

Networks with emotions

An investigation into deep belief nets and emotion recognition

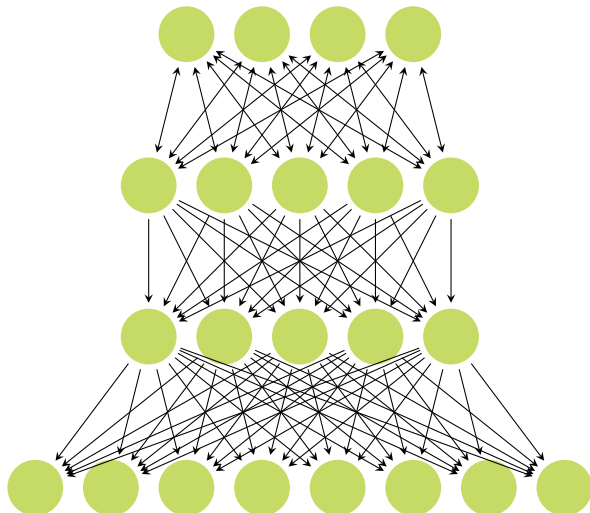
Author:
Mihaela Rosca

Supervisor:
Prof. Abbas Edalat

MEng Final Year Project

Department of Computing,
Imperial College London

June 2014



Abstract

Facial emotion recognition from images has been a field of intense research for decades. The difficulty of the problem lies in its interdisciplinary nature. Psychology, neuroscience and machine learning concern themselves with how we can teach computers to detect emotions in humans. Emotion recognition can be divided in two main tasks: feature extraction and emotion classification. We tackle the problem by using deep belief networks, a type of neural network that allows both feature detection and classification. In our quest towards a machine capable of recognizing human feelings, we define a probabilistic model that can learn if two human subjects display the same emotion and compare its performance against a model capable of distinguishing between subjects. We discuss and evaluate numerous techniques associated with deep belief nets, providing a comprehensive account of the literature. We add to our theoretical work a modular, fast, open source GPU implementation of all models used. The implementation is used to perform empirical evaluation on multiple datasets, with different subjects of different races, under different illuminations and different poses. We measure robustness by testing the network with images with patches of missing data. The results obtained on our experiments are positive: a classification accuracy of **99.3%** on the Multi-PIE dataset makes our results comparable with the state of the art.

Acknowledgements

I would like to thank my supervisor, Prof. Edalat for his constant support and guidance. His detailed feedback has helped me improve as a scientist, while his trust in giving me the freedom of choosing directions of work has helped me improve as a researcher.

My second supervisor, Prof. Murray Shanahan has shown so much enthusiasm. Our conversations about machine learning and artificial intelligence have been highly stimulating.

A special thanks to the Computing Support Group here at Imperial. This project would have been impossible without them. First, they got the high performance graphics card which enabled me to run experiments. Then they provided me with help every time something went wrong with it.

Acknowledgements go to George Papamakarios and James Thewlis for their constructive feedback on the report and to Niklas Hambüchen for valuable discussions throughout the duration of the project.

Last but not least, I have to thank my parents. For everything.

Contents

Table of contents	i
List of Figures	v
List of Tables	viii
1 Introduction	1
1.1 Motivation	2
1.2 Contributions	3
2 Background	4
2.1 Short history of neural networks	4
2.2 What are neural networks?	4
2.3 Types of learning	6
2.4 Discriminative learning in neural networks	7
2.4.1 The error function	7
2.4.2 The backpropagation algorithm	8
2.4.3 Computing the derivatives in backpropagation	9
2.4.4 Using the error derivative	11
2.4.5 Parameters and techniques: how to use the gradients	12
2.5 Softmax groups	15
2.6 Overfitting	18
2.7 Dropout	21
2.7.1 Dropout as model averaging	21
2.7.2 Biological intuition	22
2.7.3 Forward pass in dropout nets	22
3 Deep belief nets	23
3.1 Restricted Boltzmann machines	23
3.1.1 Getting there: Hopfield networks and Boltzmann machines	23

