i (o_i-\overline{o})(u_i-\overline{u}) \\
&= \frac{1}{N}\sum_i \frac{\partial}{\partial o_r}\left\{(o_i-\overline{o})(u_i-\overline{u})\right\} \\
&= \frac{1}{N}\sum_i \frac{\partial}{\partial o_r}\left\{(o_i-\overline{o})\right\}(u_i-\overline{u}) \\
&= \frac{1}{N}\sum_i (o_i'-\frac{1}{N})(u_i-\overline{u}) \\
&= \frac{1}{N}\sum_i \left\{o_i'(u_i-\overline{u})-\frac{1}{N}(u_i-\overline{u})\right\} \\
&= \frac{1}{N}\sum_i o_i'(u_i-\overline{u})-\frac{1}{N}\sum_i \frac{1}{N}(u_i-\overline{u}) \\
&= \frac{1}{N}(u_i-\overline{u})-\frac{1}{N}\sum_i \frac{1}{N}(u_i-\overline{u}) \\
&= \frac{1}{N}(u_i-\overline{u})-\frac{1}{N}\sum_i \frac{1}{N}u_i-\frac{1}{N}\overline{u} \\
&= \frac{1}{N}(u_i-\overline{u})-\frac{1}{N}\left(\sum_i \frac{1}{N}u_i-\sum_i \frac{1}{N}\overline{u}\right) \\
&= \frac{1}{N}(u_i-\overline{u})-\frac{1}{N}\left(\frac{1}{N}\sum_i u_i-\frac{1}{N}\sum_i \overline{u}\right) \\
&= \frac{1}{N}(u_i-\overline{u})-\frac{1}{N}\left(\overline{u}-\frac{1}{N}N\overline{u}\right) \\
&= \frac{1}{N}(u_i-\overline{u})
\end{aligned}
\tag{C.80}
$$

C.17 PROBABILITY INTEGRAL TRANSFORM

For a random variable X with pdf f_x and CDF F_X we prove that $Y = F_X(X)$ has a uniform distribution:

$$\begin{aligned}
f_Y(x) &= f_{F_X(X)}(x) \\
&= \left|\frac{\partial}{\partial x} F_X^{-1}(y)\right| \cdot f_X(F_X^{-1}(x)) && \text{(Change of variables)} \\
&= \left|\frac{1}{F'_X(F_X^{-1}(y))}\right| \cdot f_X(F_X^{-1}(x)) && \text{(Inverse function theorem)} \\
&= \left|\frac{1}{f_X(F_X^{-1}(y))}\right| \cdot f_X(F_X^{-1}(x)) && (F'_X = f_X \text{ almost everywhere}) \\
&= \frac{1}{f_X(F_X^{-1}(y))} \cdot f_X(F_X^{-1}(x)) && \text{(non-negativityof pdf)} \\
&= 1 \text{ if } f_X(F_X^{-1}(y)) \neq 0 && \text{(C.81)}
\end{aligned}$$

where

- (Change of variables) presupposed that F_X is monotonic, which is true because it's a CDF;
- (Inverse differentiation) assumes monotonicity, which is true for the PDFs associated with the sigmoid transfer functions we consider.

C.18 DERIVATIVE OF PSGU OBJECTIVE

$$D_{\mathrm{KL}}(\mathcal{N}_0||\mathcal{N}_1) = \frac{1}{2}\left(\mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0) + (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^\mathsf{T}\boldsymbol{\Sigma}_1^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) - k - \ln\frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|}\right) \tag{C.82}$$

Our optimal distribution has $\forall i : \boldsymbol{\mu}_{0i} = 0$

$$D_{\mathrm{KL}}(\mathcal{N}_0||\mathcal{N}_1) = \frac{1}{2}\left(\mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0) + \boldsymbol{\mu}_1^\mathsf{T}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 - k - \ln\frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|}\right) \tag{C.83}$$

$$\begin{aligned}\frac{\partial D_{\mathrm{KL}}}{\partial \boldsymbol{\Sigma}_1} &= \frac{\partial}{\partial \boldsymbol{\Sigma}_1}\frac{1}{2}\left(\mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0) + \boldsymbol{\mu}_1^\mathsf{T}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 - k - \ln\frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|}\right) \\ &= \frac{1}{2}\left(\frac{\partial}{\partial \boldsymbol{\Sigma}_1}\mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0) + \frac{\partial}{\partial \boldsymbol{\Sigma}_1}\boldsymbol{\mu}_1^\mathsf{T}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 - \frac{\partial}{\partial \boldsymbol{\Sigma}_1}\ln\frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|}\right)\end{aligned} \tag{C.84}$$

Note that for a covariance matrix $\boldsymbol{\Sigma}$ it holds that: $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}^\mathsf{T}$, and $(\boldsymbol{\Sigma}^{-1})^\mathsf{T} = (\boldsymbol{\Sigma}^\mathsf{T})^{-1}$, so $(\boldsymbol{\Sigma}^{-1})^\mathsf{T} = \boldsymbol{\Sigma}^{-1}$.

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\Sigma}_1}\mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0) &= \frac{\partial}{\partial \boldsymbol{\Sigma}_1}\mathrm{tr}(\mathbf{I}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0) \\ &= -\left(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0\mathbf{I}\boldsymbol{\Sigma}_1^{-1}\right)^\mathsf{T} && \text{(cookbook 113)} \\ &= -\left(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0\boldsymbol{\Sigma}_1^{-1}\right)^\mathsf{T} \\ &= -(\boldsymbol{\Sigma}_1^{-1})^\mathsf{T}\boldsymbol{\Sigma}_0^\mathsf{T}(\boldsymbol{\Sigma}_1^{-1})^\mathsf{T} \\ &= -\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_0\boldsymbol{\Sigma}_1^{-1} && \text{(symmetricity)}\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\Sigma}_1}\boldsymbol{\mu}_1^\mathsf{T}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 &= -(\boldsymbol{\Sigma}_1^{-1})^\mathsf{T}\boldsymbol{\mu}_1\boldsymbol{\mu}_1^\mathsf{T}(\boldsymbol{\Sigma}_1^{-1})^\mathsf{T} && \text{(cookbook 55)} \\ &= -\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1\boldsymbol{\mu}_1^\mathsf{T}\boldsymbol{\Sigma}_1^{-1} && \text{(C.85)}\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\mu}_1}\boldsymbol{\mu}_1^\mathsf{T}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 &= \left(\boldsymbol{\Sigma}_1^{-1} + (\boldsymbol{\Sigma}_1^{-1})^\mathsf{T}\right)\boldsymbol{\mu}_1 && \text{(cookbook 73)} \\ &= \left(\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_1^{-1}\right)\boldsymbol{\mu}_1 \\ &= 2\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 && \text{(C.86)}\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{\Sigma}_1} \ln \frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|} &= \frac{\partial}{\partial \boldsymbol{\Sigma}_1} \left\{ \ln |\boldsymbol{\Sigma}_0| - \ln |\boldsymbol{\Sigma}_1| \right\} \\
&= \frac{\partial}{\partial \boldsymbol{\Sigma}_1} \left\{ - \ln |\boldsymbol{\Sigma}_1| \right\} \\
&= - \frac{\partial}{\partial \boldsymbol{\Sigma}_1} \ln |\boldsymbol{\Sigma}_1| \\
&= - \frac{1}{|\boldsymbol{\Sigma}_1|} \frac{\partial}{\partial \boldsymbol{\Sigma}_1} |\boldsymbol{\Sigma}_1| \\
&= - \frac{1}{|\boldsymbol{\Sigma}_1|} |\boldsymbol{\Sigma}_1| (\boldsymbol{\Sigma}_1^{-1})^\mathsf{T} \\
&= - (\boldsymbol{\Sigma}_1^{-1})^\mathsf{T} \\
&= - \boldsymbol{\Sigma}_1^{-1}
\end{aligned} \tag{C.87}$$

