Faculty of Natural and Mathematical Sciences
Department of Informatics

The Strand
Strand Campus
London WC2R 2LS
Telephone 020 7848 **2145**
Fax 020 7848 2851

7CCSMPRJ

Individual Project Submission 2014/15

Name: Andrea Orru

Student Number: 1473018

Degree Programme: MSc Intelligent Systems

Project Title:
Design, Implementation and Testing of a Constructive Neural Network Algorithm

Word count: 10666

Supervisor: Michael Spratling

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☒ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: Andrea Orru

Date: 28/08/2015

Abstract

This paper presents a constructive version of the state-of-the-art Divisive Input Modulation algorithm for the unsupervised learning of image components. Nodes are added to the predictive layer dynamically and throughout the learning process on the basis of the reconstruction error calculated by the network. The proposed solution adapts methods from existing constructive algorithms to the negative feedback network architecture employed by the original DIM implementation. The algorithm that has been developed results in similar levels of accuracy, while at the same striving to build the smallest possible network. The conducted work thus represents a significant improvement over the original model in regards to training times and complexity of the neural network.

Acknowledgments

I would like to express my appreciation to professor **Spratling**, my supervisor, for his never-ending patience and availability when guiding me towards the completion of this project.

I will never be able to thank my best friend **Claudio** enough, as I would have never got this far without him. But I want to try anyway: thank you for all the chatting, the laughs, the *pasta parties*, and the advices, the insights, the constant inspiration. Most of all, thank you for being there all the time, making the good moments memorable and the terrible ones bearable. You never failed me: your loyalty redefined my whole concept of friendship. I owe you a life debt.

Infinite thanks to **Chiara**, *the other side of my brain*, the girl that knows what I will be thinking before I even have the time to think it myself, the only person I know that can take the big mess I am, make some sense out of it, and sometimes even fix it. You truly are unique.

Cheers to **Riccardo**: if there ever was a turning point in my life, that was following you in that crazy adventure that began one year ago and, I hope, will never end. You and I discovered the meaning of life, and is there anything more important than that?

Special thanks to **Francesco**, the demi-god (*son of Zeus* maybe?) who is apparently never tired of listening to me, who brought me here in the first place and who helped me become the person I am today. Without you, there would have been no Master at all!

Thanks to **Sid**, hero of the late-nighters and companion in *deadline-induced panic*. Thanks to **Gabriele** for all the encouragement. To **Andrea**, because *ertu*. To the yellow **Luana**. To that crazy dude that is **Nic**. To my *sadmate* **Myria**. And to **Melisa** (*daaarling!*).

Eternal gratitude to **my family**, for all the support they gave me and all the sacrifices they endured to make this year possible. Thank you for your trust, I will find a way to make you proud, I promise.

Finally, **Eva**. For the happiest days of my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Artificial neural networks in all their various incarnations have been successfully employed to solve a very wide range of machine learning problems, thanks to their good representational capabilities [25].

Theoretically, a simple feed-forward neural network with a single hidden layer and a sufficient number of neurons in that layer is an universal approximator [17, 19]. In practice however, too small networks may be unable to adequately learn the problem, while overly large networks tend to overfit the training data [22].

The problem of determining the optimal neural network topology is thus very important. However, there are currently no efficient methods to choose *a priori* the best network architecture for a given problem [22]. In real-world applications, *trial-and-error* and decisions based on previous knowledge about the problem are often the preferred approach. The chosen architecture has thus no guarantees to be the optimal one for the task [25].

To overcome this problem, solutions that involve learning both the weights of the synapses *and* the network topology have been suggested [22].

One of the proposed answers is the class of algorithms of the so-called *constructive neural networks*. The main idea behind it is starting from a minimal architecture and then adding hidden layers, nodes and connections during training [18, 25].

Such algorithms provides the flexibility to explore the space of neural network topologies in a controlled, *data-driven* fashion. Furthermore, because small solutions are found first, this method has the potential to discover near-minimal networks that approximately match the complexity of the learning task [22].

For all these reasons, applying a constructive approach to existing learning algorithms constitutes an interesting problem.

In the present project I worked on the development of a constructive version of the DIM[1] learning algorithm, as described in [30], and its derivation [28] used in the *non-linear* PC/BC model[2] [26]. The original algorithm has been developed to train a variation of *negative feedback networks* and has proven to be very successful in the unsupervised learning of image components, yielding state-of-the-art performances in various benchmarks, even in the presence of occlusion and overlapping [30].

On one hand, the main feature of negative feedback networks is the competition between nodes, which enables them to be selective for different input stimuli by making the individual synaptic weights more distinct. This is achieved by having inhibiting (*divisive*) feedback connections from the output neurons back to their input nodes [30].

On the other hand, the PC model proposes a hierarchical architecture in which alternating populations of *error-detecting* and *predicting* nodes interact with each other to carry out the perception process [29]. In particular, the higher levels of the network are not limited to passively receiving the input from the preceding nodes, but instead they actively predict the input they expect to receive. Feedback connections convey the predictions, while feedforward connection transmit the residual error between those predictions and actual input [26].

A connection can be drawn between these two models. If we consider negative feedback networks from an equivalent perspective, namely a generative one, it can be shown that the higher level of the network produces a *reconstruction* of the input via the feedback connections; while feedforward connections instead convey *residual error* between the top-down prediction and the bottom-up input [30], much like in the PC model. Symmetrically, *Predictive Coding* can be redefined as a form of *Biased Competition* [26, 27].

Algorithms that work for one model can be adapted to work for the other one. In particular, PC/BC networks trained using the DIM algorithm are referred to as PC/BC–DIM. Most notably, in this model the training of the weights of feedforward and feedback connections can be performed simultaneously and independently, thus providing a biologically sound computational theory explaining how reciprocal connections are learnt in actual cortical areas [5, 28].

[1] Divisive Input Modulation.

[2] Predictive Coding as Biased Competition.

1.2 The project

The main focus of the project has been improving the PC/BC–DIM algorithm with the ability to progressively grow a near-optimal network topology alongside the feedforward and feedback weights. In the process, I also implemented a constructive version of the basic DIM algorithm.

The project aim is to provide an enhanced version of the original algorithm which performs equally well but produces smaller networks that can be trained faster.

The structure of the problem is well suited for the design of a constructive variant. The network is fully connected between the error-detecting and the prediction layers, so there is no need to consider all the possible partial connections.

In the context of image decomposition, the pixel values act as the input which is fed to the error-detecting neurons, while the output of the nodes in the predictive layer represents the degree by which that component participates to the generation of the image. According to this interpretation, the weights associated to the connections targeting each prediction neuron are thus a "basic vector", or "elementary component" of the images to be reconstructed [29].

Furthermore, the number of neurons in the error layer is fixed to the number of input units. Thus the algorithm regulating the evolution of the network has to focus only on the output layer, by tuning its number of neurons when deemed necessary. That corresponds to determining the number of distinct components which are needed to decompose the images in the training set. As already stated, this value is in general not known *a priori*, and it has been the obvious target of the implemented algorithm.

Any type of constructive algorithm needs some kind of way to measure the error of the network to be able to decide whether and when adding a neuron to it. Conveniently, in the present case that quantity is embedded directly into the network itself, in the form of its error-detection layer. I performed experiments with different sets of criteria based on this measure, starting with those available in the literature describing the state-of-the-art, and modifying them and fine-tuning them to better fit the problem at hand.

Tested with some standard benchmarks, specifically the *bars* [28] and *squares* problem [30], the developed constructive algorithm has reported state-of-the-art results on the same level of accuracy of the original, while at the same time consistently building smaller networks.

In summary, the work conducted for this project represents a sensible improvement over the state-of-the-art DIM algorithm in term of training times and optimality of the size of the network.

1.3 Structure of the report

The main part of the report is structured as follows:

- Chapter 1 acts as an introduction to the project, its goals and the problem it solves;
- Chapter 2 provides a survey of the literature, with references to the relevant choices made during the project's undertaking;
- Chapter 3 describes the work conducted in full detail and evaluates the results against the original, state-of-the-art DIM algorithm.
- Chapter 4 concludes the body of the document by summarizing the main points of the work and pointing at possible future developments.

A list of the bibliographic references and an appendix containing the whole source code of the project terminate the report.

Chapter 2

Literature Review

2.1 Artificial Neural Networks

Artificial neural networks are a model designed to mimic the way in which the brain performs a task, usually simulated in software on a computer. Neural networks are parallel machines built upon simple, massively interconnected computing cells called *neurons* (figure 2.1). Every connection between nodes (*synapse*) has an associated *weight* [16].

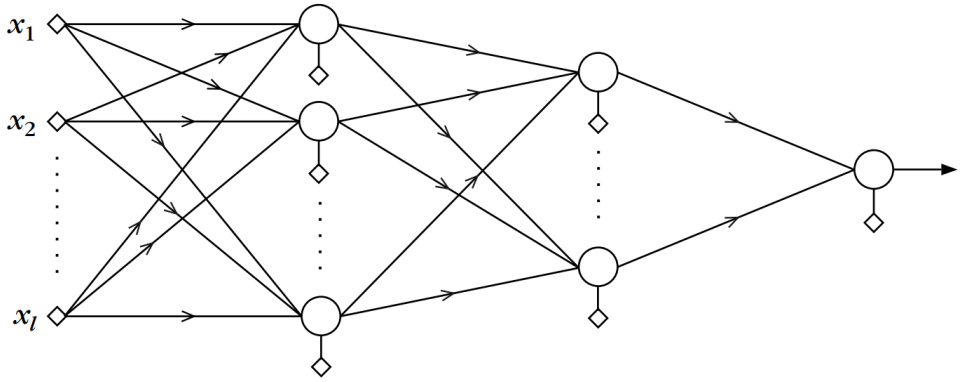

Figure 2.1: Architecture of an artificial neural network with two hidden layers and a single output neuron [31].

The way in which neural networks are usually programmed to perform a function is through a process of *learning*, which typically involves modifying the synaptic weights in such a way as to attain the desired objective [16]. As will be analysed in section 2.6, learning is not limited to that, as it is also possible for neural networks to modify their own topology, ability which is exploited by the current project.

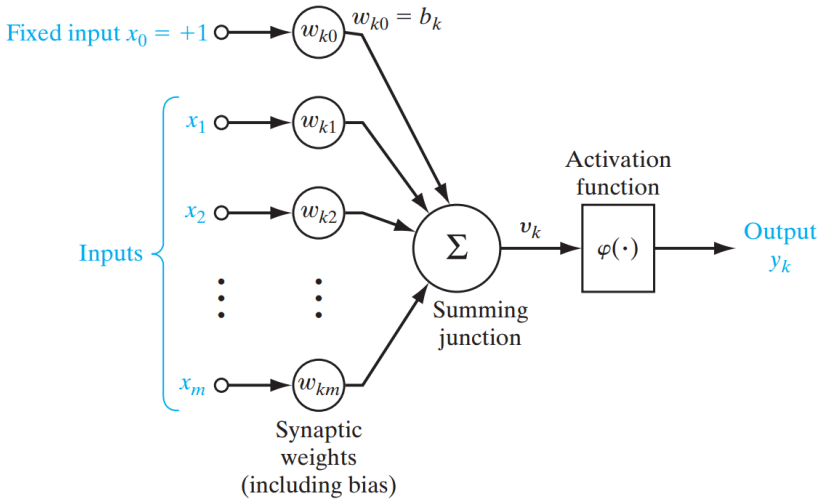

Figure 2.2: Nonlinear model of a neuron [16].

Figure 2.2 shows a representation of a generic model of an artificial neuron. In detail, its main components are:

1. A set of *synapses* or *connecting links*, characterized by a *weight* or *strength*. If a signal $\boldsymbol{x_j}$ is at the input of a synapse $\boldsymbol{j}$ connected to a neuron $\boldsymbol{k}$, the input signal gets multiplied by the synaptic weight $\boldsymbol{w_{kj}}$.

2. A *sum* function $\boldsymbol{\Sigma}$ which implements a *linear combination* of the inputs with their respective synaptic weights.

3. An *activation* or *transfer* function $\boldsymbol{\varphi}$ which limits the amplitude of the output of the neuron, usually to the closed interval $\boldsymbol{[-1, 1]}$. Nonlinear functions such as the *sigmoid* are common choices.

4. Optionally, the model can include a value $\boldsymbol{b_k}$ called *bias*, whose role is to increase or lower the net input of the activation function. This is usually implemented by having an additional, fixed input $\boldsymbol{x_0 = 1}$ multiplied by its respective weight $\boldsymbol{w_{k0}}$ [16].

Mathematically, the behaviour of a neuron can be described by the following equations:

$$\boldsymbol{v_k = \sum_{j=0}^{m} w_{kj} x_j} \tag{2.1}$$

$$\boldsymbol{y_k = \varphi(v_k)} \tag{2.2}$$

Nodes in a network are usually grouped into three classes: an *input layer*, whose neurons receive the information to be processed; an *output layer*, which yield the results of the processing; and neurons in between organized in zero or more *hidden layers* [18].

The representational power of neural networks is well defined theoretically: in fact, any continuous function can be implemented in a three-layer net, given sufficient number of hidden units, proper nonlinearities, and weights [17, 19, 21].

2.1.1 Network architectures

Feedforward networks

The simplest type of topology is the feedforward network, which allows signals to travel only one way, from input to output [18]; this layout is thus strictly acyclic. The signals resulting from the input layer are fed to the second layer, and so on till the final layer, whose output constitutes the overall response of the network to the source pattern [16].

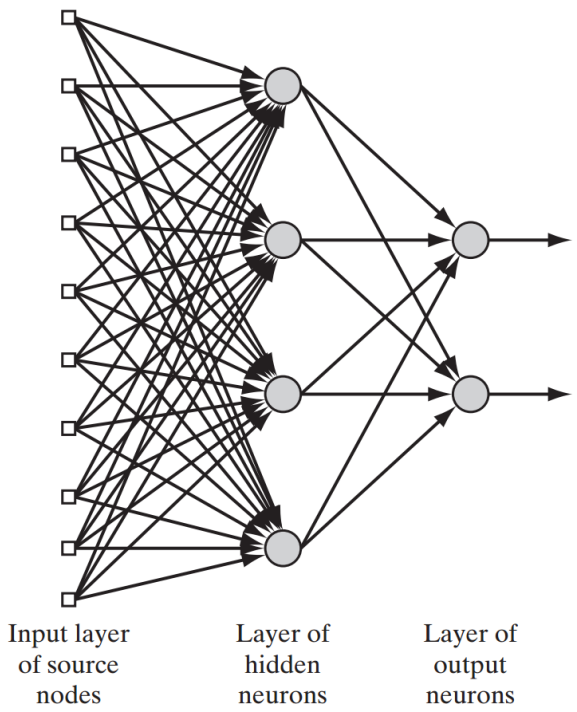

Figure 2.3: Fully connected feedforward neural network [16].