3.1.2	Restricted Boltzmann machines	25
3.1.3	Training an RBM: Contrastive divergence	27
3.1.4	Persistent Contrastive Divergence	28
3.1.5	Initialization of parameters	29
3.1.6	Monitoring learning: Free energy, reconstruction error and AIS	29
3.1.7	Convergence of training	31
3.1.8	Real valued units	31
3.1.9	Sparse hidden units	33
3.1.10	Other methods	33
3.2	Deep belief networks	34
3.2.1	Greedy pre-training	35
3.2.2	Theoretical justification of the greedy learning	37
3.2.3	Classification using deep belief nets	38
3.2.4	Better generative models: Contrastive wake sleep	39
3.2.5	Improving training in deep belief nets	40
4	Putting it all together: our model	41
4.1	Restricted Boltzmann Machine	41
4.1.1	Techniques	41
4.1.2	Sparsity constraints	41
4.2	Deep belief nets	43
4.2.1	Techniques	43
4.2.2	Pre-training heuristic	43
4.2.3	Norm constraints	44
4.2.4	Momentum	44
4.3	Types of units	44
4.4	Practical considerations	45
5	A standard benchmark: recognizing handwritten digits	49
5.1	Learning features	49
5.2	Learning labels	53

6	Emotion recognition	56
6.1	Data preprocessing	56
6.2	Cohn-Kanade, Jaffe and other databases	56
6.3	Multi PIE	59
6.3.1	Random data splits	61
6.3.2	Different subjects	61
6.3.3	Different illuminations	61
6.3.4	Different poses	62
6.3.5	Missing data	63
6.4	A wild dataset	64
6.5	Emotion similarity	65
6.5.1	Same person	67
6.5.2	Same emotion	68
6.5.3	Same subjects, different emotions	70
7	Implementation	72
7.1	Setup	72
7.2	Runtime speed concerns	73
7.2.1	Theano	73
7.2.2	OpenBlas	74
8	Evaluation	77
8.1	Handwritten digits	77
8.2	Emotion recognition	78
8.2.1	Maxout nets	79
8.2.2	Commercial applications	80
8.3	Similarities	80
8.3.1	Same subjects	80
8.4	Implementation	82

9	Future work	84
9.1	Improving the current model	84
9.2	Data preprocessing	84
9.3	Different models for unsupervised pre-training	85
9.4	Emotion recognition from video	85
10	Conclusion	86
A	Notation	87
B	Deriving the activations of a unit in a RBM	89
C	Why greedy learning works: detailed mathematical explanation	90
D	Expected value of a noisy rectified unit	92
	Index	94
	References	95

List of Figures

2.1	Representation of a feed forward neural network with 2 hidden layers.	5
2.2	Parameter updates according to classical momentum.	13
2.3	Parameter updates according to Nesterov momentum.	14
2.4	Comparison between a forward pass without dropout (left) and with dropout p (right).	22
2.5	Activation of a unit in a network trained with dropout during training (left) and testing (right).	22
3.1	Hopfield network with 8 nodes.	24
3.2	Restricted Boltzmann with 5 visible units and 4 hidden units.	25
3.3	A deep belief network as a top down generative model.	35
3.4	Required network architecture for initializing the weights of a network to the transpose of the weights of the network trained before it. The weights in black are the ones <i>after</i> training of the corresponding RBM, and the ones in red are <i>before</i> training the RBM.	36
3.5	Generative versus recognition weights in a DBN. The recognition weights are depicted in red.	37
4.1	Validation error on 10000 epochs on a network trained without Rmsprop.	47
4.2	Validation error on 1000 epochs on a network trained with Rmsprop.	48
5.1	Digit examples from the MNIST dataset.	49
5.2	Pictorial representation of how to obtain the activation of a hidden unit in a RBM. \odot denotes the Hadamard (elementwise) product between two matrices.	50
5.3	The incoming weights of 100 of the 500 hidden units of an RBM trained using Contrastive Divergence with instances of the digit 2 from the MNIST dataset. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.	51
5.4	The incoming weight vectors of 100 of the 500 hidden units of an RBM trained using Contrastive Divergence with instances of all 10 digits in the MNIST dataset. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.	51
5.5	The incoming weight vectors of 100 of the 500 hidden units of an RBM trained using Persistent Contrastive Divergence with instances of the digit 2 from the MNIST dataset. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.	52