$$\begin{aligned}
\frac{\partial D_{\mathrm{KL}}}{\partial \boldsymbol{\Sigma}_1} &= \frac{1}{2} \left(\frac{\partial}{\partial \boldsymbol{\Sigma}_1} \operatorname{tr}(\boldsymbol{\Sigma}_1^{-1} \boldsymbol{\Sigma}_0) + \frac{\partial}{\partial \boldsymbol{\Sigma}_1} \boldsymbol{\mu}_1^\mathsf{T} \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\mu}_1 - \frac{\partial}{\partial \boldsymbol{\Sigma}_1} \ln \frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|} \right) \\
&= \frac{1}{2} \left(- \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\Sigma}_0 \boldsymbol{\Sigma}_1^{-1} - \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\mu}_1 \boldsymbol{\mu}_1^\mathsf{T} \boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_1^{-1} \right) \\
&= \frac{1}{2} \boldsymbol{\Sigma}_1^{-1} \left(- \boldsymbol{\Sigma}_0 \boldsymbol{\Sigma}_1^{-1} - \boldsymbol{\mu}_1 \boldsymbol{\mu}_1^\mathsf{T} \boldsymbol{\Sigma}_1^{-1} + \mathbf{I} \right) \\
&= \frac{1}{2} \boldsymbol{\Sigma}_1^{-1} \left(\mathbf{I} - \left(\boldsymbol{\Sigma}_0 + \boldsymbol{\mu}_1 \boldsymbol{\mu}_1^\mathsf{T} \right) \boldsymbol{\Sigma}_1^{-1} \right)
\end{aligned} \tag{C.88}$$

Since the optimal $\boldsymbol{\Sigma}_0 = c\mathbf{I}$, we get

$$\frac{\partial D_{\mathrm{KL}}}{\partial \boldsymbol{\Sigma}_1} = \frac{1}{2} \boldsymbol{\Sigma}_1^{-1} \left(\mathbf{I} - \left(c\mathbf{I} + \boldsymbol{\mu}_1 \boldsymbol{\mu}_1^\mathsf{T} \right) \boldsymbol{\Sigma}_1^{-1} \right) \tag{C.89}$$

$$\begin{aligned}
\frac{\partial D_{\mathrm{KL}}}{\partial \boldsymbol{\mu}_1} &= \frac{1}{2} \left(\frac{\partial}{\partial \boldsymbol{\mu}_1} \operatorname{tr}(\boldsymbol{\Sigma}_1^{-1} \boldsymbol{\Sigma}_0) + \frac{\partial}{\partial \boldsymbol{\mu}_1} \boldsymbol{\mu}_1^\mathsf{T} \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\mu}_1 - \frac{\partial}{\partial \boldsymbol{\mu}_1} \ln \frac{|\boldsymbol{\Sigma}_0|}{|\boldsymbol{\Sigma}_1|} \right) \\
&= \frac{1}{2} \left(\frac{\partial}{\partial \boldsymbol{\mu}_1} \boldsymbol{\mu}_1^\mathsf{T} \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\mu}_1 \right) \\
&= \frac{1}{2} \left(2 \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\mu}_1 \right) \\
&= \boldsymbol{\Sigma}_1^{-1} \boldsymbol{\mu}_1
\end{aligned} \tag{C.90}$$

C.19 CONVOLUTIONAL HEBBIAN ALGORITHM

We will use location indices l:

- $l = (x, y)$
- $(x, y) < (\mathrm{w}, \mathrm{h})$ is short for $0 \leq x < \mathrm{w} \wedge 0 \leq y < \mathrm{h}$
- $(x_1, y_1) + (x_2, y_2)$ is short for $(x_1 + x_2, y_1 + y_2)$

For a convolution layer with pooling and linear transfer function and no bias we have:

$$p^k_{l_\mathrm{f}} = s\left(Z^k_{(x_\mathrm{f}, y_\mathrm{f})}\right) = s\left(Z^k_{l_\mathrm{f}}\right) \tag{C.91}$$

$$Z^k_{l_\mathrm{f}} = \left\{a^k_{l_\mathrm{f}+l} \mid l < (\mathrm{w_p}, \mathrm{h_p})\right\} \tag{C.92}$$

$$\forall a^k_{l_\mathrm{p}} \in Z^k_{l_\mathrm{f}} : a^k_{l_\mathrm{p}} = \sum_{l_\mathrm{i} < (\mathrm{w}c, \mathrm{h}c), z < N} i^z_{l_\mathrm{p}+l_\mathrm{i}} w^k_{(l_\mathrm{i}, z)} + b_k \tag{C.93}$$

where

- i, a and p are neurons representing input, convoluted activation and pooled output;
- $i^z_{(x,y)}$ is the input on location (x, y) in the map of feature z;
- $a^k_{(x,y)}$ is the activation and output on location (x, y) in the convolution map of neuron k;
- $p^k_{(x,y)}$ is the output on location (x, y) in the pooling map of feature k;
- N is the number of input maps, i.e. the number of input features;
- x_f and y_f are the coordinates of the final output neuron in the output map;
- $Z^k_{l_\mathrm{f}}$ is the pool of neurons connected to the final output neuron;
- b_k is the bias of weight configuration k;
- $\mathrm{w_p}$ and $\mathrm{h_p}$ are the width and height of the pool;
- w_c and h_c are the width and height of the weight configuration k;
- $w^k_{((x,y),z)}$ is the weight on the connection from feature z at a relative location (x, y) for weight configuration k.