Typically (but not necessarily) the neurons in each layer are only connected to the ones into the immediately following layer. If every node in each layer is connected to every other node in the adjacent layer, the neural network is said to be *fully connected* (respectively *partially connected*) [16]. Figure 2.3 shows an example of such a network.

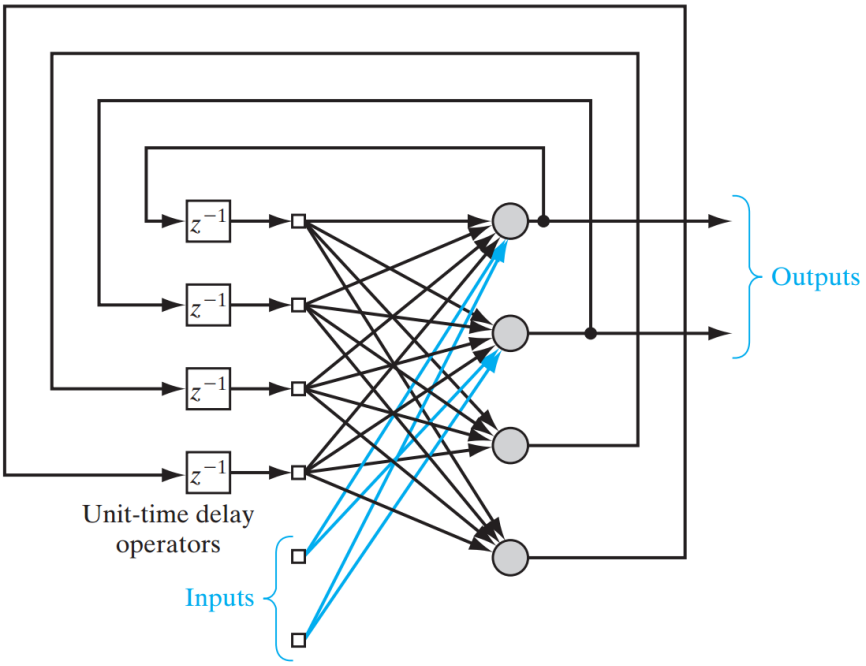

Figure 2.4: Recurrent neural network with hidden neurons [16].

Recurrent networks

Contrarily to feedforward networks, *recurrent* neural networks have at least one *feedback* loop. Feedback is present in a system when the output of an element influences (at least in part), the input applied to that same element [16]; connections among its neurons thus contribute to form a directed cycle in the topology [32]. The presence of these cycles has a deep impact on the learning capabilities of recurrent neural networks [16]. The topology employed by the DIM algorithm makes use of feedback loops (see section 2.2).

Recurrent networks are general computers, much more powerful than simple feedforward ones [24]; in fact, cycles account for the creation of an internal state which allows them to deal with temporal behaviour [32]. As a significant example, they have the ability to both create and recognize sequences of input patterns [9, 23].

Deep networks

A *deep* neural network is a neural network with multiple hidden layers. Even though, theoretically, *shallow* (as opposed as deep) networks have the ability to represent every kind of function, in practice recent findings indicate that *deep architectures* may be better suited to the task of representing some complicated functions (e.g. as those found in vision and language). Specifically, *"functions that can be compactly represented by a depth* $\boldsymbol{k}$ *ar-*

chitecture might require an exponential number of computational elements to be represented by a depth $\boldsymbol{k-1}$ *architecture"* [4].

The main reason behind that and the motivation for deep networks is that those complex behaviours are modelled by mathematical functions which are highly nonlinear in terms of the input signals. The statistical relationships governing that extreme variability may be to complex and not readily deductible at that level. Therefore each hidden layer of the deep network is assigned with the task of extracting *features* from the previous stage, in this way transforming the original input into increasingly more abstract representations [4].

The PC/BC model on which the project is based (described in section 2.4) in its general, hierarchical form can be considered a case of deep network.

2.1.2 Classes of training algorithms

The process of learning can be categorized in three distinct classes: *supervised*, *unsupervised*, and *reinforcement learning*.

In *supervised learning*, training is performed on the basis of a set of input-output examples provided from an external source of knowledge. Input vectors are fed to the network and the resulting output is compared with the expected one; then, through a process called *error-correction*, the weights of the network are modified in such a way as to minimize the error between the expected outcome and the actual output [16].

In *unsupervised learning*, no examples are provided and the training algorithm has to rely on a task-indipendent measure which quantifies the quality of the representation. The network is thus optimized with respect to that measure. Most unsupervised algorithms achieve training through *competition* [30]: in this case, an input layer of neurons sends activations to a competitive layer, whose neurons compete for the chance to respond to features in the data. Therefore nodes which represent features in the data are made more likely to activate, according to a certain rule [16]. Negative feedback networks (presented in section 2.2) employed by the DIM algorithm are an example of a model that provides competition.

In *reinforcement learning*, the network learns through interaction with the *environment*: the result of the output is evaluated and the network is updated to minimize an *index of performance*.

2.2 Negative Feedback Networks

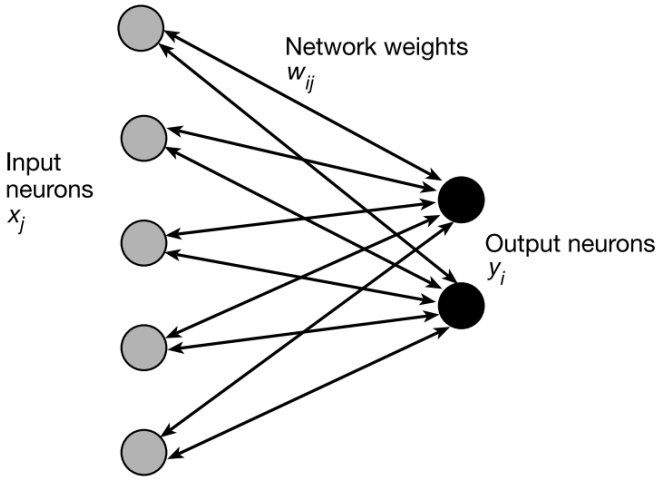

Figure 2.5: Negative feedback network where the activation is fed forward from inputs $\boldsymbol{x}$ through weights $\boldsymbol{W}$ to outputs $\boldsymbol{y}$. It is then fed back as subtractive *inhibition* through the same weights [7].

Negative feedback networks are a model for unsupervised learning in which competition is achieved, rather than by lateral *inhibition* (where nodes influence the output of other nodes), by inhibiting the inputs of the competing nodes. In this kind of networks, the activation of the output nodes is *fed back* to their inputs and subtracted, effectively implementing the competition [30], as in the example shown in figure 2.5.

2.2.1 Fyfe's Network

One good example of negative feedback network algorithm is the one described by Fyfe and his colleagues in [6, 7, 8, 12]. These networks are based on the initial assumption that both weights and outputs can not be negative; constraints are thus set so that, if that happens, they are instead set to zero. The motivations behind this are that (1) the kind of data that is presented to the network will always be positive, so it is reasonable to expect the activity in the network to be positive as well, and (2) synapses in the brain are not believed to be able to change from excitatory to inhibitory [7].

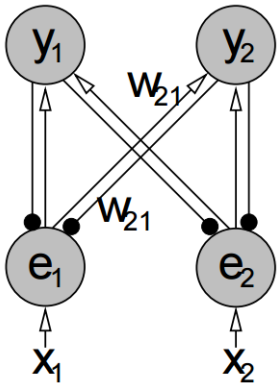

Figure 2.6: Two-node, two-input neural network with inhibitory feedback. Excitatory synapses are shown as arrows while inhibitory ones as filled circles [30]. This is the basic form of all the model on which the algorithms involved in the project are based.

If the network has $\boldsymbol{m}$ inputs and $\boldsymbol{n}$ competing nodes, then the equations regulating the activity of the networks are as following:

$$\boldsymbol{y = Wx} \tag{2.3}$$

$$\boldsymbol{e = x - W^T y} \tag{2.4}$$

$$\boldsymbol{W \leftarrow W + \beta y e^T} \tag{2.5}$$

Where $\boldsymbol{y = [y_1, ..., y_n]^T}$ represents the vector of the outputs, $\boldsymbol{x = [x_1, ..., x_m]^T}$ the vector of inputs, $\boldsymbol{W = [w_1, ..., w_n]^T}$ an $\boldsymbol{n}$ by $\boldsymbol{m}$ matrix of weights, with each row containing the weights received by a single node, and $\boldsymbol{e}$ is the vector of inhibited inputs[1].

The third equation describes the training rule, where $\boldsymbol{\beta}$ is a parameter known as *learning rate*. As it can be seen from all the equations, the inhibited inputs only influence the learning phase and not the output of the network directly, so this model does not fully implement competition which, in the application we are concerned of, results in the inability of the network to correctly represent the input, even when the weights have been correctly learned.

2.2.2 Harpur's Network

Harpur proposes a different algorithm based on the same model [13, 14, 15] which does provide competition which influences the network output

[1] As explained in the introduction (section 1.1), inhibited inputs can be equivalently interpreted as representing the *reconstruction error* of the network, without any changes to the mathematical model [30].

directly. The equations describing its behaviour are the following:

$$\boldsymbol{e} = \boldsymbol{x} - \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{y} \tag{2.6}$$

$$\boldsymbol{y} \leftarrow \boldsymbol{y} + \boldsymbol{\mu}\boldsymbol{W}\boldsymbol{e} \tag{2.7}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{\beta}\boldsymbol{y}\boldsymbol{e}^{\boldsymbol{T}} \tag{2.8}$$

As it can be seen, the learning rule is the same of the one used by Fyfe's algorithm. Note that weights are clipped to be in the range $[\mathbf{0}, \mathbf{1}]$. What is different from the previous case is how the values for $\boldsymbol{y}$ and $\boldsymbol{e}$ are calculated. First, and for each new input image, $\boldsymbol{y}$ is initialized to zero; then equations 2.6 and 2.7 are iterated until they converge to the final value. Negative values of $\boldsymbol{y}$ are clipped to zero at every iteration.

The parameter $\boldsymbol{\mu}$ has to be carefully chosen: big values will cause some elements of $\boldsymbol{y}$ to grow too large in a single step, causing the values in $\boldsymbol{e}$ responsible for the activation of those elements to decrease, and so on. Output values thus oscillate without reaching a steady-state. For this reason, $\boldsymbol{\mu}$ has to be small, which however causes $\boldsymbol{y}$ to be updated slowly, thus requiring many iterations to achieve convergence [30].

2.3 Non-negative Matrix Factorisation

Non-negative matrix factorisation is a method to find two factors $\boldsymbol{W}^{\boldsymbol{T}}$ and $\boldsymbol{Y}$ of a non-negative matrix $\boldsymbol{X}$ [30], such that:

$$\boldsymbol{X} \approx \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{Y} \tag{2.9}$$

$$\begin{aligned} \boldsymbol{W} &\not< \mathbf{0} \\ \boldsymbol{Y} &\not< \mathbf{0} \end{aligned} \tag{2.10}$$

Since image components are non-negative, theoretically this is a suitable method to find parts-based decomposition of images [10].

If $\boldsymbol{m}$ is the size of the images, $\boldsymbol{n}$ is the number of possible image components, and $\boldsymbol{p}$ is the number of images in the training set, then:

- $\boldsymbol{X} = [\boldsymbol{x_1}, ..., \boldsymbol{x_p}]$ is an $\boldsymbol{m} \times \boldsymbol{p}$ matrix, with each column containing the pixels of a training image.
- $\boldsymbol{W}^{\boldsymbol{T}}$ is an $\boldsymbol{m} \times \boldsymbol{n}$ matrix of weight values, with each column being a component (or basis vector) into which an image can be decomposed.
- $\boldsymbol{Y} = [\boldsymbol{y_1}, ..., \boldsymbol{y_p}]$ is an $\boldsymbol{n} \times \boldsymbol{p}$ matrix, with each column describing the activation of each component in the corrisponding training image.

An individual training image can be reconstructed as:

$$\boldsymbol{x_k} \approx \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{y_k} \tag{2.11}$$

Non-negative matrix factorisation shares mathematical similarity with negative feedback neural networks. In particular, NMF can be seen as a *divisory form* of feedback inhibition. However, the main difference is that NMF operates in batch mode, while negative feedback neural network is an on-line algorithm [30]. There exists a number of algorithms to approximate the solutions for that equation. Even so, all of them operate on all the training data at once.

2.3.1 Sequential version

In analogy to negative feedback networks, the term

$$\boldsymbol{E} = \boldsymbol{X} \oslash (\boldsymbol{W}^{\boldsymbol{T}} \boldsymbol{Y})$$

which measures the residual error in the reconstruction of $\boldsymbol{X}$, can be introduced. By performing the relevant substitutions and on the basis of some simplifying assumptions (detailed in [30]), the formulas can be rewritten and iterative rules for the vectors relative to a single training image (written in lowercase) can be derived:

$$\boldsymbol{e} = \boldsymbol{x} \oslash (\boldsymbol{\epsilon} + \boldsymbol{W}^{\boldsymbol{T}} \boldsymbol{y}) \tag{2.12}$$

$$\boldsymbol{y} \leftarrow (\boldsymbol{\epsilon} + \boldsymbol{y}) \otimes (\boldsymbol{W}\boldsymbol{e}) \oslash \tilde{\boldsymbol{w}} \tag{2.13}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} \otimes (\boldsymbol{1} + \boldsymbol{\beta}\boldsymbol{y}(\boldsymbol{e}^{\boldsymbol{T}} - \boldsymbol{1})) \tag{2.14}$$

Where $\tilde{\boldsymbol{w}}$ is a column vector each element of which represents the total synaptic weight received by one output node.