5.6	The incoming weight vectors of 100 of the 500 hidden units of an RBM trained using Persistent Contrastive Divergence with instances of all 10 digits. Weights are reshaped to have the same shape as the image inputs for visualisation purposes. . .	52
5.7	The visible reconstruction of a random pattern using an RBM trained using Contrastive Divergence with 2s from the MNIST dataset.	54
5.8	The reconstruction of a random pattern using an RBM trained using Contrastive Divergence with samples from all 10 digits.	54
5.9	The reconstruction of an instance of the digits 7 using an RBM trained using Contrastive Divergence with instances of all digits.	54
5.10	The reconstruction of an instance of the digits 7 using an RBM trained using Contrastive Divergence with only instances of the digits 2.	54
5.11	Examples of misclassified digits from MNIST.	55
6.1	Visualization of image preprocessing techniques used before training a classifier to label emotions.	57
6.2	Images from the Cropped Kanade database. Emotions displayed (left to right): anger, disgust, fear, happiness, sadness, surprise and neutral.	58
6.3	Reconstruction of a face from the Cropped Kanade database, using a Restricted Boltzmann machine.	58
6.4	Visual comparison of weights of an RBM trained with and without Rmsprop. . . .	59
6.5	Images of subjects from the Multi PIE database. Emotions (from left to right): neutral, surprise, squint, smile, disgust, scream. Each subject is shown in a different pose and different illumination.	60
6.6	The 5 different poses displayed in the Multi PIE database.	63
6.7	Test images from the missing data experiment.	63
6.8	Facial areas according to contribution to the classification error in the missing data experiments. A stronger colour indicates a higher error when the network was deprived of the image pixels in the corresponding area (by making the pixels black). We notice that the most important area is around the mouth, followed by the region between the eyes (which is helpful to distinguish frowning and screaming). . . .	64
6.9	Images from the dataset used in the Kaggle competition for emotion recognition. .	66
6.10	Network architecture used for similarity detection.	67
6.11	Example of inputs presented when detecting if the emotions are the same. 7 pairs are shown, aligned vertically.	70
7.1	Comparison between Theano (left) and Numpy (right) code for matrix multiplication.	74
7.2	Comparison between Theano (left) and Numpy (right) raising to matrix power. Adapted from [65].	75

7.3	A part of the Theano expression graph of the training function used for the deep belief network implementation in this project.	75
8.1	A maxout net with 5 visible units and 3 hidden units and 2 pieces for each hidden unit.	80
A.1	A layer in a neural network. Used to exemplify the notation in this thesis.	88

List of Tables

4.1	Relation between test accuracy and number of training epochs when not using Rmsprop on a network built to classify emotions.	48
5.1	Classification results obtained on MNIST. Details about the experiments are found in text.	55
6.1	Databases used for emotion recognition.	58
6.2	Accuracy obtained when training a deep belief net with various unlabelled datasets. The only labelled dataset is the Cropped Kanade dataset. Details found in text. . . .	59
6.3	Average confusion matrix obtained by training splitting data in folds with a ratio 4 to 1 (train to test). Average classification rate: 99.3%	61
6.4	The classification accuracies obtained when testing with an illumination to which the network was not exposed to during training. Average classification rate: 89% . .	62
6.5	Confusion matrix obtained when testing with illumination type 3 and training with illuminations 1, 2, 4, 5. Experiment classification accuracy: 59.8%	62
6.6	Average confusion matrix obtained when equalizing the input. Average obtained accuracy is 94.8%	62
6.7	The classification accuracies obtained when testing with a pose to which the network was not exposed to during training.	63
6.8	Comparison of experiments performed on the Multi PIE database.	65
6.9	Results from the classification task of determining if the people displayed in 2 images are the same or not.	68
6.10	Results from the classification task of determining if the people displayed in 2 images are the same or not.	69
6.11	Average predicted probabilities by a similarity network when it is presented with two instances of the same subject, but under different emotions.	71
7.1	Speed comparison between multiple libraries when training a deep belief net with 5 layers with MNIST data. Training set size 10000. Testing set size 10. Experiments run on a GPU used a Quadro 6000 graphics card, and the ones on an Intel Core i7-2600 CPU with 3.40GHz, 8 cores and 8GB RAM. The experiments were performed by fixing the seed of the random generators, in order to accurately compare results. Due to the limited computation performed on the CPU, adding too many threads decreases performance, due to context switch overhead.	76
8.1	Classification results obtained on the MNIST dataset by various techniques. Results which involve no preprocessing, in order to be able compare with our results. The values were obtained from the MNIST official page and [7]. The estimated human accuracy is obtained from [66].	77

8.2	Overview of techniques for emotion recognition from images and their results on different datasets. Details about each method found in text.	79
8.3	Accuracy of maxout nets when classifying the emotions in the Multi Pie dataset. . .	80
8.4	Comparison of experiments performed for detecting if two images represent the same person.	81
8.5	Comparison between the library presented in this paper (pydeeplearn) and nolearn . Details found in text.	83
A.1	Abbreviations used in this thesis.	88

1 | Introduction

In the last decade the progress made by machine learning has been astounding, allowing it to grow in popularity both as a research field and as a basis for commercial applications.