Note that when using a linear transfer function, the activation of a convoluted neuron is equal to its output.

$$E(p_{l_\mathrm{f}}^k) = \frac{1}{2}\left(p_{l_\mathrm{f}}^k - \mu_k\right)^2 \tag{C.94}$$

$$C_k = \left\{o_{jk} \mid j < k\right\} \cup \{n_k\} \tag{C.95}$$

$$o_{jk} = \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w_{(l_\mathrm{i},z)}^j w_{(l_\mathrm{i},z)}^k = \mathbf{w}_j^\mathsf{T}\mathbf{w}_k \tag{C.96}$$

$$n_j = 1 - \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \left(w_{(l_\mathrm{i},z)}^j\right)^2 = 1 - \mathbf{w}_j^\mathsf{T}\mathbf{w}_j \tag{C.97}$$

$$\Lambda_k = \left\{\lambda_{jk} \mid j < k\right\} \cup \{\lambda_k\} \tag{C.98}$$

where

- $E(p_{l_\mathrm{f}}^k)$ is the objective function of output neuron k;
- C_k is the set of constraints for output neuron k;
- o_{jk} is an orthogonalization constraint between the weight vectors of neuron j and k;
- n_j is a normalization constraint on the weight vector of neuron j;
- $\mathbf{w}_j$ is the vector containing all weights of weight configuration j;
- Λ_k is the set of Lagrange multipliers for output neuron k;
- λ_{jk} is the Lagrange multiplier for orthogonalization constraint o_{jk}, i.e. $\lambda_{jk} = \lambda_{o_{jk}}$;
- λ_j is the Lagrange multiplier for normalization constraint n_j, i.e. $\lambda_j = \lambda_{n_j}$.

For each orthogonalization constraint o_{jk} we have a corresponding Lagrange multiplier given by λ_{jk} and for each normalization constraint n_j we have a corresponding Lagrange multiplier given by λ_j.

Partial derivatives with respect to the weights are given by:

$$\frac{\partial E(p_{l_\mathrm{f}}^k)}{\partial w_{(l_\mathrm{i},z)}^j} = \begin{cases} \dfrac{\partial E(p_{l_\mathrm{f}}^k)}{\partial p_{l_\mathrm{f}}^k} \displaystyle\sum_{a^k \in Z_{l_\mathrm{f}}^k} \frac{\partial p_{l_\mathrm{f}}^k}{\partial a^k} \frac{\partial a^k}{\partial w_{(l_\mathrm{i},z)}^j} & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{C.99}$$

$$\frac{\partial a_{l_\mathrm{p}}^k}{\partial w_{(l_\mathrm{i},z)}^j} = i_{l_\mathrm{p}+l_\mathrm{i}}^z \tag{C.100}$$

$$\frac{\partial E(p_{l_\mathrm{f}}^k)}{\partial w_{(l_\mathrm{i},z)}^k} = \left(p_{l_\mathrm{f}}^k - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p_{l_\mathrm{f}}^k}{\partial a_{l_\mathrm{f}+l_\mathrm{p}}^k} i_{l_\mathrm{f}+l_\mathrm{p}+l_\mathrm{i}}^z \tag{C.101}$$

$$\frac{\partial o_{lk}}{\partial w^j_{(l_\mathrm{i},z)}} = \begin{cases} w^l_{(l_\mathrm{i},z)} & \text{if } j = k \\ w^k_{(l_\mathrm{i},z)} & \text{if } j = l \\ 0 & \text{otherwise} \end{cases} \tag{C.102}$$

$$\frac{\partial n_k}{\partial w^j_{(l_\mathrm{i},z)}} = \begin{cases} -2w^k_{(l_\mathrm{i},z)} & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{C.103}$$

We can fill in the partial derivatives of the orthogonalization Lagrange multipliers in formula C.60 and simplify:

$$\begin{aligned} \frac{\partial \widetilde{L}_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_{gk}} =& \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial E(p^k_{l_\mathrm{f}})}{\partial w^k_{(l_\mathrm{i},z)}} \\ &+ \sum_{c\in C_k} \lambda_c \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial c_m}{\partial w^k_{(l_\mathrm{i},z)}} \\ =& \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial E(p^k_{l_\mathrm{f}})}{\partial w^k_{(l_\mathrm{i},z)}} \\ &+ \sum_{j<k} \lambda_{jk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial o_{jk}}{\partial w^k_{(l_\mathrm{i},z)}} \\ &+ \lambda_k \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \\ =& t_1 + t_2 + t_3 \end{aligned} \tag{C.104}$$

where

- M is the number of output maps, i.e. the number of weight configurations.

$$\begin{aligned} t_1 =& \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial E(p^k_{l_\mathrm{f}})}{\partial w^k_{(l_\mathrm{i},z)}} \\ =& \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w^g_{(l_\mathrm{i},z)} \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} i^z_{l_\mathrm{f}+l_\mathrm{p}+l_\mathrm{i}} \\ =& \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w^g_{(l_\mathrm{i},z)} i^z_{l_\mathrm{f}+l_\mathrm{p}+l_\mathrm{i}} \\ =& \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^g_{l_\mathrm{f}+l_\mathrm{p}} \end{aligned} \tag{C.105}$$

$$
\begin{aligned}
t_2 &= \sum_{j<k} \lambda_{jk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial o_{jk}}{\partial w^k_{(l_\mathrm{i},z)}} \\
&= \sum_{j<k,j\neq g} \lambda_{jk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial o_{jk}}{\partial w^k_{(l_\mathrm{i},z)}} \\
&\quad + \lambda_{gk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \\
&= \sum_{j<k} \lambda_{jk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w^g_{(l_\mathrm{i},z)} w^j_{(l_\mathrm{i},z)} \\
&\quad + \lambda_{gk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \left(w^g_{(l_\mathrm{i},z)}\right)^2 \\
&= \sum_{j<k} \lambda_{jk} \mathbf{w}_a^\mathsf{T} \mathbf{w}_j + \lambda_{gk} \mathbf{w}_g^\mathsf{T} \mathbf{w}_g
\end{aligned} \tag{C.106}
$$

$$
\begin{aligned}
t_3 =& \lambda_k \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial o_{gk}}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \\
=& \lambda_k \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w^g_{(l_\mathrm{i},z)} \left(-2 w^k_{(l_\mathrm{i},z)}\right) \\
=& -2\lambda_k \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w^g_{(l_\mathrm{i},z)} w^k_{(l_\mathrm{i},z)} \\
=& -2\lambda_k \mathbf{w}_a^\mathsf{T} \mathbf{w}_k
\end{aligned} \tag{C.107}
$$

$$
\begin{aligned}
\frac{\partial \widetilde{L}_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_{gk}} =& \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^g_{l_\mathrm{f}+l_\mathrm{p}} \\
&+ \sum_{j<k} \lambda_{jk} \mathbf{w}_a^\mathsf{T} \mathbf{w}_j + \lambda_{gk} \mathbf{w}_g^\mathsf{T} \mathbf{w}_g - 2\lambda_k \mathbf{w}_a^\mathsf{T} \mathbf{w}_k
\end{aligned} \tag{C.108}
$$