Comparing equation 2.12 with equation 2.6, the similarity between NMF and negative feedback networks is obvious: specifically, NMF implements a divisive, rather than subtractive, feedback.

In contrast to Fyfe's networks, sequential NMF provides competition between the nodes of the network. Moreover, the resulting algorithm is more stable than Harpur's algorithm [30].

2.4 PC/BC

2.4.1 Predictive coding

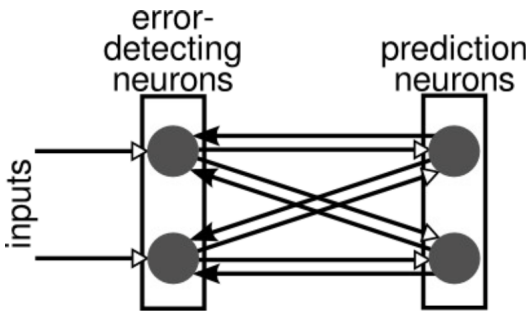

Figure 2.7: A graphical representation of a pair of adjacent populations in the PC model. [29].

Predictive coding is commonly used to model information processing in cerebral cortex. It is typically implemented as hierarchical neural networks with alternating populations of *error-detecting* and *prediction* neurons. Prediction neurons' activity produces a reconstruction of the inputs via feedback connections to the preceding error-detecting neurons; error-detecting neurons calculate the residual error between the reconstructed and the actual inputs [29].

By having the weights in a matrix $\boldsymbol{W}$, its rows, each of which corresponds to the weights targeting a single prediction neuron, represent the *elementary components* from which the images can be reconstructed. The strength of activation of the prediction neuron thus encodes the *degree of belief* that that pattern of inputs is present in the current image [29].

The matrix $\boldsymbol{W}$ can thus be thought as a *dictionary*, a model of the external world [28, 29].

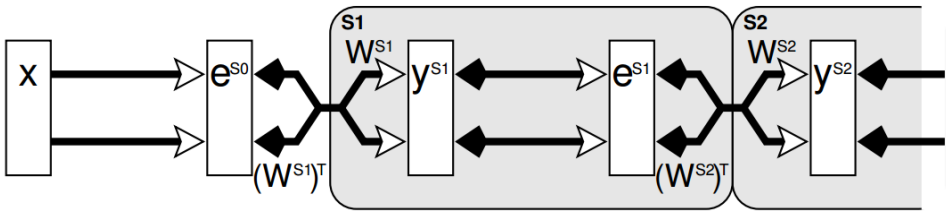

Figure 2.8: Multi-stage, hierarchical predictive coding model. Open arrows signify excitatory connections, filled arrows inhibitory ones [27].

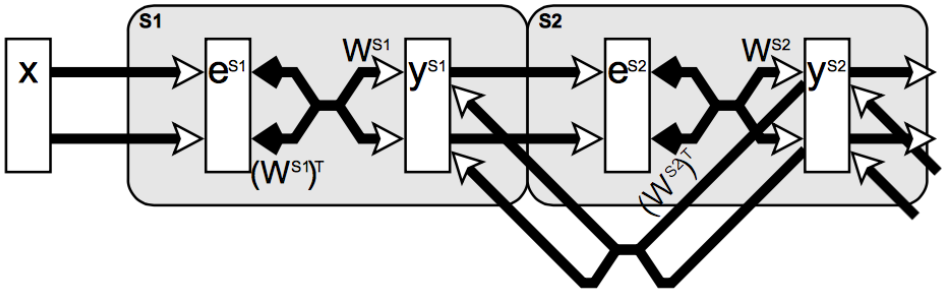

Figure 2.9: A graphical representation of the PC/BC model [26].

2.4.2 Predictive coding as biased competition

In the biased competition model, competition is influenced not only by bottom-up, sensory-driven, activity but also by top-down biases. These top-down influences increase the response of the neural activity generated for the attended stimulus and thus affects competition between neurons. Even though this model seems intuitively incompatible with the predictive coding one, they have been proved mathematically identical by Spratling [27].

The resulting model, called PC/BC (Predictive Coding as Biased Competition), shown in figure 2.9 is equivalent to PC but uses direct excitatory (rather than inhibitive) feedback from one population of prediction nodes to the preceding one [26].

With some modifications, the rearranged model can be implemented by means of a negative feedback neural network as the ones analysed previously [27].

2.5 Divisive Input Modulation

Notwithstanding the improvements of the sequential NMF algorithm compared to Fyfe and Harpur's networks, the former still suffers from a weakness that makes it poor at learning image components when overlapping between components is present in the training images [30].

The problem is illustrated by figure 2.10. Intuitively, one might expect that as the strengths of the weights increase, so would the response of the output node. In sequential NMF however, as the weights increase the output node inhibits the input it receives more strongly, and the response actually decreases. To fix this behaviour, Spratling [30] proposes an algorithm called *Divisive Input Modulation*, which is the one on which the current project is based on. The algorithm makes use of the following equations:

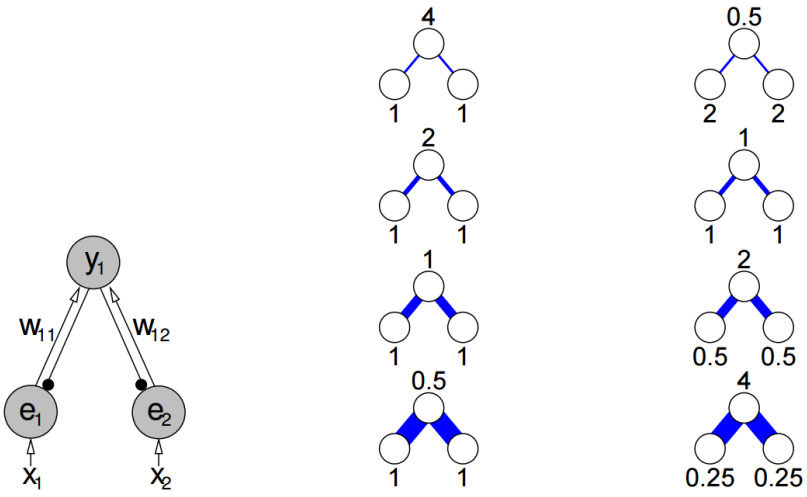

Figure 2.10: Weight update in (b) Non-negative Matrix Factorization and (c) Divisive Input Modulation. Each error-detecting node receives equal strength input from two image pixels (i.e., $\boldsymbol{x_1 = x_2 = 1}$) [30].

$$\boldsymbol{e = x \oslash (\epsilon + \hat{W}^T y)} \tag{2.15}$$

$$\boldsymbol{y \leftarrow (\epsilon + y) \otimes We} \tag{2.16}$$

$$\boldsymbol{W \leftarrow W \otimes (1 + \beta y(e^T - 1))} \tag{2.17}$$

Where $\hat{\boldsymbol{W}}$ is a matrix representing the same synaptic weight values as $\boldsymbol{W}$ but with rows normalised to have a maximum value of 1, and $\boldsymbol{\epsilon}$ is a small constant used to prevent division-by-zero errors. Following learning, weights are clipped at zero.

Divisive Input Modulation has shown results superior to existing methods in learning image components, especially in the presence of occlusion and overlapping [30].

2.5.1 PC/BC–DIM

Activation rules for a PC/BC model as the one in figure 2.9 implemented as a hierarchical neural network are as follows:

$$\boldsymbol{e^{Si} = G(y^{Si-1}) \oslash [\epsilon_2 + (V^{Si})^T y^{Si}]}$$

$$\boldsymbol{y^{Si} \leftarrow (\epsilon_1 + y^{Si}) \otimes W^{Si} e^{Si} \otimes [1 + \eta G((U^{Si+1}))^T y^{Si+1}]}$$

Where:

- $\boldsymbol{Si}$ indicates the stage $\boldsymbol{i}$ of the hierarchical network.

- $\boldsymbol{e^{Si}}$ is a $\boldsymbol{m} \times \boldsymbol{1}$ vector of error-detecting neuron activations.
- $\boldsymbol{y^{Si}}$ is a $\boldsymbol{n} \times \boldsymbol{1}$ vector of prediction neuron activations.
- $\boldsymbol{W^{Si}}$, $\boldsymbol{V^{Si}}$, $\boldsymbol{U^{Si}}$ are $\boldsymbol{n} \times \boldsymbol{m}$ matrces of synaptic weights.
- $\boldsymbol{\epsilon_1}$, $\boldsymbol{\epsilon_2}$, $\boldsymbol{\eta}$ are parameters.
- $\boldsymbol{G}$ is a function that clips values at 1.

In the first equation, the first term represent the actual input, while the second term represents the reconstructed input from the prediction neurons.

In the second equation, the first term represents the previous value of $\boldsymbol{y^{Si}}$, the second term is the feedforward input from error-detecting neurons to prediction ones, and the third term represents the modulation coming from the top-down predicitions of the next stage, if any.

In [28], Spratling proposes an algorithm to iteratively train the weights of the three matrixes independently by applying the DIM algorithm to the PC/BC model. This is extremely important because it makes the model biologically plausible. The weights update rules are as following:

$$\boldsymbol{W^{Si}} \leftarrow \boldsymbol{W^{Si}} \otimes \left\{ \boldsymbol{1} + \boldsymbol{\beta y^{Si}} \left[\left(\boldsymbol{e^{Si}} \right)^{\boldsymbol{T}} - \boldsymbol{1} \right] \right\} \tag{2.18}$$

$$\boldsymbol{V^{Si}} \leftarrow \boldsymbol{V^{Si}} \otimes \left\{ \boldsymbol{1} + \boldsymbol{\beta y^{Si}} \left[\left(\boldsymbol{e^{Si}} \right)^{\boldsymbol{T}} - \boldsymbol{1} \right] + \boldsymbol{\beta H}(\boldsymbol{y^{Si}} - \boldsymbol{1}) \overrightarrow{\boldsymbol{1}} \right\} \tag{2.19}$$

$$\boldsymbol{U^{Si}} \leftarrow \boldsymbol{U^{Si}} \otimes \left\{ \boldsymbol{1} + \boldsymbol{\beta y^{Si}} \left[\left(\boldsymbol{y^{Si-1}} \oslash \left[\boldsymbol{\epsilon_2} + (\boldsymbol{U^{Si}})^{\boldsymbol{T}} \boldsymbol{y^{Si}} \right] \right)^{\boldsymbol{T}} - \boldsymbol{1} \right] \right\} \tag{2.20}$$

Where:

- $\boldsymbol{H}$ is the Heaviside function.
- $\overrightarrow{\boldsymbol{1}}$ is a vector of length $\boldsymbol{m}$ in which each element is equal to 1.

2.6 Constructive Neural Networks

Constructive algorithms are used to adjust the structure of a neural network during the learning phase. Different approaches are possible: a purely constructive one adds layers, neurons and connections starting from a minimal architecture, whereas *pruning* starts from a complex structure and removes redundant parts; the two approaches can also be combined [22].

The advantages of using a constructive algorithm are, among the others:

- It is flexible: the whole space of possible neural network configurations can be explored.

- The initial configuration of the model is minimal and straightforward.
- Final configurations can be thought as estimating the real complexity of the learned problem.

The two main branches of constructive algorithms are Cascade Correlation (CC) and Dynamic Node Creation (DNC):

The CC algorithm produces neural networks with multiple hidden layers, with one neuron each that is connected to all other neurons previously added [25].

DNC add neurons in a hidden layer until the network achieves an approximation of the desired performances. Every time a neuron is added, all the weights are retrained. This is effective but at the expense of computational cost. [25]

OHL-FNN is an extension of DNC that freezes the neural network weights that have been previously trained, and that retrains only the weights affected by the insertion [20].

Chapter 3

Main Result

3.1 Analysis and Design

3.1.1 Motivation

There are various advantages in developing a constructive neural network algorithm over a standard one; some of them are obvious while some of them are more subtle.

First of all, as stated in the introductory section, differences in architecture and size can be all the difference needed in making a network useful for the problem. In this field, choices are often the result of trial and error and, as such, often result in designs which are larger than what would be actually needed. This has two main repercussions on the performance of the network.

On one hand, training a bigger network is computationally more expensive. On the other hand, models with too many degrees of freedom, while they can perform more complicated mappings, tend to overfit the training data and are not able to perform well over unseen instances. This ability, called *generalization*, implies lower order mappings [2].

An analogy with the problem of curve fitting using polynomials serves to illustrate the point: a polynomial with too few coefficients will be unable to capture the underlying function, while a polynomial with too many coefficients will fit the noise in the data and also result in a poor representation of the function. On the contrary, the optimal number of coefficients will result in the best representation of the function and also the best predictions for new data [20]. Constructive algorithms solve this problem by incorporating the evolution of the network structure into the learning algorithm.

Less obviously, but importantly, algorithms that prefer small solutions have the possibility to discover a minimal (or near-minimal) network which potentially suits the intrinsic complexity of the learning task [22].

3.1.2 Analysis of the current algorithm

The details of the Divisive Input Modulation algorithm and the theory on which is based have already been discussed in section 2.5; in this subsection we are only interested in focusing on the aspects that are most relevant in the design of its constructive version.

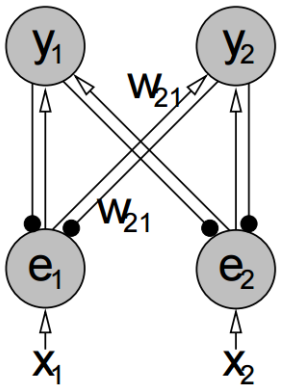

Figure 3.1: Basic structure of the neural network used by DIM [30].

Specifically, we should analyse the parameters involved, to discover which ones makes sense to modify at runtime and which ones offer information useful for the constructive process. The $\boldsymbol{m}$ pixels in the images, indicated by $\boldsymbol{x_1, ..., x_m}$, which are mapped one-to-one to the *error-detecting* neurons $(\boldsymbol{e_1, ..., e_m})$ are fixed parameters depending on the training set. The part of the network that can be grown or shrunk in size is the output or *predictive* layer $(\boldsymbol{y_1, ..., y_n})$ and, consequently, the number of rows in the matrix of weights $\boldsymbol{W}$; in short, the parameter $\boldsymbol{n}$.