Hard coded rules and rigid behaviour have been replaced with algorithms capable of generalization, leading to recent developments in speech recognition, computer vision, natural language processing as well as computational finance and medical diagnosis.

The present work focuses on a specific domain of machine learning, namely artificial neural networks. Since Geoffrey Hinton introduced deep belief nets (DBNs) in 2006, neural networks have become a popular tool for vision and speech tasks, breaking record after record in known benchmarks (such as TIMIT [1]).

Deep neural nets have been around for more than 30 years, but standard training methods have serious limitations when used on architectures of more than 2 layers. DBNs solve some of the problems associated with deep networks through pre-training: learning hierarchical features from data in an unsupervised fashion by stacking together multiple Restricted Boltzmann machines.

The main application of deep belief networks presented in this thesis is emotion recognition from images of faces. We use unsupervised training to learn facial features and supervised training to associate these features with emotions.

Classifying emotions straight from raw pixels is infeasible. Defining a direct mapping between a high dimensional image and a class label is prevented by the little individual influence of each of the pixels on the label. In order to learn how to classify emotions, higher level facial features need to be extracted from the raw input images. These features can then be associated with emotions. It is argued [2] that in order to overcome the current problems of emotion detection algorithms scientists should focus on accurate feature extraction, rather than on the classification process. This justifies our choice of model inside the neural network family: unsupervised pre-training in deep belief networks achieves accurate feature extraction in the face space, before performing any classification task. Unsupervised feature learning comes in handy especially when labelled data is scarce.

The features of our model are constructed through DBN training. Another type of features, action units, express the position and movement of facial muscles [3]. Automatic detection of action units can be done from sequences of images by tracking muscle movements in time [4] or from still images, in which case deep belief nets can be used [5]. Local binary patterns are yet another type of features which are used in emotion recognition [6]. It could be argued that we could use action units or local binary patterns as inputs to DBNs, instead of raw images. However, deep belief nets thrive by detecting the natural structure high dimensional data, so presenting unstructured, low dimensional features (such as AUs) we would lead to a decrease in classification performance. Moreover, the network will be merely used as a classifier, thus not making use of its feature learning capabilities. While most methods for emotion recognition require two pipelines, one for feature extraction and one for classification, the advantage of deep belief networks is that they learn how to perform both tasks during training.

1.1 Motivation

The human ability of detecting emotions evolves in infancy and is associated with what is known as “emotional intelligence”. When trying to explain how humans perceive and detect emotions, two main models have been proposed by neuroscientists: the *continuous* and the *categorical* model [2]. In the continuous model each emotion is a feature vector in the face space, while in the categorical model each discrete emotion has a classifier associated with it. Both models have advantages and disadvantages: the continuous model accounts for the different intensities of emotions while the categorical model explains why a morphing sequence between two emotions is perceived to be one of the two emotions, not something in between. In machine learning the categorical model predominates emotion recognition. We believe this to be the case due to the compatibility of the categorical framework with standard classification techniques present in the field.

Apart from the theoretical interests which arise from the subject of emotion recognition, there are numerous applications which make this field an active research topic.

Autistic children do not have the ability of recognizing emotions (or their ability to do so is diminished, depending on the severity of the condition). However, the children can be taught how to detect an emotion, by presenting them sequences of images of faces with their corresponding emotions attached. While this can be done with a standard labelled dataset, that dataset is finite and probably does not contain the family and friends of the child, hence becoming less familiar and motivating. A system that identifies emotions on previously unseen faces can be given to classify pictures or videos of loved ones, gradually helping the child develop a association between facial expressions and emotions.

A similar system can be used in a **judicial** setting to detect if a suspect is displaying genuine feelings or just pretending. The aim would be to exceed the human capability of detecting emotions, by spotting brief moments of weakness in which the individual is not maintaining its false, pretended pose.