At the optimum the constraints are met; since we are trying to find the optimum we will suppose the parameters are in the feasible space. We also know that at the optimum, the partial derivative of the modified Lagrangian w.r.t. any Lagrange multiplier must be zero. We can therefore simplify formula C.108 and derive an expression for

$$
\frac{\partial \widetilde{L}_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_{gk}} = \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^g_{l_\mathrm{f}+l_\mathrm{p}} + \lambda_{gk} \tag{C.109}
$$

$$
\lambda_{gk} = -\left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^g_{l_\mathrm{f}+l_\mathrm{p}} \tag{C.110}
$$

Now we do the same for the normalization constraints; we fill in the partial derivatives of the normalization multipliers in formula C.60 and simplify:

$$
\begin{aligned}
&\frac{\partial \widetilde{L}_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_k} \\
&\qquad = \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial E(p^k_{l_\mathrm{f}})}{\partial w^k_{(l_\mathrm{i},z)}} \\
&\qquad\quad + \sum_{c\in C_k} \lambda_c \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial c}{\partial w^k_{(l_\mathrm{i},z)}} \\
&\qquad = \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial E(p^k_{l_\mathrm{f}})}{\partial w^k_{(l_\mathrm{i},z)}} \\
&\qquad\quad + \sum_{j<k} \lambda_{jk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial o_{jk}}{\partial w^k_{(l_\mathrm{i},z)}} \\
&\qquad\quad + \lambda_k \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \frac{\partial n_k}{\partial w^k_{(l_\mathrm{i},z)}} \\
&\qquad = \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} -2w^k_{(l_\mathrm{i},z)} \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} i^z_{l_\mathrm{f}+l_\mathrm{p}+l_\mathrm{i}} \\
&\qquad\quad + \sum_{j<k} \lambda_{jk} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} -2w^k_{(l_\mathrm{i},z)} w^j_{(l_\mathrm{i},z)} \\
&\qquad\quad + \lambda_k \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} 4\left(w^k_{(l_\mathrm{i},z)}\right)^2 \\
&\qquad = -2\left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} \sum_{l_\mathrm{i}<(\mathrm{w}_c,\mathrm{h}_c),z<N} w^k_{(l_\mathrm{i},z)} i^z_{l_\mathrm{f}+l_\mathrm{p}+l_\mathrm{i}} \\
&\qquad\quad - 2\sum_{j<k} \lambda_{jk} \mathbf{w}_k^\mathsf{T} \mathbf{w}_j + 4\lambda_k \mathbf{w}_k^\mathsf{T} \mathbf{w}_k \\
&\qquad = -2\left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^k_{l_\mathrm{f}+l_\mathrm{p}} \\
&\qquad\quad - 2\sum_{j<k} \lambda_{jk} \mathbf{w}_k^\mathsf{T} \mathbf{w}_j + 4\lambda_k \mathbf{w}_k^\mathsf{T} \mathbf{w}_k
\end{aligned}
\tag{C.111}
$$

At the optimum, similar to the derivation of λ_{gk}, we then derive that

$$
\begin{aligned}
\frac{\partial \widetilde{L}_{(l_\mathrm{f},k)}(\mathbf{w}_k,\Lambda_k)}{\partial \lambda_k} &= -2\left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^k_{l_\mathrm{f}+l_\mathrm{p}} + 4\lambda_k = 0 \\
\lambda_k &= \frac{1}{2}\left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p}<(\mathrm{w}_\mathrm{p},\mathrm{h}_\mathrm{p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^k_{l_\mathrm{f}+l_\mathrm{p}}
\end{aligned}
\tag{C.112}
$$

We fill substitute the expressions

$$\begin{aligned} L_{(l_\mathrm{f},k)}(\mathbf{w}_k, \Lambda_k) =& E(p^k_{l_\mathrm{f}}) + \sum_{c \in C_k} \lambda_c c \\ =& \frac{1}{2}\left(p^k_{l_\mathrm{f}} - \mu_k\right)^2 + \sum_{j<k} \lambda_{jk} o_{jk} + \lambda_k n_k \\ =& \frac{1}{2}\left(p^k_{l_\mathrm{f}} - \mu_k\right)^2 \\ & - \sum_{j<k} o_{jk}\left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p} < (\mathrm{w_p}, \mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^j_{l_\mathrm{f}+l_\mathrm{p}} \\ & + \frac{1}{2} n_k \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p} < (\mathrm{w_p}, \mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^k_{l_\mathrm{f}+l_\mathrm{p}} \end{aligned} \tag{C.113}$$

We can go further and derive:

$$\begin{aligned} =& \frac{1}{2}\left(p^k_{l_\mathrm{f}} - \mu_k\right)^2 \\ & - \sum_{j<k} \mathbf{w}_j^\mathsf{T} \mathbf{w}_k \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p} < (\mathrm{w_p}, \mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^j_{l_\mathrm{f}+l_\mathrm{p}} \\ & + \frac{1}{2}\left(1 - \mathbf{w}_k^\mathsf{T} \mathbf{w}_k\right) \left(p^k_{l_\mathrm{f}} - \mu_k\right) \sum_{l_\mathrm{p} < (\mathrm{w_p}, \mathrm{h_p})} \frac{\partial p^k_{l_\mathrm{f}}}{\partial a^k_{l_\mathrm{f}+l_\mathrm{p}}} a^k_{l_\mathrm{f}+l_\mathrm{p}} \end{aligned} \tag{C.114}$$