A question which is important to answer is: what does tuning this parameter means in terms of image decomposition? As it is known from the literature [29], the weights received by a single output neuron are contained in the respective row in the matrix $\boldsymbol{W}$; each row corresponds to an elementary component discovered in the image. So tuning the parameter $\boldsymbol{n}$ equals to identify the exact number of components needed to describe the images in the training set.

Another important point is: what factor should drive the decision of adding neurons to the network? The way the algorithm is engineered, the values of the error-detecting neurons provide an estimation of how accurately the generative model is reconstructing the original input, and thus explaining its composition. Specifically, when a value within $\boldsymbol{e}$ is greater than $\mathbf{1}$, the corresponding element in the input is under-represented in the reconstruction, whereas if that value is less than $\mathbf{1}$, the opposite true. The interpretation of the image is totally exact if $\boldsymbol{e}$ is equal to $\mathbf{1}$ [30]. So it makes perfect sense to somehow consider by how far the values of $\boldsymbol{e}$ are from the

unity in order to approximate the accuracy the network has reached. From now on, when talking about the *error* in the context of the constructive algorithm, I will be referring to this measure.

In general, another variable to specify would be how the new neuron has to be connected to the network; however, in a *fully connected* network like the one we are dealing with, the solution is simple: connections have to be added to all the error-detecting nodes.

3.1.3 Classes of constructive algorithms

There are two angles from which tackling the problem of learning a network topology. One involves using a larger than needed network and training it until a solution is found; then, units are removed if they are not actively used. This is called *pruning*. The other, the regular *constructive* approach, starts with a small network and then grows additional units. The constructive approach has various advantages over pruning: first of all, it is really easy to specify an initial network, whereas for pruning algorithms is not obvious how big the initial network has to be; secondly, constructive algorithm search for small solutions first, so they are more computationally efficient [20]. For these reasons and because of the results in the initial experiments I performed, the constructive process was picked over pruning.

According to [11], two main approaches can be identified in the development of constructive neural network algorithms: they are *Cascade Correlation* (CC) and *Dynamic Node Creation* (DNC). Cascade Correlation and its derived algorithms are not suited for the problem at hand, as they create neural networks with multiple hidden layers, whereas DIM expects a single constructive layer to grow in the number of neurons.

For this reason, the class of algorithms known as DNC was considerate more appropriate for the task. The original model proposed by Ash [3] is quite simple: the algorithm adds neurons to the designated layer until the precision of the output reaches the desired level. The entire neural network is retrained every time a neuron is added.

Even though these algorithm were conceived for simple feed forward neural networks, they have been proven successful even when applied to recurrent networks [3].

3.1.4 Criteria for adding neurons

One of the main decisions to be made when designing a constructive algorithm is under what conditions should new neurons be added to the network.

At first, I experimented with a very simple criterion: simply add a neuron if the overall average error, measured as the absolute distance from $\mathbf{1}$ among all the elements of $\boldsymbol{e}$ and for one epoch, is higher than a certain threshold. Results were not very good and the network tended to grow bigger and

bigger, which initially led me to introduce a pruning phase to overcome the problem. As stated in section 3.1.3, pruning is an inherently inferior solution compared to construction, so I moved forward to explore different approaches.

An analysis of the shape of a typical curve describing the overall squared error in a neural network over time helps in deriving a better criterion. First of all, note that the error monotonically decreases over time. However, at some point the learning tends to reach a plateau and any further improvement will follow the rule of diminishing returns [3]. Just before that happens is the right moment when to change the topology of the network (i.e. adding a neuron) to prevent the plateau to occur.

Mathematically, if:

- $\boldsymbol{a_t}$ is the average squared error at time t across all the network,
- $\boldsymbol{t_0}$ is the time when the last node was added,
- $\boldsymbol{t}$ is the current time,
- $\boldsymbol{w}$ is the size of the time window over which the slope is evaluated,
- $\boldsymbol{\Delta_T}$ is the *trigger slope*, namely the value of the slope under which a neuron is added,

then a neuron is added to the network if the following conditions are met:

$$\frac{|\boldsymbol{a_t} - \boldsymbol{a_{t-w}}|}{\boldsymbol{a_{t_0}}} < \boldsymbol{\Delta_T} \tag{3.1}$$

$$\boldsymbol{t - w \geq t_0} \tag{3.2}$$

Equation 3.1 detects when the error curve gets into a plateau, according to the user-set parameter $\boldsymbol{\Delta_T}$, while equation 3.2 ensures that the first one always deals with the same topology.

This proved to be a much better measure than the original one, however the network would still keep growing indefinitely, trying to get a lower error even when the mapping is learned. Clearly, criteria for when to stop adding neurons to the network are needed. So, the algorithm will not add any nodes no matter what once the average error goes under a certain specified threshold:

$$\boldsymbol{a_t \leq C_a} \tag{3.3}$$

where $\boldsymbol{C_a}$ is the over-mentioned, user-defined threshold.

With the last modification, the algorithm performed fairly well, however there was still room for improvement. Specifically, as it is not possible to know *a priori* how big the network is going to be, the algorithm has to start

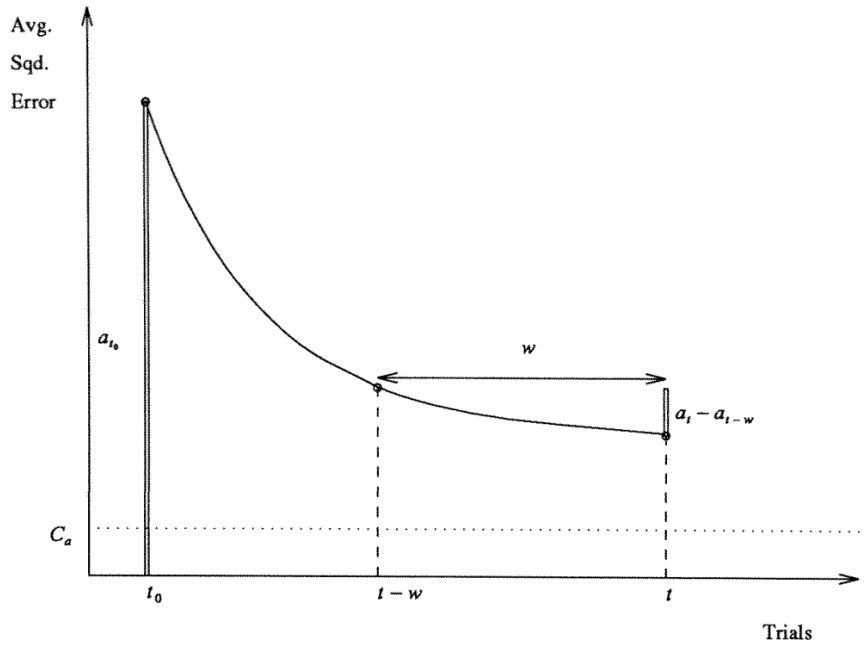

Figure 3.2: Graphic of the overall error as a function of training time [3].

with the smallest possible network, which is the one with a single predictive neuron. However, it can take various iterations to bring the network to an adequate size, especially if the number of components which make up the images is high.

Intuition suggests, and experiments have confirmed, that when the number of neurons in the network is largely inadequate to describe the images, then error-detecting neurons are going to yield values whose distance from the unity (and thus from correct representation) is very high. In this case, and thus if the error is over a certain threshold, the algorithm increases the number of neurons by a multiplicative factor until it goes under that threshold. It then starts behaving as described early, i.e. by adding neurons one at a time. This procedure is inspired by the (otherwise unrelated) TCP *congestion control* algorithm [1], in particular by its *exponential growth* phase. Mathematically speaking, the number of neurons is increased by multiplying it by a factor $\boldsymbol{M}$ while the following condition is true:

$$\boldsymbol{a_t \geq C_e} \tag{3.4}$$

Where $\boldsymbol{C_e}$ is a user-defined constant.

3.1.5 Division in epochs

To efficiently implement the concept of the *window*, explained in the previous section, I decided to modify the original algorithm, by dividing the training procedure in *epochs*. Throughout an epoch, each of which is constituted by a tunable number of cycles, the average of the error across the training set is kept updated. At the end of an epoch, and before starting the next one, the previously mentioned criteria are checked and the topology of the network modified accordingly, if necessary.

3.1.6 Weight training

Another outstanding question is how should weight training occur when the network grows in size? From the point of view of computational efficiency, and assuming that the units already present in the network are useful in part of the reconstruction [20], it would make sense to implement *weight freezing*, that is, keeping the weights targeting the already existing neurons fixed, and allowing the new ones to vary.

However, a case is made by Ash [3] that holding the existing network values constant only allows finding a solution in an affine subset of the weight space. The experiments I performed confirmed this thesis and the final version of my algorithm thus retrains whole the weights after each topology update. In fact, practical experience has shown that when the topology of the network changes, better results are obtained by just restarting the training from scratch.

The rationale behind it is that the growth phase is meant to just bring the network to an approximation of its correct size and it usually lasts very few epochs anyway.

3.1.7 Stopping criteria

An additional advantage that the designed algorithm has compared to the original one is the presence of a *stopping criterion* based on the error of the network. In fact, I found experimentally that when the error definitely goes under a certain threshold $\boldsymbol{C_s}$, the network most likely discovered all the image components. In that case, further training is deemed unnecessary and the algorithm can stop.

3.2 Implementation

In this section I will present some of the most important points related to the actual implementation. The crucial parts of the code will be provided and discussed. Sections which are not relevant for the understanding of the project will be summarized by comments in the code and the symbol "`...`"

will indicate missing code lines. For a full listing of the source code, refer to Appendix A.

3.2.1 Constructive DIM

The first algorithm I implemented and that I am going to analyse is a constructive version of the negative feedback network with divisive modulation described in section 2.5.

Figure 3.3: Parameters and important variables in the Constructive DIM algorithm.

```
1  %set network parameters
2  beta=0.05;       %learning rate
3  iterations=50;   %number of iterations to calculate y
4  epsilon=1e-10;
5
6  %define task
7  p=6;             %length of one side of input image
8  s=3;             %size of square image components
9  n=4;             %number of nodes
10 m=p*p;           %number of inputs
11 epochs=40;       %number of training epochs
12 cycs=1000;       %number of training cycles per epoch
13 patterns=1000;   %number of training patterns in training set
14
15 ...
16
17 %constructive parameters
18 t0 = 1;            %time of last added neuron
19 window = 2;        %window for slope calculation
20 tslope = 0.05;     %trigger slope
21 exptsh = 0.34;     %average error until exponential growth
22 cutavg = 0.21;     %average error to cut growing
23 stpavg = 0.1975;   %average error to stop
24 mult   = 1.5;      %multiplicative factor for growing
25
26 grow   = 1;        %boolean value to control growing
27 eavgs  = zeros(1, epochs);  %average errors per epoch
```

Figure 3.3 shows the initialisation part of the algorithm, mostly setting up the parameters needed for the actual learning phase. Most of this is straightforward: notice the parameter `n` (line 9) which regulates how big the initial network is going to be; `epochs` (line 11) and `cycs` (line 12) which are respectively the number of epochs the training is divided into, and the number of cycles per epoch (refer to section 3.1.5 for the rationale behind this).

Lines 18-24 initialize the parameters described in section 3.1.4, 3.1.6 and 3.1.7 which regulate the behaviour of the constructive algorithm. Specifically, `t0` keeps track of the last epoch when the network structure was modified and corresponds to the variable $\boldsymbol{t_0}$; `window` controls how far back to look into the epochs history to calculate the slope and corresponds to $\boldsymbol{w}$; `tslope` is the trigger slope that causes a neuron to be added ($\boldsymbol{\Delta_T}$); `exptsh` is $\boldsymbol{C_e}$, the threshold over which the size of the network gets multiplied by `mult` ($\boldsymbol{M}$) at each epoch; `cutavg` ($\boldsymbol{C_a}$) is the average error under which no neurons are added to the network anymore; under `stpavg` ($\boldsymbol{C_s}$) the algorithm stops running.

Line 26 defines the boolean variable `grow`, which regulates whether neuron should be added to the network or if the constructive phase is finished. Line 27 sets up a vector (`eavgs`) which will hold the history of the average errors per epoch, so that the slope can be calculated at runtime.

Following the parameters initialisation, the matrix $\boldsymbol{W}$ and the dataset are randomly initialized and then the main algorithm runs.

Figure 3.4 shows the main loops that compose the algorithm. The external loop (lines 1-52) iterates over the epochs. For each epoch, an average error throughout that epoch is calculated (`eavg`); this value gets initialised to zero at line 3.

The internal loop (lines 5-17) iterates over the cycles for each epoch; here, $\boldsymbol{y}$ and $\boldsymbol{e}$ are calculated for each training image and weights are updated according to the rules described in section 2.5. Lines 10-13 maintain a running average of the error throughout the epoch, calculated as a distance from the unity of the non-zero error-detection activations.

For each epoch, the calculated average error gets saved into the history vector (line 20). Then the criteria are checked: first (lines 23-26) if learning has converged and the algorithm can be stopped; then if no more neurons have to be added anymore (lines 28-31). Then, if network growing is still enabled (line 34), multiplication of the number of neurons gets performed if the error average for the current epoch is over `exptsh` ($\boldsymbol{C_e}$), or alternatively, if equations 3.1 (line 44) and 3.2 (line 43) are satisfied (that is, the error is not decreasing at the desired rate), a single neuron is added to the network (line 46). In either cases, the matrix of weights gets reinitialised (lines 39 and 47).