Emotion recognition has multiple applications in **entertainment**. Gaming experiences can be enhanced if the action increases pace when the player is bored and slows down when the player is overwhelmed. A music phone application which shuffles songs according to the listener’s face expression would provide a more personalized experience. In the context of **social media**, automatic emotion detection from images can be used for targeted advertisement or tagging of images, as well as status updates: the user can upload an image of themselves, and their status can change to reflect their emotional state.

Finally, we mention that any true **artificial intelligent system** that interacts with humans should detect our emotions (think of an intelligent operating system or a robot).

1.2 Contributions

The key contributions of this thesis are:

- Evaluating deep belief nets on emotion recognition by using three different labelled datasets of different sizes, depicting people from different angles and under various illumination conditions,
- Defining a new model to test the capacity of Restricted Boltzmann machines of learning similarities between emotions,
- Introducing a theoretical extension of two types of sparsity constraints for rectified linear units,
- Producing a complete survey on the literature on neural networks (with special focus on Restricted Boltzmann machines and deep belief nets) providing a comprehensive account of recently introduced techniques such as dropout, rectified linear units and rmsprop,
- Providing a first assessment on the performance in emotion recognition of a recently introduced type of network: maxout nets [7],
- Analysing the methods available for facial emotion recognition from still images,
- Defining a heuristic for setting the learning rate for the first Restricted Boltzmann machine that is stacked to form a deep belief network,
- Implementing a modular, high performance, open source Python library that runs on the GPU for deep belief nets and Restricted Boltzmann machines

Deep belief nets can be used to simulate the brain on a metaphorical level, enabling researchers to emulate physiological experiments. The results obtained on emotion recognition in this thesis will be used in other projects to create a model for psychotherapy and modelling attachment types.

Appendix A introduces the reader to the notations and the abbreviations used in this thesis.

2 | Background

Artificial neural networks (ANNs) are machine learning models inspired by the animal brain in the hope that they can be used to reverse engineer how animals learn to perform certain tasks.

Even though ANNs are inspired by biological neural networks, they are far from being biologically realistic, as both the structure and the building blocks of ANNs are oversimplifications of their counterparts found in nature.

In the following, when we refer to a neural network we will mean artificial neural network.

2.1 Short history of neural networks

Neural networks were first introduced in 1943 by Warren McCulloch and Walter Pitts, who were aiming to create a mathematical model of a brain [8]. In 1949 Donald Hebb introduced what is now known as Hebbian learning, usually summed up under the slogan “Neurons that fire together wire together”.

An increased interest in the field was observed after Frank Rosenblatt introduced the perceptron in 1958, showing how a simple mathematical algorithm can be used to train a two layer network [9]. However, Seymour Papert and Marvin Minsky exposed some limitations of neural networks, proving that the exclusive-or function cannot be learned by a perceptron, as well as demonstrating that at the time it was simply infeasible to train a large neural net, due to the restricted amount of computer power resources [10].

After this drawback in neural network research, interest increased again in the late 70s and beginning of the 80s due to the discovery of backpropagation (which solved the exclusive-or problem) and the advances in computing architectures.

In the 90s other machine learning methods such as support vector machines [11] became popular. They had a clear theoretical explanation and solved a multitude of problems better than neural nets. This is now thought to be due to the usage of small datasets, not enough CPU resources and very little use of backpropagation.

Since the 2000s, neural networks have risen increasing interest due to the advances in convolutional neural nets (known as CNNs, or LeNet after Yan LeCun [12], who has improved the initial design made by Kuniyuki Fukushima), and deep belief nets (proposed by Geoffery Hinton in 2006 [13]).

2.2 What are neural networks?

The structure of a neural network can be seen as a graph. In this context the vertices are called *nodes* or *neurons* and edges are called *synaptic connections*. An important characteristic of ANNs is that the synaptic connections have strengths (called weights) which adapt in the learning process (similarly with what happens in the brain of an animal, especially in the early stages of life).