We substitute the Lagrange multipliers in the definition of the original Lagrangian of formula C.55 for the expressions derived above and simplify:

$$
\begin{aligned}
\frac{\partial L_{(l_{\mathrm{f}},k)}(\mathbf{w}_k,\Lambda_k)}{\partial w^k_{(l_{\mathrm{i}},z)}} =& \frac{\partial E(p^k_{l_{\mathrm{f}}})}{\partial w_{ik}} + \sum_{c\in C_k} \lambda_c \frac{\partial c}{\partial w^k_{(l_{\mathrm{i}},z)}} \\
=& \frac{\partial E(p^k_{l_{\mathrm{f}}})}{\partial w_{ik}} + \sum_{j<k} \lambda_{jk} \frac{\partial o_{jk}}{\partial w^k_{(l_{\mathrm{i}},z)}} + \lambda_k \frac{\partial n_k}{\partial w^k_{(l_{\mathrm{i}},z)}} \\
=& \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} i^z_{l_{\mathrm{f}}+l_{\mathrm{p}}+l_{\mathrm{i}}} \\
& - \sum_{j<k} \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^j_{l_{\mathrm{f}}+l_{\mathrm{p}}} \frac{\partial o_{jk}}{\partial w^k_{(l_{\mathrm{i}},z)}} \\
& + \frac{1}{2}\left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}} \frac{\partial n_k}{\partial w^k_{(l_{\mathrm{i}},z)}} \\
=& \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} i^z_{l_{\mathrm{f}}+l_{\mathrm{p}}+l_{\mathrm{i}}} \\
& - \sum_{j<k} \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^j_{l_{\mathrm{f}}+l_{\mathrm{p}}} w^j_{(l_{\mathrm{i}},z)} \\
& + \frac{1}{2}\left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}} \left(-2 w^k_{(l_{\mathrm{i}},z)}\right) \\
=& \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} i^z_{l_{\mathrm{f}}+l_{\mathrm{p}}+l_{\mathrm{i}}} \\
& - \sum_{j<k} \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^j_{l_{\mathrm{f}}+l_{\mathrm{p}}} w^j_{(l_{\mathrm{i}},z)} \\
& - \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}} w^k_{(l_{\mathrm{i}},z)} \\
=& \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} i^z_{l_{\mathrm{f}}+l_{\mathrm{p}}+l_{\mathrm{i}}} \\
& - \sum_{j\leq k} \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^j_{l_{\mathrm{f}}+l_{\mathrm{p}}} w^j_{(l_{\mathrm{i}},z)} \\
=& \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} i^z_{l_{\mathrm{f}}+l_{\mathrm{p}}+l_{\mathrm{i}}} \\
& - \left(p^k_{l_{\mathrm{f}}} - \mu_k\right) \sum_{j\leq k} w^j_{(l_{\mathrm{i}},z)} \sum_{l_{\mathrm{p}}<(\mathrm{w_p},\mathrm{h_p})} \frac{\partial p^k_{l_{\mathrm{f}}}}{\partial a^k_{l_{\mathrm{f}}+l_{\mathrm{p}}}} a^j_{l_{\mathrm{f}}+l_{\mathrm{p}}}
\end{aligned}
\tag{C.115}
$$

C.20 LINEARITY OF soft arg max

The soft approximation to the arg max of $f(\cdot)$ is given by:

$$\begin{aligned}\underset{z\in Z_j}{\text{soft arg max}}\, f(z) &= \sum_{z_k\in Z_j}\left\{z_k\frac{e^{pf(z_k)}}{\sum_{z_l\in Z_j}e^{pf(z_l)}}\right\}\\ &= \frac{1}{\sum_{z_k\in Z_j}e^{pf(z_k)}}\sum_{z_k\in Z_j}z_k e^{pf(z_k)}\end{aligned}\tag{C.116}$$

We derive:

$$\begin{aligned}c\,\underset{z\in Z_j}{\text{soft arg max}}\, f(z) &= c\frac{1}{\sum_{z_k\in Z_j}e^{pf(z_k)}}\sum_{z_k\in Z_j}z_k e^{pf(z_k)}\\ &= \frac{1}{\sum_{z_k\in Z_j}e^{pf(z_k)}}\sum_{z_k\in Z_j}cz_k e^{pf(z_k)}\end{aligned}\tag{C.117}$$

Let's suppose that $f(\cdot)$ distributes over multiplication: $f(cx) = f(c)f(x)$. This property holds for the linear function, the absolute function and the square function, which are commonly used functions for f. We can then derive that:

$$\begin{aligned}c\,\underset{z\in Z_j}{\text{soft arg max}}\, f(z) &= \frac{1}{\sum_{z_k\in Z_j}e^{pf(z_k)}}\sum_{z_k\in Z_j}cz_k e^{pf(z_k)}\\ &= \frac{1}{\sum_{z_k\in Z_j}e^{p\frac{1}{f(c)}f(c)f(z_k)}}\sum_{z_k\in Z_j}cz_k e^{p\frac{1}{f(c)}f(c)f(z_k)}\\ &= \frac{1}{\sum_{z_k\in Z_j}e^{\frac{p}{f(c)}f(cz_k)}}\sum_{z_k\in Z_j}cz_k e^{\frac{p}{f(c)}f(cz_k)}\end{aligned}\tag{C.118}$$

So for $p\mapsto\frac{p}{f(c)}$ we have that $c\,\underset{z\in Z_j}{\text{soft arg max}}\, f(z)\mapsto\underset{z\in Z_j}{\text{soft arg max}}\, f(cz)$