Figure 3.4: The main body of the Constructive DIM algorithm.

```
for t=1:epochs
  ...
  eavg = 0;

  for k=1:cycs
    ...
    %iterate to calculate node activations
    ...

    %update average error
    if ~isempty(e(e>0)),
      eavg = (eavg*(k-1) + mean(abs(e(e>0) - 1))) / k;
    end;

    %update weights
    ...
  end

  ...
  eavgs(t) = eavg;  %save average error into vector
  ...

  %check stop condition
  if eavgs(t) <= stpavg,
      break;
  end;

  %check stop growing condition
  if eavgs(t) <= cutavg,
      grow = 0;
  end;

  %check growing condition
  if grow,
    %exponential growth
    if eavg >= exptsh,
      t0 = t;
      n = round(n * mult);
      W=(1/16)+(1/64).*randn(n,m);
      W(W<0)=0;

    %gradual growing
    elseif t - window >= t0,
      if (abs(eavgs(t) - eavgs(t - window)) / eavgs(t0)) <
    tslope,
        t0 = t;
        n = n + 1;
        W=(1/16)+(1/64).*randn(n,m);
        W(W<0) = 0;
      end;
    end;
  end;
end
```

3.2.2 Constructive PC/DC–DIM

The main implementation difference that went into developing a constructive version of the PC/DC–DIM algorithm, compared to the DIM one, is that three matrices of weights get trained simultaneously. So the topology of all three must be updated. Figure 3.5 shows the relevant part.

Figure 3.5: How the topology gets modified in Constructive PC/BC–DIM.

```
1  ...
2  %check growing condition
3  if grow,
4    %exponential growth
5    if eavg >= exptsh,
6      t0 = t;
7      n = round(n * mult);
8      [W,V,U]=weight_initialisation_random(n,m);
9      y = [];
10
11   %gradual growing
12   elseif t - window >= t0,
13     if (abs(eavgs(t) - eavgs(t-window)) / eavgs(t0)) < tslope,
14       t0 = t;
15       n = n + 1;
16       [W,V,U]=weight_initialisation_random(n,m);
17       y = [];
18     end;
19   end;
20 end;
21 ...
```

Lines 9 and 19 augment the $\boldsymbol{y}$ variable (the output of the network) with an extra element. This is needed because PC/BC–DIM supports *continuous learning*, that is, the network is not reset after every image is presented.

The rest of the code is functionally equivalent to the Constructive DIM algorithm described in the previous section.

3.3 Results

In order to assess the quality of the results, the algorithm has been tested against two sets of benchmarks: the *squares* problem and the *bars* problem.

Both problems test the ability of the algorithm to learn the component parts from which a set of artificial training images are composed [30, 28]. The tests focus on three aspects: accuracy of the recognition, size of the network and training time.

3.3.1 Divisive Input Modulation

Squares problem

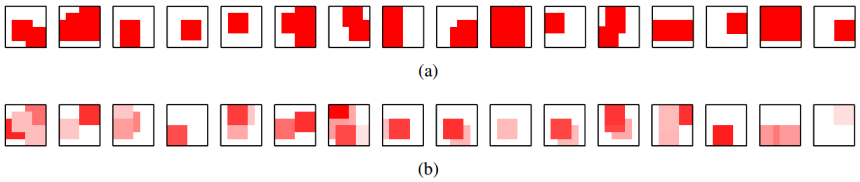

Figure 3.6: Example of 6-by-6 pixel images generated from 3-by-3 pixel square components [30].

The squares problem is a more challenging variation of the bars problem with more significant overlapping between components. Each image in the dataset is generated by selecting, at random, elements from a fixed set of elementary image components, which are all s-by-s pixel squares. Each component gets assigned a probability regulating the frequency of appearance in the dataset; for each image, every selected component gets randomly assigned a *contrast* and a (unique) *depth*. Each pixel in the image thus gets a greyscale value corresponding to the contrast of the foremost square in that location, if any.

The algorithms try to identify the components from which the dataset has been composed. Figure 3.7 shows the Constructive DIM in the middle of recognizing 3-by-3 pixel components in 6-by-6 pixel images. In this case a bare minimum of 16 nodes is required to learn the given problem and the network is still growing towards that size.

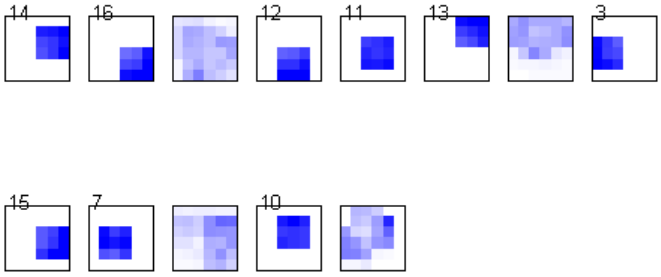

Figure 3.7: Constructive DIM algorithm while in the process of learning 3-by-3 components in 6-by-6 pixel images. Each cell shows a visual representation of what a neuron responds to. Numbers over images show the corresponding component that the neuron has learned, if any.

Evaluation of the results

Algorithm	*Accuracy*	*Cycles*	*Training time*	*Nodes*
DIM (32 nodes)	98.01%	20000 (fixed)	11.50s (average)	32 (fixed)
Constructive DIM	**97.94%**	**21720 (average)**	**8.23s (average)**	**22.66 (average)**
DIM (time-constrained)	96.19%	19911 (average)	8.23s (constrained)	$\lceil 22.66 \rceil = 23$ (fixed)
DIM (16 nodes)	89.31%	20000 (fixed)	6.15s (average)	16 (fixed)

Table 3.1: Performances in squares problem as measured in a dataset of 6-by-6 pixel images with 3-by-3 pixel components. *Accuracy* is defined as the percentages of components correctly identified; *cycles* and *training time* refer respectively to cycles and time spent to solve an instance of the problem; *nodes* refers to the number of neurons in the final network.

To assess the performance of the new algorithm against the original one, I first ran the constructive algorithm over a set of 1000 instances of the problem and collected the relevant data. I then performed two sets of tests on the original algorithm. The obtained results are reported in table 3.1.

On the first one, I ran the algorithm with the default parameters (20000 cycles) and with the maximum number of nodes that was ever reached during all the trials performed by the constructive algorithm (that is, 32). What I wanted to verify was how much better could DIM perform in the ideal case where the number of needed neurons is known. The standard algorithm was approximately 40% slower and was more accurate by merely 0.07 percentage points.

The second test was performed over a slightly modified version of the original algorithm that is forced to terminate in a given number of seconds. I set the number of neurons to the average that was needed by the constructive process and the maximum time to the average time needed by the constructive algorithm. I wanted to examine exactly how inferior the original algorithm was when using the most common learned topology and in the same amount of time that the constructive algorithm usually needs. Under these conditions, the original algorithm was less accurate than the developed one by 1.75 percentage points.

Furthermore, the results shows that, in order to converge smoothly to the correct solution, the DIM algorithm seems to require a slightly larger number of neurons compared to theoretical minimum, which is of course the number of components from which the images are generated. In the case of the tests performed in table 3.1 for example, the number of 3-by-3 pixel components is 16; however, the smallest network evolved by the constructive algorithm contained 21 predictive neurons. Tests performed with the original algorithm and 16 neurons (also listed in table 3.1), showed poor results, confirming this thesis.

3.3.2 PC/BC–DIM

Bars problem

The bars problem is a problem which is similar (but easier) to the squares one. The components from which the images are generated are lines (*bars*) of pixels. A common variation of the standard bars problem uses 5-by-5 pixel images in which the 10 possible (one pixel wide) horizontal and vertical bars are independently selected to be present with a probability of $\frac{1}{5}$.

To increase the difficulty of the task, noise can be added to the training images by changing the value of each pixel in the training set with a certain probability [28].

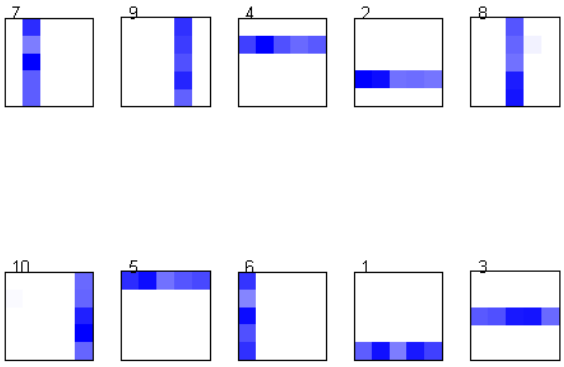

Figure 3.8: Bars component in 5-by-5 pixel images learned by the algorithm. Each cell shows a visual representation of what a neuron responds to. Numbers over images show the corresponding component that the neuron has learned, if any.

Evaluation of the results

Algorithm	*Accuracy*	*Cycles*	*Training time*	*Nodes*
PC/BC-DIM (24 nodes)	98.33%	20000 (fixed)	26.83s (average)	24 (fixed)
Constructive PC/BC-DIM	**96.15%**	**20000 (average)**	**16.47s (average)**	**18.35 (average)**
PC/BC-DIM (10 nodes)	90.33%	20000 (fixed)	14.24s (average)	10 (fixed)

Table 3.2: Performances in bars problem as measured in a dataset of 5-by-5 pixel images with 0.1 noise. *Accuracy* is defined as the percentages of components correctly identified; *cycles* and *training time* refer respectively to cycles and time spent to solve an instance of the problem; *nodes* refers to the number of neurons in the final network.

Performances of the constructive PC/BC–DIM algorithm have been performed following the same procedure of the constructive DIM one. Obtained results are qualitatively similar, even though they are quantitatively different enough to deserve a separated analysis.

As shown in table 3.2, the developed algorithm was 5.82 percentage points more accurate than the original one when using the theoretically minimal network (which is composed by 10 neurons in the given problem), and 2.18 percentage points less accurate than the original one with size 24 (which was the maximum number of neurons evolved in an instance of the problem by the constructive algorithm).

This is a more sensible difference compared to the DIM results. The reasons for this are to be found in the fact that the activation of the error-detecting neurons seem to be less stable in PC/BC–DIM, possibly because of the three matrix of weights that are trained independently and are sometimes slightly inconsistent with each other. Because of that, decisions on the growth of the network are somewhat less informed than the one performed by the constructive DIM algorithm. For the same reason, it was impossible to set an error under which stopping the algorithm, as that often leads to false positives.

Chapter 4

Conclusions

The project I developed and the results I obtained prove that there is room for improvement in the original, state-of-the-art Divisive Input Modulation algorithm.

I showed how it is possible, by only using knowledge internal to the network (that is, the output of its error-detecting neurons) and in a purely unsupervised fashion, to derive indications about the optimal size of the network and employ them to evolve its architecture while learning is taking place.

Importantly, the developed algorithm eliminates the need to perform *trial-and-error* to discover the right size of the network and can thus be applied to tasks where the number of image components is unknown beforehand. Furthermore, the evolved networks often come close to the theoretically optimal size, with the immediate consequence of being faster, both to train and to use.

These advantages in flexibility and speed were obtained at very little costs in term of accuracy; the constructive algorithm performed almost identically to the original one even when the latter was set up to train networks that were double the theoretically minimal size.

Some further research paths could be pursued in the future. First, a *pruning* phase based on removing the least activated neurons in the predictive layer could be added as a second step to the algorithm, further reducing the size of the obtained networks. Second, criteria that are more tolerant to the instability of the error in the PC/BC–DIM algorithm may be experimented. And finally, the algorithm can be adapted to the multi-stage, hierarchical PC/BC model to evolve deep architectures that perform additional feature extraction.

Bibliography

[1] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[2] Ethem Alpaydin. Gal: Networks that grow when they learn and shrink when they forget. *International journal of pattern recognition and Artificial Intelligence*, 8(01):391–414, 1994.

[3] Timur Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1(4):365–375, 1989.

[4] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[5] Edward M Callaway. Local circuits in primary visual cortex of the macaque monkey. *Annual review of neuroscience*, 21(1):47–74, 1998.

[6] D Charles, C Fyfe, and DW Pearson. Discovering independent sources with an adapted pca neural network. In *Proceedings of the 2nd International ICSC Symposium on Soft Computing (SOCO97)*. NAISO Academic Press, Nımes, France, 1997.

[7] Darryl Charles and Colin Fyfe. Modelling multiple-cause structure using rectification constraints. *Network: Computation in Neural Systems*, 9(2):167–182, 1998.

[8] Darryl Charles, Colin Fyfe, Donald McDonald, and Jos Koetsier. Unsupervised neural networks for the identification of minimum overcomplete basis in visual data. *Neurocomputing*, 47(1):119–143, 2002.

[9] Axel Cleeremans. *Mechanisms of implicit learning: Connectionist models of sequence processing*. MIT press, 1993.

[10] Tao Feng, Stan Z Li, Heung-Yeung Shum, and HongJiang Zhang. Local non-negative matrix factorization as a visual representation. In *Development and Learning, 2002. Proceedings. The 2nd International Conference on*, pages 178–183. IEEE, 2002.

[11] Bruno JT Fernandes, George DC Cavalcanti, and Tsang I Ren. Constructive autoassociative neural network for facial recognition. *PloS one*, 9(12):e115967, 2014.

[12] Colin Fyfe. A neural network for pca and beyond. *Neural Processing Letters*, 6(1-2):33–41, 1997.

[13] George F Harpur and Richard W Prager. A fast method for activating competitive self-organising neural networks. In *Proceedings of the International Symposium on Artificial Neural Networks*, pages 412–8, 1994.

[14] George F Harpur and Richard W Prager. Development of low entropy coding in a recurrent network. *Network: computation in neural systems*, 7(2):277–284, 1996.

[15] George Francis Harpur. *Low entropy coding with unsupervised neural networks*. PhD thesis, Citeseer, 1997.

[16] Simon S Haykin, Simon S Haykin, Simon S Haykin, and Simon S Haykin. *Neural networks and learning machines*, volume 3. Pearson Education Upper Saddle River, 2009.

[17] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[18] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, pages 3–24, 2007.

[19] Vera Kuurkova. Kolmogorov's theorem and multilayer neural networks. *Neural networks*, 5(3):501–506, 1992.

[20] Tin-Yau Kwok and Dit-Yan Yeung. Objective functions for training new hidden units in constructive neural networks. *Neural Networks, IEEE Transactions on*, 8(5):1131–1148, 1997.

[21] Hak-Keung Lam. Pattern recognition lecture slides, 2015.

[22] Rajesh Parekh, Jihoon Yang, and Vasant Honavar. Constructive neural-network learning algorithms for pattern classification. *Neural Networks, IEEE Transactions on*, 11(2):436–451, 2000.

[23] Barak Pearlmutter. Dynamic recurrent neural networks. 1990.

[24] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[25] Sudhir Kumar Sharma and Pravin Chandra. Constructive neural networks: a review. *International journal of engineering science and technology*, 2(12):7847–7855, 2010.

[26] Michael W Spratling. Predictive coding as a model of biased competition in visual attention. *Vision research*, 48(12):1391–1408, 2008.

[27] Michael W Spratling. Reconciling predictive coding and biased competition models of cortical function. *Frontiers in Computational Neuroscience*, 2, 2008.

[28] Michael W Spratling. Unsupervised learning of generative and discriminative weights encoding elementary image components in a predictive coding model of cortical function. *Neural Computation*, 24(1):60–103, 2012.

[29] Michael W Spratling. Predictive coding. In *Encyclopedia of Computational Neuroscience*. 2014.

[30] Michael W Spratling, Kris De Meyer, and R Kompass. Unsupervised learning of overlapping image components using divisive input modulation. *Computational intelligence and neuroscience*, 2009, 2009.

[31] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Fourth Edition*. Academic Press, 4th edition, 2008.

[32] Dong Yu and Li Deng. Recurrent neural networks and related models. In *Automatic Speech Recognition*, pages 237–266. Springer, 2015.

Appendix A

Source Code

A.1 Constructive DIM

This code is based on the Divisive Input Modulation algorithm coded by Spratling [30]. The original version (`dim_squares.m`) can be downloaded from his website:

`http://www.inf.kcl.ac.uk/staff/mike/code.html`

dim_squares.m

```
function [n,recognized,cycles] = dim\_squares()

%set network parameters
beta=0.05;                  %learning rate
iterations=50;              %number of iterations to calculate y
    (for each input)
epsilon=1e-10;

%define task
p=6;                       %length of one side of input image
s=3;                       %size of square image components
n=4;                       %number of nodes
m=p*p;                     %number of inputs
epochs=40;                 %number of training epochs
cycs=1000;                 %number of training cycles per epoch
patterns=1000;             %number of training patterns in
    training set
numsquares=(p-s+1).^2;
probs=0.1*ones(1,numsquares);
mincontrast=1;
%probs=0.02+0.18*rand(1,numsquares);
%mincontrast=0.1;

%constructive parameters
t0 = 1;                          %time of last added neuron
window = 2;                      %window for slope calculation
tslope = 0.05;                   %trigger slope
```

```
exptsh = 0.34;                %average error until exponential
    growth
cutavg = 0.21;                %average error to cut growing
stpavg = 0.1975;              %average error to stop
mult   = 1.5;                 %multiplicative factor for growing

grow   = 1;                   %boolean value to control growing
eavgs  = zeros(1, epochs);    %average errors per epoch

%generate a fixed pattern set
clear data
for k=1:patterns
  data(:,k)=squares\_pattern\_randprob(p,s,probs,mincontrast);
end

%define initial weights
W=(1/16)+(1/64).*randn(n,m);%Gassian distributed weights with
    given
               %mean and standard deviation
W(W<0)=0;

%learn receptive fields
for t=1:epochs
  fprintf(1, 'Epoch %i, ', t);
  eavg = 0;

  for k=1:cycs
    patternNum=fix(rand*patterns)+1; %random order
    x=data(:,patternNum);

    What=W./(epsilon+(max(W')'*ones(1,m)));%weights into nodes
    normalised by
                                          %maximum value

    %iterate to calculate node activations
    y=zeros(n,1);
    for i=1:iterations
      e=x./(epsilon+(What'*y));
      y=(epsilon+y).*(W*e);
    end

    %update average error
    if ~isempty(e(e>0)),
      eavg = (eavg*(k-1) + mean(abs(e(e>0) - 1))) / k;
    end;

    %update weights
    W=W.*( 1 + beta.*( y*(e'-1) ));
    W(W<0)=0;
  end

  %show weights
  fprintf(1,'nodes: %i, error: %f, ',n,eavg);
  eavgs(t) = eavg;  %save average error into vector
```

```
77    recognized = squares\_plot(s,W);
78
79    %check stop condition
80    if eavgs(t) <= stpavg,
81        break;
82    end;
83
84    %check stop growing condition
85    if eavgs(t) <= cutavg,
86        grow = 0;
87    end;
88
89    %check growing condition
90    if grow,
91      %exponential growth
92      if eavg >= exptsh,
93        t0 = t;
94        n = round(n * mult);
95        W=(1/16)+(1/64).*randn(n,m);
96        W(W<0)=0;
97
98      %gradual growing
99      elseif t - window >= t0,
100       if (abs(eavgs(t) - eavgs(t - window)) / eavgs(t0)) <
      tslope,
101         t0 = t;
102         n = n + 1;
103         W=(1/16)+(1/64).*randn(n,m);
104         W(W<0) = 0;
105       end;
106     end;
107   end;
108 end
109
110 s=sum(W'), disp(num2str([max(s),min(s),max(max(W)),min(min(W))
      ]))
111 disp('');
112 cycles = t*cycs;
113
114
115 function [x,patterns,input\_set\_components]=squares\_pattern\
      _randprob(m,s,prob,mincontrast)
116 %function [x,patterns,input\_set\_components]=squares\_pattern\
      _randprob(m,s,prob,mincontrast)
117 %
118 %create a mxm pixel image in which overlapping sxs squares are
      randomly
119 %active with probability 'prob', where prob is a vector
      defining the
120 %independent probability for each separate componenet.
121 %The contast of each component is assigned randomly (between
      mincontrast and
122 %1) in each generated pattern.
123
```

```
nsquares=(m-s+1);

%choose one square to be present, so that each pattern will
    contain at least one
%square. Need to make choice based on probability of each
    component being
%present.
c=rand*sum(prob);
included=min(find(c<cumsum(prob)));
depthorder=randperm(nsquares^2);%randomly assign a depth to
    each possible
                %component to decide which contrast goes on top
%randomly assign a contrast between mincontrast and 1 to each
    possible component
contrast=mincontrast+(1-mincontrast).*rand(1,nsquares^2);

%add patterns to input
npattern=0;
patterns=[];
x=zeros(m,m);
depth=zeros(m,m);
for c=1:nsquares
  for r=1:nsquares
  npattern=npattern+1;
  if (rand<prob(npattern) | npattern==included) & contrast(
    npattern)>0
    %decide at which pixels current component is infront
    depth(r:r+s-1,c:c+s-1)=max(depth(r:r+s-1,c:c+s-1), ...
                  depthorder(npattern));
    %fill in those pixels with contrast of current component
    for j=r:r+s-1
    for i=c:c+s-1
      if depth(j,i)==depthorder(npattern)
      x(j,i)=contrast(npattern);
      end
    end
    end

    %record fact that this component has been selected
    patterns=[patterns,npattern];
  end
  end
end


%remove any patterns from the list of included patterns that
    are entirely
%occluded by other patterns
npattern=0;
occludedpatterns=[];
for c=1:nsquares
  for r=1:nsquares
  npattern=npattern+1;
  if ismember(npattern,patterns)
```

```
172     %see if this pattern is ontop at any pixel
173     if ~ismember(depthorder(npattern),depth(r:r+s-1,c:c+s-1));
174     occludedpatterns=[occludedpatterns,npattern];
175     end
176   end
177   end
178 end
179
180 patterns=setdiff(patterns,occludedpatterns);
181
182 x=x(:);
183 if sum(x)==0
184   disp('missing');
185   [x,patterns,input\_set\_components]=squares\_pattern(m,s,prob
      ,noise);
186 end
187
188 input\_set\_components=zeros(1,nsquares^2);
189 input\_set\_components(patterns)=1;
190
191
192
193
194 function [nrepanycomplete]=squares\_plot(sqsize,weights)
195 %function [nrepanycomplete]=squares\_plot(sqsize,weights)
196
197 if nargin<2
198   weights=load('weights\_1basal.dat');
199 end
200 [n,m]=size(weights);
201 p=sqrt(m);
202 scale=max(max(weights))*0.85;
203
204 plot\_per\_row=min(n,8);
205 num\_rows=ceil(n/plot\_per\_row);
206 clf
207
208 sqsize=sort(sqsize);
209 nsqsizes=length(sqsize);
210 totalsquares=0;
211 for k=1:nsqsizes
212   s=sqsize(k);
213   totalsquares=totalsquares+(p-s+1)^2;
214 end
215 representedcomplete=zeros(1,totalsquares);
216
217 for j=1:n
218
219   w=reshape(weights(j,:),p,p);
220   subplot(num\_rows,plot\_per\_row,j),hinton\_plot(w,scale
      ,3,1,1);
221
222   %determine degree of match between weights and all possible
      input patterns
```

```
  npattern=0;
  lpref=[];
  for k=1:nsqsizes
  s=sqsize(k);
  nsquares=(p-s+1);
  for c=1:nsquares
    for r=1:nsquares
    npattern=npattern+1;
    wOut=w; wOut(r:r+s-1,c:c+s-1)=0;
    wIn=w(r:r+s-1,c:c+s-1);

    if sum(sum(wIn))>3*sum(sum(max(0,wOut))) & ...
        min(min(wIn))>max(max(wOut)) & ...
        min(min(wIn))>mean(mean(max(0,w)))
      lpref=[lpref,npattern];
    end
    end
  end
  end
  if length(lpref)>1
  lpref
  end

  representedcomplete(lpref)=representedcomplete(lpref)+1;

  if representedcomplete(lpref)>1,
  text(1,0.25,['(',int2str(lpref),')']);
  else
  text(1,0.25,int2str(lpref));
  end
end
nrepanycomplete=length(find(representedcomplete>0));
disp(['network represents ', int2str(nrepanycomplete), '
    patterns ']);
norep=find(representedcomplete==0);
if length(norep>1) disp(['FAILED to represent pattern = ',
    int2str(norep)]); end



function hinton\_plot(W, scale, colour, type, equal)
%function hinton\_plot(W, scale, colour, type, equal)
% type 0 = variable size boxes (size relates to strength)
% type 1 = image (color intensity relates to strength)
% type 2 = equal size squares (color intensity relates to
    strength)
% type 3 = same as type 0 but with outerboarder showing maximum
     size of box

W(find(W<0))=0;

if (type==1)
  %draw as an image: strength indicated by pixel darkness

```

```
273   %W is true data value (greater than 0) and is scaled to be
        between 0 and 255
274   imagesc(uint8(round((W./scale)*255)),[0,255]);%,'CDataMapping
        ','scaled'),
275   colormap(gray)
276   map=colormap;
277   map=flipud(map);
278   map(1:64,colour)=map(1:64,colour)*0.0+1;
279   colormap(map)
280   axis on
281   if(equal==1), axis equal, end
282   if(equal==1), axis tight, end
283
284 elseif (type==0 | type==3)
285   %draw as squares: strength indicated by size of square
286
287   colstr=['r','g','b'];
288   if(equal==0)
289   %calc aspect ratio - if not going to set axis equal
290   plot(size(W,2)+0.5,size(W,1)-0.5,'bx');
291   hold on
292   plot(0.5,-0.5,'bx');
293   axis equal
294   a=axis;
295   aspectX=size(W,2)/abs(a(2)-a(1));
296   aspectY=size(W,1)/abs(a(4)-a(3));
297   aspectXX=aspectX./max(aspectX,aspectY);
298   aspectYY=aspectY./max(aspectX,aspectY);
299   hold off
300   else
301   aspectXX=1;
302   aspectYY=1;
303   end
304   for i=1:size(W,2)
305   for j=1:size(W,1)
306     box\_widthX=aspectXX*0.5*W(j,i)/(scale);
307     box\_widthY=aspectYY*0.5*W(j,i)/(scale);
308     h=fill([i-box\_widthX,i+box\_widthX,i+box\_widthX,i-box\
      _widthX],size(W,1)+1-[j-box\_widthY,j-box\_widthY,j+box\
      _widthY,j+box\_widthY],colstr(colour));
309     hold on
310     if (isnan(W(j,i)))
311     plot(i,size(W,1)+1-j,'kx','MarkerSize',20);
312     end
313
314     if (type==3)
315     set(h, 'EdgeColor','w');
316     box\_widthX=aspectXX*0.5*1;
317     box\_widthY=aspectYY*0.5*1;
318     h=fill([i-box\_widthX,i+box\_widthX,i+box\_widthX,i-box\
      _widthX],size(W,1)+1-[j-box\_widthY,j-box\_widthY,j+box\
      _widthY,j+box\_widthY],'w','FaceAlpha',0);
319     end
320
```

```
321   end
322   end
323   %axis off
324   %axis tight
325
326   if(equal==1), axis equal, end
327   axis([0.5,size(W,2)+0.5,+0.5,size(W,1)+0.5])
328
329 else
330   %draw as equal sized squares: strength indicated by darkness
        of square
331
332   if(equal==0)
333   %calc aspect ratio - if not going to set axis equal
334   plot(size(W,2)+0.5,size(W,1)-0.5,'bx');
335   hold on
336   plot(0.5,-0.5,'bx');
337   axis equal
338   a=axis;
339   aspectX=size(W,2)/abs(a(2)-a(1));
340   aspectY=size(W,1)/abs(a(4)-a(3));
341   aspectXX=aspectX./max(aspectX,aspectY);
342   aspectYY=aspectY./max(aspectX,aspectY);
343   hold off
344   else
345   aspectXX=1;
346   aspectYY=1;
347   end
348   box\_widthX=aspectXX*0.33;
349   box\_widthY=aspectYY*0.33;
350   for i=1:size(W,2)
351   for j=1:size(W,1)
352     fill([i-box\_widthX,i+box\_widthX,i+box\_widthX,i-box\
        _widthX],size(W,1)+1-[j-box\_widthY,j-box\_widthY,j+box\
        _widthY,j+box\_widthY],ones(1,4).*round((W(j,i)./scale)*
        255),'FaceColor','flat');
353     hold on
354   end
355   end
356   colormap(gray)
357   map=colormap;
358   map=flipud(map);
359   map(1:64,colour)=map(1:64,colour)*0.0+1;
360   colormap(map)
361   caxis([0,255])%if we remove this then each subplot is scaled
        independently
362   %axis off
363   %axis tight
364
365   if(equal==1), axis equal, end
366   axis([0.5,size(W,2)+0.5,+0.5,size(W,1)+0.5])
367
368 end
369 %set(gca,'YTickLabel',[' ';' ';' ';' ';' ';' ';' '])
```

```
%set(gca,'XTickLabel',[' ';' ';' ';' ';' ';' ';' '])
set(gca,'YTick',[])
set(gca,'XTick',[])
drawnow
```