D

RESULTS

D.1 WEIGHT VECTOR IMAGES

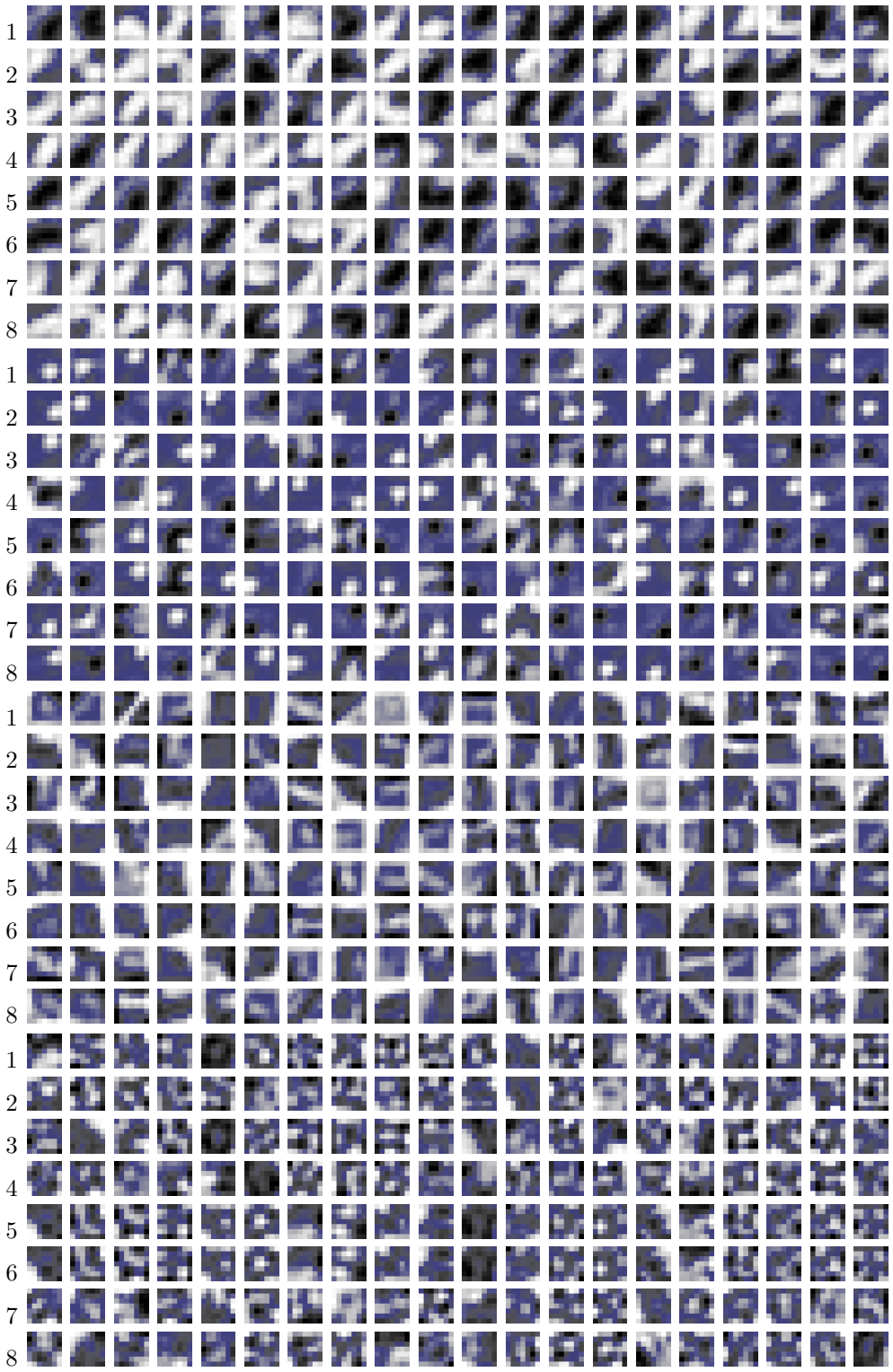


Figure D.1: Weight images for EED using settings 7⟩7} HP, 7⟩2} HP, 7⟩7} SP and 7⟩2} SP.

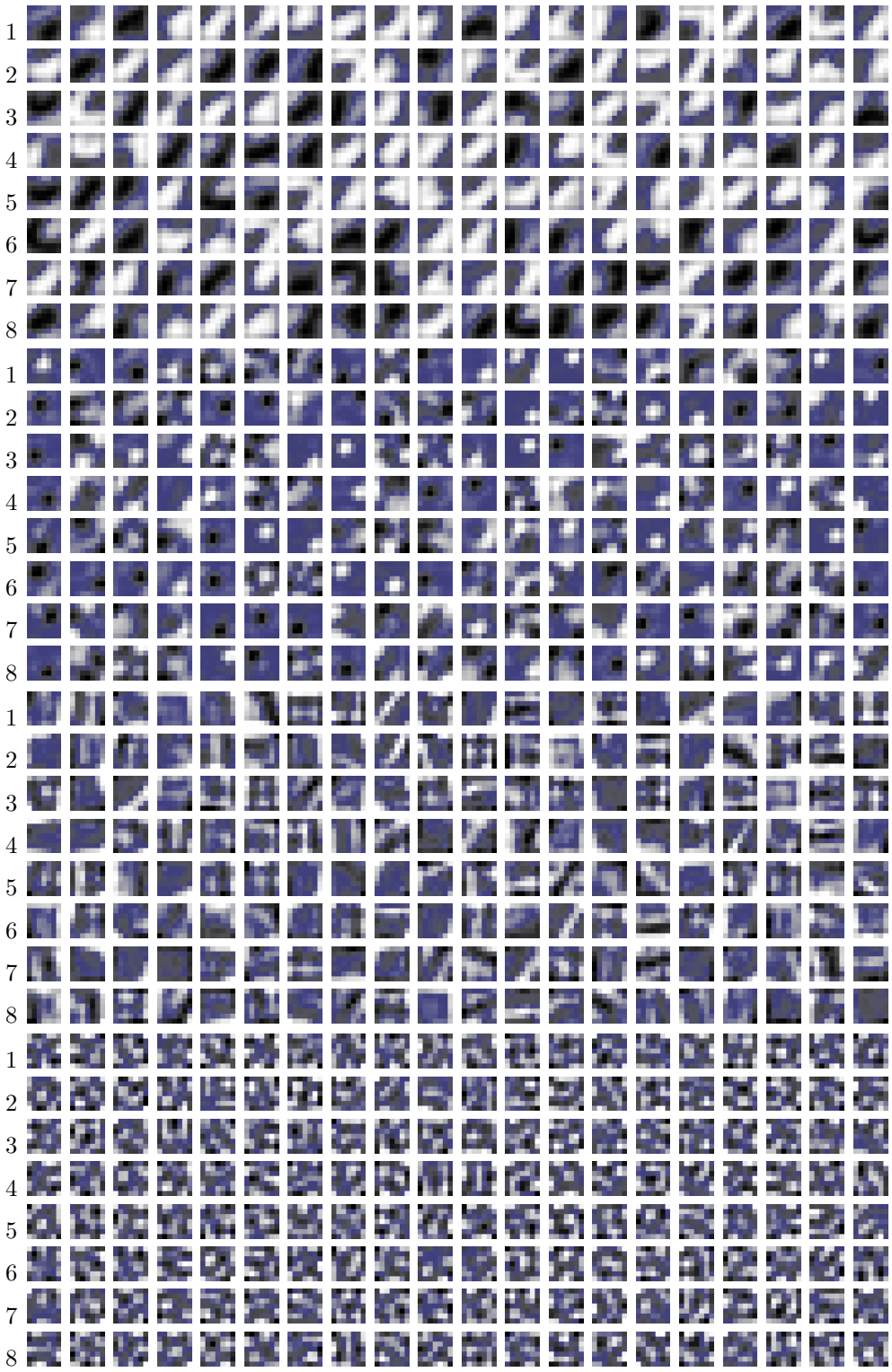


Figure D.2: Weight images for PSGU using settings 7⟩7} HP, 7⟩2} HP, 7⟩7} SP and 7⟩2} SP.