A.2 Constructive PC/BC–DIM

This code is based on the PC/BC–DIM algorithm coded by Spratling [28]. The original version (`dim_learn_recip_weights.zip`) can be downloaded from his website:

`http://www.inf.kcl.ac.uk/staff/mike/code.html`

learn_bars_feedback.m

```
function [X,W,V,U,n,recognized,cycles]=learn_bars_feedback(gen,
    W,V,U)
%DEFINE NETWORK PARAMETERS
beta=0.005;                %learning rate
iterations=200;            %number of iterations to calculate y (
    for each input)
if nargin<1
  gen='std';               %type of bars pattern
end
p=5;                       %length of one side of input image/
    number of bars at
                           %each orientation
n=4;                       %number of nodes
m=p*p;                     %number of inputs
epochs=20;                 %number of training epochs
cycs=1000;                 %number of training cycles per epoch
patterns=400;              %number of training patterns in
    training set
noise=0.1;                 %amount of noise to add to each
    training pattern
continuousLearning=0;      %learn at every iteration, or using
    steady-state responses
dispresults=1;
figoffset=0;

%constructive parameters
t0 = 1;                          %time of last added neuron
window = 1;                      %window for slope calculation
tslope = 0.05;                   %trigger slope
exptsh = 1.20;                   %average error until exponential
    growth
cutavg = 0.50;                   %average error to cut growing
stpavg = 0.0;                    %average error to stop
mult   = 1.5;                    %multiplicative factor for growing

grow   = 1;                      %boolean value to control growing
eavgs  = zeros(1, epochs);       %average errors per epoch

```

```
32
33 %GENERATE TRAINING DATA
34 prob=1/p;
35 switch gen
36  case 'doubleoverlap'
37   p=p+1; m=p^2;
38  case 'quadwidth'
39   m=(p*4)^2;
40  case 'doublewidth'
41   m=(p*2)^2;
42   n=n*4
43  case 'unequal'
44   p=16;
45   m=p.^2;
46   %n=n*4
47 end
48 for k=1:patterns
49   X(:,k)=pattern_bars(p,prob,gen,noise);
50 end
51
52 %DEFINE INITIAL WEIGHTS
53 if nargin<2,
54   [W,V,U]=weight_initialisation_random(n,m);
55 end
56 if dispresults
57   figure(1+figoffset),clf, plot_bars(gen,W);
58   figure(2+figoffset),clf, plot_bars(gen,V);
59   figure(3+figoffset),clf, plot_bars(gen,U);
60 end
61
62 ymax=0;
63 y=[];
64 %TRAIN NETWORK
65 for t=1:epochs
66   fprintf(1, 'Epoch %i, ', t);
67   eavg = 0;
68
69   for k=1:cycs
70     %choose an input stimulus from the training set
71     patternNum=fix(rand*patterns)+1; %random order
72     x=X(:,patternNum);
73     %OR
74     %generate a new pattern at each training cycle
75     %x=pattern_bars(p,prob,gen,noise);
76
77     if continuousLearning
78     %calculate node activations and learn at each iteration
79     [y,e,W,V,U]=dim_activation(W,x,y,1+floor(rand*2*iterations)
       ,V,beta/iterations,U);
80     else
81     %OR calculate node activations for a set number of
       iterations then learn
82     [y,e]=dim_activation(W,x,y,iterations,V);
83     [W,V]=dim_learn(W,V,y,e,beta);
```

```
84      U=dim_learn_feedback(U,y,x,beta);
85      end
86
87      %update average error
88      if ~isempty(e(e>0)),
89        eavg = (eavg*(k-1) + mean(abs(e(e>0) - 1))) / k;
90      end;
91
92      ymax=max([ymax,max(y)]);
93    end
94
95    fprintf(1, 'nodes: %i, error: %f\n',n,eavg);
96    eavgs(t) = eavg;  %save average error into vector
97
98    %show results
99    if dispresults
100     recognized = 0;
101     set(0,'CurrentFigure',1+figoffset); recognized = recognized
       + plot_bars(gen,W);
102     set(0,'CurrentFigure',2+figoffset); recognized = recognized
       + plot_bars(gen,V);
103     set(0,'CurrentFigure',3+figoffset); recognized = recognized
       + plot_bars(gen,U);
104     disp([' ymax=',num2str(ymax),' wSum=',num2str(max(sum(W')))
       ,' vSum=',...
105         num2str(max(sum(V'))),' uSum=',num2str(max(sum(U')))]);
106     ymax=0;
107   end
108
109   %check stop condition
110   if eavgs(t) <= stpavg,
111       break;
112   end;
113
114   %check stop growing condition
115   if eavgs(t) <= cutavg,
116       grow = 0;
117   end;
118
119   %check growing condition
120   if grow,
121     %exponential growth
122     if eavg >= exptsh,
123       t0 = t;
124       n = round(n * mult);
125       [W,V,U]=weight_initialisation_random(n,m);
126       y = [];
127
128     %gradual growing
129     elseif t - window >= t0,
130       if (abs(eavgs(t) - eavgs(t - window)) / eavgs(t0)) <
       tslope,
131         t0 = t;
132         n = n + 1;
```

```
        W=[W; weight_initialisation_random(1,m)];
        V=[V; weight_initialisation_random(1,m)];
        U=[U; weight_initialisation_random(1,m)];
        y = [];
      end;
    end;
  end;
end

if dispresults
  sw=sum(W'), disp(num2str([max(sw),min(sw),max(max(W)),min(min(W))]))
  sv=sum(V'), disp(num2str([max(sv),min(sv),max(max(V)),min(min(V))]))
  su=sum(U'), disp(num2str([max(su),min(su),max(max(U)),min(min(U))]))
  disp('');
end

cycles = t*cycs;
```

Heaviside.m

```
function xt=Heaviside(x)

xt=x;
xt(x==0)=0.5;
xt(x<0)=0;
xt(x>0)=1;
```

dim_activation.m

```
function [y,e,W,V,U]=dim_activation(W,x,y,iterations,V,beta,U)
[n,m]=size(W);
psi=1;%5000; %need to learn using e-(1/psi)
epsilon1=0.0001; %>0.001 this becomes significant compared to y and hence
                 %produces sustained responses and more general suppression
epsilon2=100*epsilon1*psi;%this determines scaling of initial transient response

if nargin<3 || isempty(y), y=zeros(n,1,'single'); end

if nargin<4, iterations=25; end
if nargin<5,
  %set feedback weights equal to feedforward weights normalized by maximum value
  %V=W./(1e-9+(max(W')'*ones(1,m)));
  V=bsxfun(@rdivide,W,(1e-9+max(W,[],2)));
  U=V;
end

x=min(1,x);
```

```
%x=tanh(pi.*x);
for i=1:iterations
  %update responses
  e=x./(epsilon2+(V'*y));
  y=(epsilon1+y).*(W*e);

  %perform learning at every step - if required
  if nargout>2
  [W,V]=dim_learn(W,V,y,e,beta);
  end
  if nargout>4
  U=dim_learn_feedback(U,y,x,beta);
  end
end
```

dim_learn.m

```
function [W,V]=dim_learn(W,V,y,e,beta)
if nargin<5, beta=0.005; end

%update forward weights
delta=beta.*(y*(e'-1));
W=W.*(1 + delta);
W(W<0)=0;

if nargout>1
  %update feedback weights
  scale=beta.*Heaviside(y-1)*ones(size(e'));

  V=V.*(1 + delta+scale);
  V(V<0)=0;
end
```

dim_learn_feedback.m

```
function [U]=dim_learn_feedback(U,y,x,beta)
if nargin<4, beta=0.005; end

e=x./(1e-2+(U'*y));
delta=beta.*(y*(e'-1));

%update weights
U=U.*(1 + delta);
U(find(U<0))=0;
```

highest_integer_factors.m

```
function [a,b]=highest_integer_factors(x)
b=ceil(sqrt(x));
nofac=1;
b=b-1;
while b<x & nofac
  b=b+1;
```

```
  if x/b==floor(x/b)
  nofac=0;
  end
end
a=x/b;
```

hinton_plot.m

```
function hinton\_plot(W, scale, colour, type, equal)
%function hinton\_plot(W, scale, colour, type, equal)
% type 0 = variable size boxes (size relates to strength)
% type 1 = image (color intensity relates to strength)
% type 2 = equal size squares (color intensity relates to
    strength)
% type 3 = same as type 0 but with outerboarder showing maximum
     size of box

W(find(W<0))=0;

if (type==1)
  %draw as an image: strength indicated by pixel darkness

  %W is true data value (greater than 0) and is scaled to be
    between 0 and 255
  imagesc(uint8(round((W./scale)*255)),[0,255]);%,'CDataMapping
    ','scaled'),
  colormap(gray)
  map=colormap;
  map=flipud(map);
  map(1:64,colour)=map(1:64,colour)*0.0+1;
  colormap(map)
  axis on
  if(equal==1), axis equal, end
  if(equal==1), axis tight, end

elseif (type==0 | type==3)
  %draw as squares: strength indicated by size of square

  colstr=['r','g','b'];
  if(equal==0)
  %calc aspect ratio - if not going to set axis equal
  plot(size(W,2)+0.5,size(W,1)-0.5,'bx');
  hold on
  plot(0.5,-0.5,'bx');
  axis equal
  a=axis;
  aspectX=size(W,2)/abs(a(2)-a(1));
  aspectY=size(W,1)/abs(a(4)-a(3));
  aspectXX=aspectX./max(aspectX,aspectY);
  aspectYY=aspectY./max(aspectX,aspectY);
  hold off
  else
  aspectXX=1;
  aspectYY=1;
```

```
43    end
44    for i=1:size(W,2)
45    for j=1:size(W,1)
46      box\_widthX=aspectXX*0.5*W(j,i)/(scale);
47      box\_widthY=aspectYY*0.5*W(j,i)/(scale);
48      h=fill([i-box\_widthX,i+box\_widthX,i+box\_widthX,i-box\
        _widthX],size(W,1)+1-[j-box\_widthY,j-box\_widthY,j+box\
        _widthY,j+box\_widthY],colstr(colour));
49      hold on
50      if (isnan(W(j,i)))
51      plot(i,size(W,1)+1-j,'kx','MarkerSize',20);
52      end
53
54      if (type==3)
55      set(h, 'EdgeColor','w');
56      box\_widthX=aspectXX*0.5*1;
57      box\_widthY=aspectYY*0.5*1;
58      h=fill([i-box\_widthX,i+box\_widthX,i+box\_widthX,i-box\
        _widthX],size(W,1)+1-[j-box\_widthY,j-box\_widthY,j+box\
        _widthY,j+box\_widthY],'w','FaceAlpha',0);
59      end
60
61    end
62    end
63    %axis off
64    %axis tight
65
66    if(equal==1), axis equal, end
67    axis([0.5,size(W,2)+0.5,+0.5,size(W,1)+0.5])
68
69  else
70    %draw as equal sized squares: strength indicated by darkness
        of square
71
72    if(equal==0)
73    %calc aspect ratio - if not going to set axis equal
74    plot(size(W,2)+0.5,size(W,1)-0.5,'bx');
75    hold on
76    plot(0.5,-0.5,'bx');
77    axis equal
78    a=axis;
79    aspectX=size(W,2)/abs(a(2)-a(1));
80    aspectY=size(W,1)/abs(a(4)-a(3));
81    aspectXX=aspectX./max(aspectX,aspectY);
82    aspectYY=aspectY./max(aspectX,aspectY);
83    hold off
84    else
85    aspectXX=1;
86    aspectYY=1;
87    end
88    box\_widthX=aspectXX*0.33;
89    box\_widthY=aspectYY*0.33;
90    for i=1:size(W,2)
91    for j=1:size(W,1)
```

```
    fill([i-box\_widthX,i+box\_widthX,i+box\_widthX,i-box\
    _widthX],size(W,1)+1-[j-box\_widthY,j-box\_widthY,j+box\
    _widthY,j+box\_widthY],ones(1,4).*round((W(j,i)./scale)*
    255),'FaceColor','flat');
    hold on
  end
  end
  colormap(gray)
  map=colormap;
  map=flipud(map);
  map(1:64,colour)=map(1:64,colour)*0.0+1;
  colormap(map)
  caxis([0,255])%if we remove this then each subplot is scaled
    independently
  %axis off
  %axis tight

  if(equal==1), axis equal, end
  axis([0.5,size(W,2)+0.5,+0.5,size(W,1)+0.5])

end
%set(gca,'YTickLabel',[' ';' ';' ';' ';' ';' ';' '])
%set(gca,'XTickLabel',[' ';' ';' ';' ';' ';' ';' '])
set(gca,'YTick',[])
set(gca,'XTick',[])
drawnow
```

pattern_bars.m

```
function x=pattern_bars(p,prob,gen,noise)
%function x=pattern_bars(p,prob,gen,noise)
%
%create a pxp pixel image in which horizontal and vertical bars
%are randomly active with probability 'prob'

x=zeros(p,p,'single');

switch gen

 case 'std'
  %disp('standard bars');
  for l=1:2*p
  if (rand<prob)
    if(l<=p)                  %horizontal line - activate a row
    x(l,:)=1;
    else                      %vertical line - activate a column
    x(:,l-p)=1;
    end
    end
  end

 case 'diags'
  for l=1:2*p
  if (rand<prob)
```

```
26      if(l<=p)                    %horizontal line - activate a row
27      x(l,:)=1;
28      else                        %vertical line - activate a column
29      x(:,l-p)=1;
30      end
31      end
32    end
33    dl=3;
34    for d=-dl:dl
35    if (rand<0.5*prob)
36      dm=diag(ones(1,p-abs(d)),d);          %diagonal - backward
        sloping
37      x=max(x,dm);
38    end
39    end
40    for d=-dl:dl
41    if (rand<0.5*prob)
42      dm=fliplr(diag(ones(1,p-abs(d)),d)); %diagonal - forward
        sloping
43      x=max(x,dm);
44    end
45    end
46
47  case 'dbldiags'
48    for l=1:2*p
49    if (rand<prob)
50      if(l<=p)                    %horizontal line - activate a row
51      x(l,:)=1;
52      else                        %vertical line - activate a column
53      x(:,l-p)=1;
54      end
55      end
56    end
57    num=7;
58    for d=-ceil(num/2):2:ceil(num/2)
59    if (rand<0.5*prob)
60      dm1=diag(ones(1,p-abs(d)),d);          %diagonal - backward
        sloping
61      x=max(x,dm1);
62      dm2=diag(ones(1,p-abs(d+1)),d+1);          %diagonal -
        backward sloping
63      x=max(x,dm2);
64    end
65    end
66    for d=-ceil(num/2):2:ceil(num/2)
67    if (rand<0.5*prob)
68      dm1=fliplr(diag(ones(1,p-abs(d)),d)); %diagonal - forward
        sloping
69      x=max(x,dm1);
70      dm2=fliplr(diag(ones(1,p-abs(d+1)),d+1)); %diagonal -
        forward sloping
71      x=max(x,dm2);
72    end
73    end
```

```
74
75  case 'tripplediags'
76   for l=1:2*p
77   if (rand<prob)
78     if(l<=p)                    %horizontal line - activate a row
79     x(l,:)=1;
80     else                        %vertical line - activate a column
81     x(:,l-p)=1;
82     end
83     end
84   end
85
86   for d=-7:3:7
87   if (rand<0.25*prob)
88     for dd=0:2
89     dm=diag(ones(1,p-abs(d+dd)),d+dd);          %diagonal -
       backward sloping
90     x=max(x,dm);
91     end
92   end
93   end
94   for d=-7:3:7
95   if (rand<0.25*prob)
96     for dd=0:2
97     dm=fliplr(diag(ones(1,p-abs(d+dd)),d+dd)); %diagonal -
       forward sloping
98     x=max(x,dm);
99     end
100  end
101  end
102
103 case 'stdlinear'
104  %disp('linear bars');
105  for l=1:2*p
106  if (rand<prob)
107    if(l<=p)                    %horizontal line - activate a row
108    x(l,:)=x(l,:)+1;
109    else                        %vertical line - activate a column
110    x(:,l-p)=x(:,l-p)+1;
111    end
112    end
113  end
114
115 case 'onebar'
116  %disp('one bar patterns');
117  l=fix(rand*2*p)+1;
118  if(l<=p)                   %horizontal line - activate a row
119    x(l,:)=1;
120  else                       %vertical line - activate a column
121    x(:,l-p)=1;
122  end
123
124 case 'oneorient'
125  %disp('one orientation bars');
```

```
126   orientation=round(rand);
127   for l=1:2*p
128     if (rand<prob);
129     if(l<=p & orientation==0)            %horizontal line -
      activate a row
130         x(l,:)=1;
131     elseif(l>p & orientation==1)          %vertical line -
      activate a column
132       x(:,l-p)=1;
133     end
134     end
135   end
136
137  case 'horizonly'
138   orientation=0;
139   for l=1:2*p
140     if (rand<prob);
141     if(l<=p & orientation==0)            %horizontal line -
      activate a row
142         x(l,:)=1;
143     elseif(l>p & orientation==1)          %vertical line -
      activate a column
144       x(:,l-p)=1;
145     end
146     end
147   end
148
149  case 'vertonly'
150   orientation=1;
151   for l=1:2*p
152     if (rand<prob);
153     if(l<=p & orientation==0)            %horizontal line -
      activate a row
154         x(l,:)=1;
155     elseif(l>p & orientation==1)          %vertical line -
      activate a column
156       x(:,l-p)=1;
157     end
158     end
159   end
160
161  case 'twobars'
162   %disp('two bar patterns');
163   x=fixedbars(x,p,2);
164
165  case 'threebars'
166   %disp('three bar patterns');
167   x=fixedbars(x,p,3);
168
169  case 'fourbars'
170   %disp('four bar patterns');
171   x=fixedbars(x,p,4);
172
173  case 'fivebars'
```

```
 %disp('five bar patterns');
 x=fixedbars(x,p,5);

case 'sixbars'
 %disp('six bar patterns');
 x=fixedbars(x,p,6);

case 'sevenbars'
 %disp('seven bar patterns');
 x=fixedbars(x,p,7);

case 'doublewidth'
 %disp('double width bar patterns (no overlap)');
 x=zeros(p*2,p*2);
 for l=1:2*p
 if (rand<prob)
   if(l<=p)                %horizontal line - activate a row
   x(l*2-1:l*2,:)=1;
   else                      %vertical line - activate a column
   x(:,(l-p)*2-1:(l-p)*2)=1;
   end
   end
 end

case 'quadwidth'
 %disp('4 pixel wide bar patterns (no overlap)');
 x=zeros(p*4,p*4);
 for l=1:2*p
 if (rand<prob)
   if(l<=p)                %horizontal line - activate a row
   x(l*4-3:l*4,:)=1;
   else                      %vertical line - activate a column
   x(:,(l-p)*4-3:(l-p)*4)=1;
   end
 end
 end

case 'doubleoverlap'
 %disp('double width bar patterns');
 for l=[1:p-1,p+1:2*p-1]
 if (rand<prob)
   if(l<=p)                %horizontal line - activate a row
   x(l:l+1,:)=1;
   elseif (l>p) %vertical line - activate a column
   x(:,l-p:l+1-p)=1;
   end
   end
 end

case 'unequal'
 if p~=16, disp('ERROR: image size of 16 expected for unequal
   bars problem');end
 prob_scale=4;
 %horizontal= 7x 1-pixel-wide plus 1x 9-pixel-wide
```

```
  for l=1:7
  if (rand<prob/prob_scale)
    x(l,:)=1;
  end
  end
  if (rand<prob/prob_scale) x(8:16,:)=1; end
  %vertical = 7x 1-pixel-wide plus 1x 9-pixel-wide
  for l=1:7
  if (rand<prob)
    x(:,l)=1;
  end
  end
  if (rand<prob) x(:,8:16)=1; end

 otherwise
  disp('ERROR: no bars problem defined');
end

x=x(:);

%add noise
if noise>0
  %add noise with given variance
%  x=x+(randn(size(x)).*sqrt(noise));
%  x(find(x>1))=1;
%  x(find(x<0))=0;

  %add bit-flip noise
  for i=1:length(x)
  if rand<noise
    if x(i)>0.5, x(i)=0;
    else x(i)=1; end
  end
  end
end


function x=fixedbars(x,p,num)
randSet=[];
for j=1:num
  newbar=0;
  while newbar==0
  l=fix(1+(rand*2*p));
  newbar=1;
  for i=1:j-1
    if l==randSet(i)
    newbar=0;
    end
  end
  end
  randSet(j)=l;
  if(l<=p)                    %horizontal line - activate a row
  x(l,:)=1;
  else                        %vertical line - activate a column
```

```
  x(:,l-p)=1;
  end
end
```

plot_bars.m

```
function [nrepanycomplete]=plot_bars(gen,weights,label_plots,
    scale,scale_bar,noplot)
%function [nrep]=plot_bars(gen,weights,label_plots,scale,
    scale_bar)
[n,m]=size(weights);
p=sqrt(m);
thres=2;
if nargin<3, label_plots=1; end
if nargin<4 | isempty(scale), scale=max(max(weights))*0.9; end
if nargin<5, scale_bar=0; end
if nargin<6, noplot=0; end
[plotRows,plotCols]=highest_integer_factors(n);
clf

switch gen
 case 'tripplediags'
  diags=-7:3:7;
  lines=2*(p+length(diags));
 case 'dbldiags'
  diags=-ceil(7/2):2:ceil(7/2);
  lines=2*(p+length(diags));
 case 'diags'
  diags=-3:3;
  lines=2*(p+length(diags));
 case 'doublewidth'
  lines=2*(p/2);
 case 'quadwidth'
  lines=2*(p/4);
 case 'doubleoverlap'
  lines=2*(p-1);
  thres=1.5;
 case 'unequal'
  lines=16;
  thres=1.5;
 otherwise
  lines=2*p;
end
representedcomplete=zeros(1,lines);

for j=1:n
  %test and plot weights corresponding to one node
  w=reshape(weights(j,:),p,p);
  if ~noplot
  if scale_bar
    maxsubplot(plotRows+scale_bar,plotCols,j+plotCols),
  else
    subplot(plotRows,plotCols,j),
  end
```

```
47  hinton_plot(flipud(w'),scale,3,1,1);
48  end
49
50  %calculate minimum weight corresponding to all possible
      components
51  rfmin=[min(w), min(w')];
52  switch gen
53   case 'tripplediags'
54  for k=diags
55    rfmin=[rfmin, min([diag(w,k);diag(w,k+1);diag(w,k+2)]), min
      ([diag(fliplr(w),k);diag(fliplr(w),k+1);diag(fliplr(w),k+2)
      ])];
56  end
57   case 'dbldiags'
58  for k=diags
59    rfmin=[rfmin, min([diag(w,k);diag(w,k+1)]), min([diag(
      fliplr(w),k);diag(fliplr(w),k+1)])];
60  end
61   case 'diags'
62  for k=diags
63    rfmin=[rfmin, min(diag(w,k)), min(diag(fliplr(w),k))];
64  end
65   case 'doublewidth'
66  k=0;
67  for i=1:2:p
68    k=k+1;
69    twobarmin(k)=min([rfmin(i),rfmin(i+1)]);
70    twobarmin(k+p/2)=min([rfmin(i+p),rfmin(i+p+1)]);
71  end
72  rfmin=twobarmin;
73   case 'quadwidth'
74  k=0;
75  for i=1:4:p
76    k=k+1;
77    fourbarmin(k)=min([rfmin(i),rfmin(i+1),rfmin(i+2),rfmin(i
      +3)]);
78    fourbarmin(k+p/4)=min([rfmin(i+p),rfmin(i+p+1),rfmin(i+p+2)
      ,rfmin(i+p+3)]);
79  end
80  rfmin=fourbarmin;
81   case 'doubleoverlap'
82  for i=1:p-1
83    twobarmin(i)=min([rfmin(i),rfmin(i+1)]);
84    twobarmin(i+p-1)=min([rfmin(i+p),rfmin(i+p+1)]);
85  end
86  rfmin=twobarmin;
87   case 'unequal'
88  rfmin=[min(w(1:7,:)'),min(min(w(8:16,:))),min(w(:,1:7)),min(
      min(w(:,8:16)))];
89  end
90
91
92  %calculate maximum weight corresponding to all possible
      components
```

```
93  rfmag=[sum(w), sum(w')];
94  switch gen
95   case 'tripplediags'
96  for k=diags
97    rfmag=[rfmag, sum([diag(w,k);diag(w,k+1);diag(w,k+2)]), sum
     ([diag(fliplr(w),k);diag(fliplr(w),k+1);diag(fliplr(w),k+2)
     ])];
98  end
99   case 'dbldiags'
100 for k=diags
101   rfmag=[rfmag, sum([diag(w,k);diag(w,k+1)]), sum([diag(
     fliplr(w),k);diag(fliplr(w),k+1)])];
102 end
103  case 'diags'
104 for k=diags
105   rfmag=[rfmag, sum(diag(w,k)), sum(diag(fliplr(w),k))];
106 end
107  case 'doublewidth'
108 k=0;
109 for i=1:2:p
110   k=k+1;
111   twobarmag(k)=rfmag(i)+rfmag(i+1);
112   twobarmag(k+p/2)=rfmag(i+p)+rfmag(i+p+1);
113 end
114 rfmag=twobarmag;
115  case 'quadwidth'
116 k=0;
117 for i=1:4:p
118   k=k+1;
119   fourbarmag(k)=rfmag(i)+rfmag(i+1)+rfmag(i+2)+rfmag(i+3);
120   fourbarmag(k+p/4)=rfmag(i+p)+rfmag(i+p+1)+rfmag(i+p+2)+
     rfmag(i+p+3);
121 end
122 rfmag=fourbarmag;
123  case 'doubleoverlap'
124 for i=1:p-1
125   twobarmag(i)=rfmag(i)+rfmag(i+1);
126   twobarmag(i+p-1)=rfmag(i+p)+rfmag(i+p+1);
127 end
128 rfmag=twobarmag;
129  case 'unequal'
130 rfmag=[sum(w(1:7,:)'),sum(sum(w(8:16,:)))/9,sum(w(:,1:7)),sum
     (sum(w(:,8:16)))/9];
131 end
132 [rfmax,lpref]=max(rfmag);
133 rfmag(lpref)=0; %so that max(rfmag) will now be for the 2nd
     best
134                 %represented component
135
136 %decide any component meets criteria for being represented
137 if rfmax>thres*max(rfmag) & rfmin(lpref)>mean(mean(w))
138 representedcomplete(lpref)=representedcomplete(lpref)+1;
139 if label_plots
140   if representedcomplete(lpref)>1,
```

```
    text(1,0.25,['(',int2str(lpref),')']);
    else
    text(1,0.25,int2str(lpref));
    end
  end
  end
end
nrepanycomplete=length(find(representedcomplete>0));
disp(['weigths represent ', int2str(nrepanycomplete), '
    patterns ']);
norep=find(representedcomplete==0);
if length(norep>1) disp(['FAILED to represent pattern = ',
    int2str(norep)]); end

if scale_bar & ~noplot
  maxsubplot(plotRows+scale_bar,1,1)
  plot_bar([0,scale],'t');
  set(gcf,'PaperPosition',[1 1 6 5]);
end

function plot_bar(range,position)
imagesc(range(1),range);
axis('off')
if position=='l'
  colorbar('East');
elseif position=='r'
  colorbar('West');
elseif position=='t'
  colorbar('North');
elseif position=='b'
  colorbar('South');
end
```

weight_initialisation_random.m

```
function [W,V,U]=weight_initialisation_random(n,m,wMean,wStd)
if nargin<3, wMean=0.5; end
if nargin<4, wStd=0.05; end

W=gaussian_weights(n,m,wMean,wStd);
V=gaussian_weights(n,m,wMean,wStd);
U=gaussian_weights(n,m,wMean,wStd);


function W=gaussian_weights(n,m,wMean,wStd)
W=wMean+wStd.*randn(n,m,'single'); %Gaussian distributed
    weights with given
W(W<0)=0;                          %mean and standard deviation
```