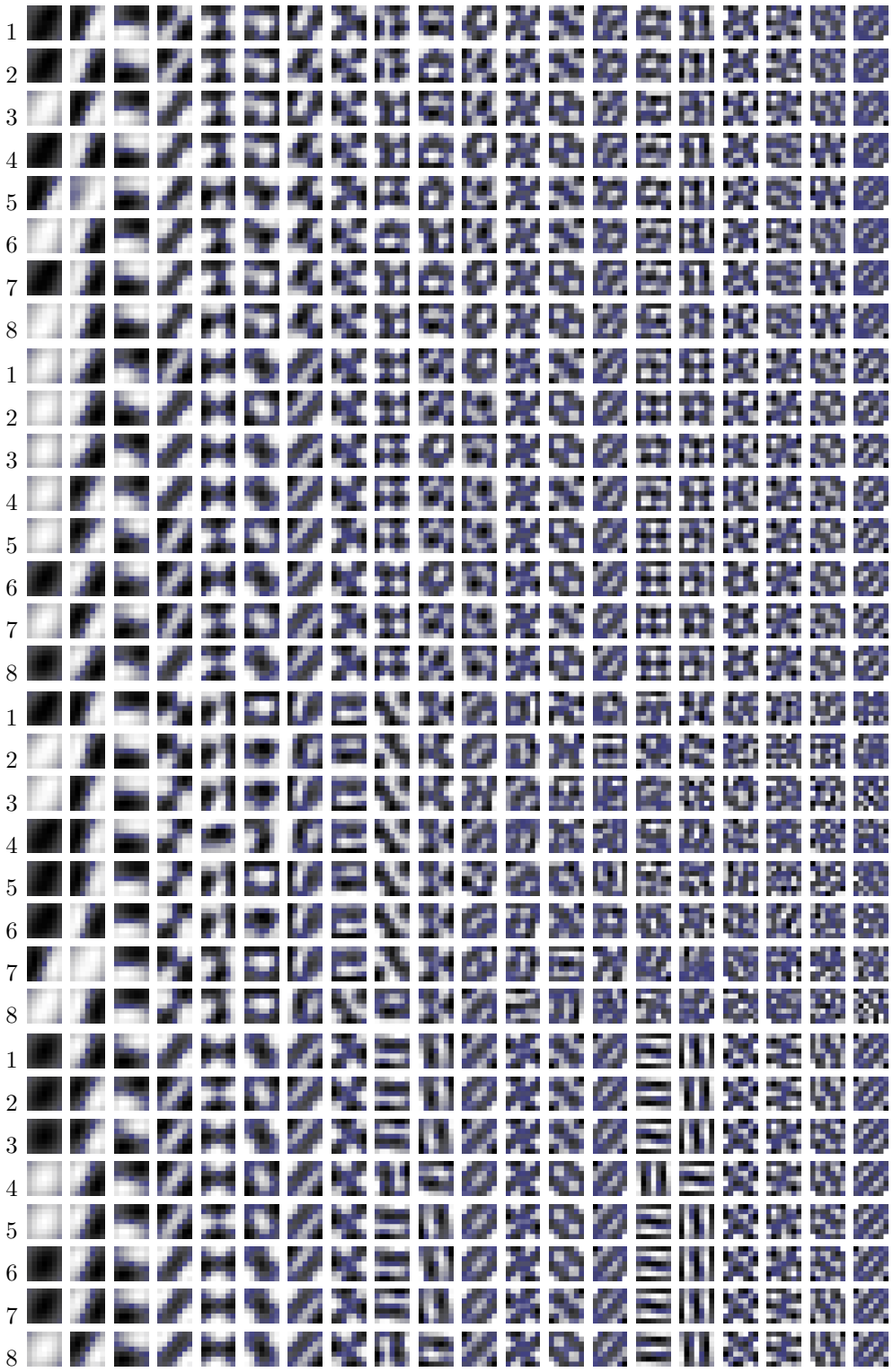


Figure D.3: Weight images for CHA-SM using settings 7⟩7} HP, 7⟩2} HP, 7⟩7} SP and 7⟩2} SP.

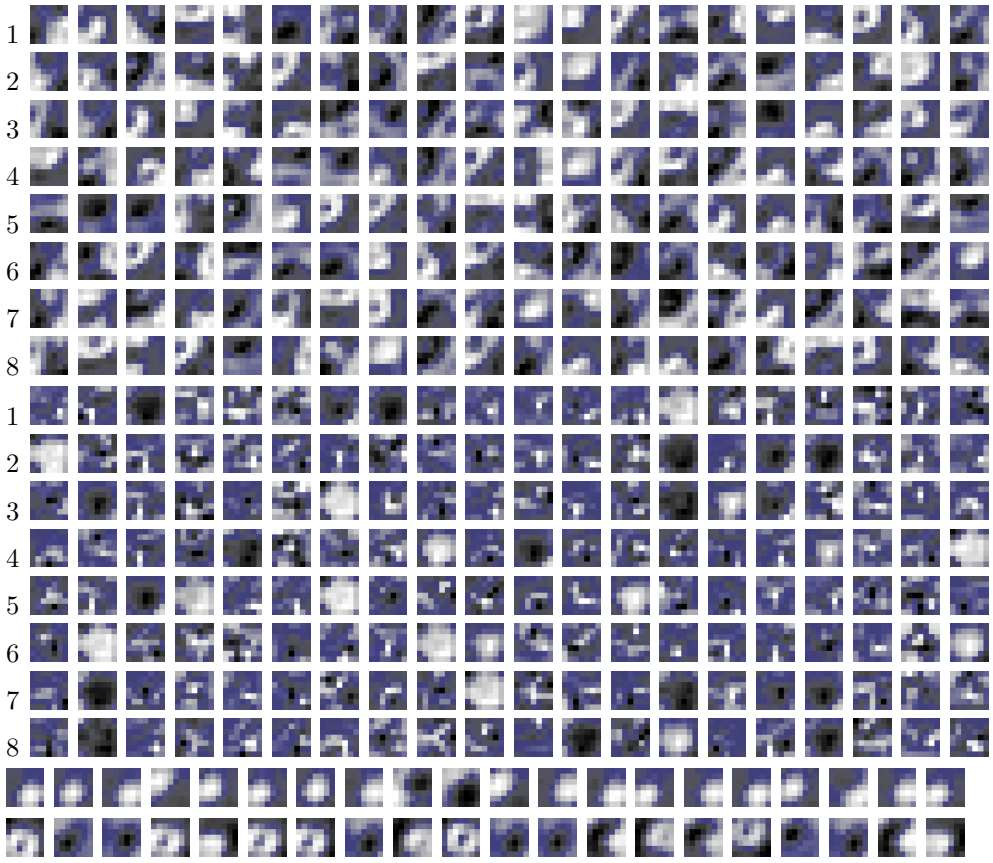


Figure D.4: Weight images for PCAEs using settings 7⟩7} HP, 7⟩2} HP, 7⟩7} SP (single run) and 7⟩2} SP (single run).

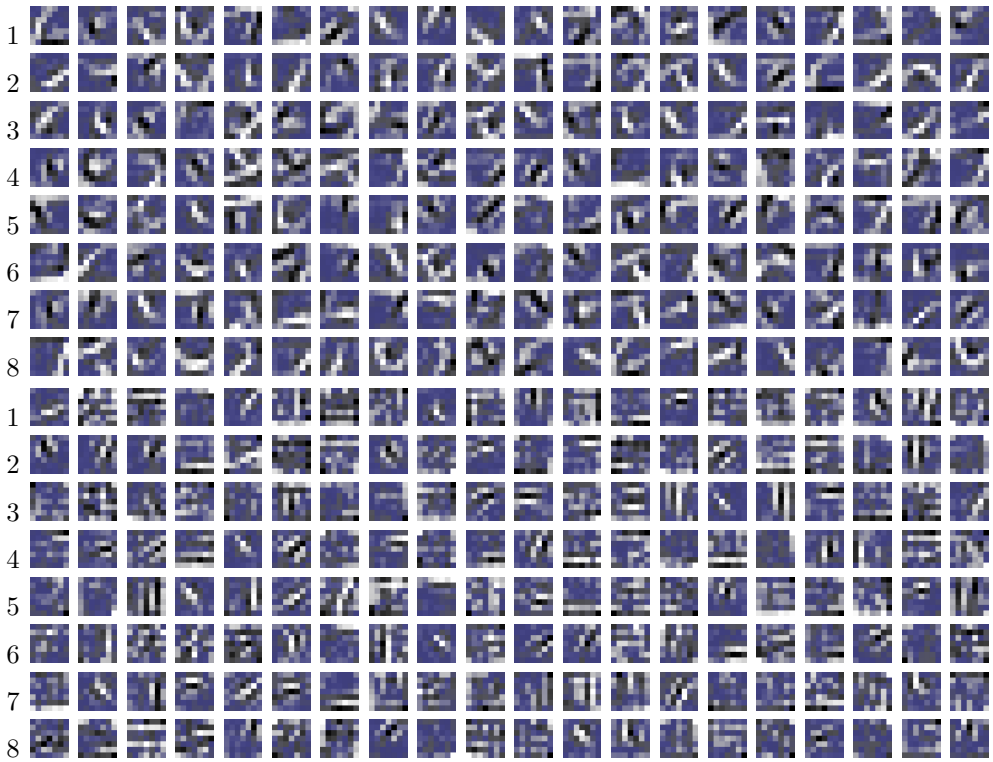


Figure D.5: Weight images for purely supervised learning using hard pooling and soft pooling.

D.2 PERFORMANCE

			Simple						**LeNet-1**		
			7⟩7}			7⟩2}			5⟩4}		
			PRE	P+P	POST	PRE	P+P	POST	PRE	P+P	POST
EED-BIN	HP	WD	-			-			-		
		X	8.4			6.89					
	SP	WD	-			-			-		
		X	51.57			48.35					
EED	HP	WD	-			-			-		
		X	5.14		4.31	5.5		5.94	23.37		7.1
	SP	WD	-			-			-		
		X	1.6		1.94	1.97		1.47	4.9		2.15
PSGU	HP	WD	-		6.7	-			-		
		X	4.7		4.87	7.57		5.89	19.42		7.4
	SP	WD	-			-			-		
		X	3.27			4.81			7.68		2.31
CHA-D	HP	WD	-			-			-		
		X	18.05			19.65					
	SP	WD	-			-			-		
		X	80.1			48.35					
CHA-U	HP	WD	-	15.36	14.71	-		17.1	-	6.53	6.16
		X	7.79	7.58	5.19	7.57	7.28	8.75	8.66	5.56	5.67
	SP	WD	-			-			-	4.35	9.01
		X	3.77	4.06	3.78	3.95	7.57	4.43	11.88	4.35	6.39
PCAE	HP	WD	-			-			-	6.16	17.1
		X	19.12			78.76			15.76	8.74	10.66
	SP	WD	-			-			-		
		X	3.7*		4.3*	2.29*		3.23*	6.34*	4.86*	6.67*
none	HP	WD	-	-	-	-	-	8.03	-	-	6.31
		X	-	-	-	-	-	29.43	-	-	5.67
	SP	WD	-	-	-	-	-	12	-	-	1.77
		X	-	-	-	-	-	10.91	-	-	6.39

Table D.1: Average error rate of each experimented setting. (*The experiments on PCAEs with soft pooling function have only been performed once or twice.)

			Simple						**LeNet-1**		
			7⟩7}			7⟩2}			5⟩4}		
			PRE	P+P	POST	PRE	P+P	POST	PRE	P+P	POST
EED-BIN	HP	WD	-			-			-		
		X	7.88			6.55					
	SP	WD	-			-			-		
		X	48.84			46.94					
EED	HP	WD	-			-			-		
		X	3.71		4.38	4.38		5.71	20.36		5.57
	SP	WD	-			-			-		
		X	1.3		1.57	1.59		1.31	4.52		1.66
PSGU	HP	WD	-		6.54	-			-		
		X	3.51		3.54	5.17		5.52	17.61		5.91
	SP	WD	-			-			-		
		X	3.1			4.58			6.94		2.02
CHA-D	HP	WD	-			-			-		
		X	15.11			18.5					
	SP	WD	-			-			-		
		X	73.26			46.94					
CHA-U	HP	WD	-	9.31	11.07	-		8.69	-	5.55	4.9
		X	3.82	4.61	3.65	5.17	4.95	5.4	8.21	4.93	4.88
	SP	WD	-			-			-	3.87	6.48
		X	2.72	2.76	2.95	3.23	5.17	3.86	9.02	3.87	4.87
PCAE	HP	WD	-			-			-	5.35	8.69
		X	11.37			74.4			13.67	8.04	9.05
	SP	WD	-			-			-		
		X	3.7*		4.3*	2.29*		3.23*	5.48*	4.39*	6.42*
none	HP	WD	-	-	-	-	-	6.95	-	-	5.98
		X	-	-	-	-	-	4.45	-	-	4.88
	SP	WD	-	-	-	-	-	9.26	-	-	1.44
		X	-	-	-	-	-	7.56	-	-	4.87

Table D.2: Error rate of the best run of each experimented setting. (*The experiments on PCAEs with soft pooling function have only been performed once or twice.)

INDEX