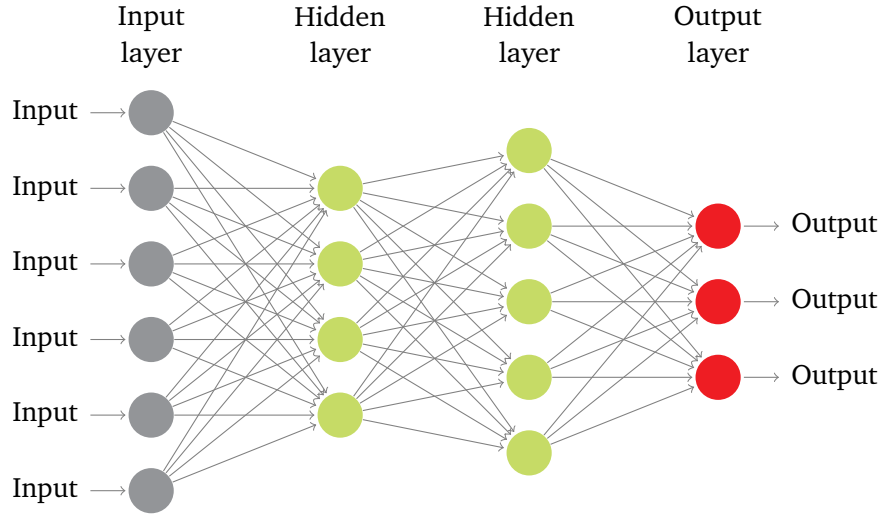


Figure 2.1: Representation of a feed forward neural network with 2 hidden layers.

The networks are structured in layers, according to the connections between nodes. Generally, the nodes inside a layer are not connected with each other, but are connected to all the nodes in the following layer.

The layers can be grouped as follows:

- The input layer storing the given data
- The hidden layers used to define a better representation of the data than the one given by the input
- The output layer contains the output, after a pass through the network. Only required for nets used for discrimination.

The input data is transformed into a vector of real numbers and presented to the network. A pass is performed and the neurons get activated, taking values (binary -on/off - or real numbers) also called *states* or *activations*. The output layer can be missing, depending on the task performed by the neural network.

The state (or activation of a neuron) is typically a real value that depends on the activities in the previous layer and the weights, as follows:

$$y_i^{(l+1)} = \sigma \left(\sum_j w_{ij} y_j^{(l)} + b_i \right) \quad (2.1)$$

Here, b_i is the bias associated with the unit and σ is referred to as the *activation function*. Figure A.1 exemplifies the notation we will use when referring to different layers in a neural net.

The activation function plays an important role in the network, as it is used to restrict the range of values the activity of a neuron can have. Frequently activation functions come from the sigmoid family (such as the logistic sigmoid and tanh). The range of the logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ is $(0, 1)$, making the function particularly suitable when the activations of the neurons represent probabilities. Recent developments have shown that rectifier functions such as $\max(0, x)$ perform better for certain kind of tasks [14].

Neural networks can be split in different categories, according to the connections between layers:

- In **feed forward nets** the connections between units do not form a directed cycle.
- **Recurrent neural nets** are characterized by forming a cycle in the connections between neurons. This allows them to process sequences of inputs, by using their internal state, which can be viewed as form of memory.

2.3 Types of learning

Discriminative learning

The aim of discriminative learning is to find a map from the input data to labels. The labels can be discrete (classification) or continuous (regression).

When solving a discrimination problem, the neural network is given a labelled dataset, of the form (x, y) , where x is the data instance (represented as a vector of real numbers) and y is the label. The aim of the neural network is to learn a *target function* f such that $f(x) = y$ and be able to predict the value of the function for unseen data.

Example 2.1.

Given data: $((1,4),5)$, $((4,5),9)$, $((20,11), 31)$, $((100,19), 119)$ the network tries to learn a function for which

$$\begin{aligned} f(1,4) &= 5 \\ f(4,5) &= 9 \\ f(20,11) &= 31 \\ f(100,19) &= 119 \end{aligned}$$

There is no guarantee that the network learns a function that will fit the input data perfectly. In fact, this is highly improbable for a big dataset. Moreover, it is undesirable for generalization (see section 2.6 on overfitting).

After learning we can give the network the input $(34, 56)$ and see what output it produces. If it has learned the function $f(x) = x + y$ then the output should be 90.

In a more general setting, discriminative models aim to compute the conditional probability of a label (y) given a data instance (x): $p(y|x)$.

Generative learning

The aim of generative learning is to compute a probability distribution that is very likely to have generated the data. Unlike discriminative models, generative models can also be used in a unsupervised setting, in which there are no labels given for the data, making them suitable for applications like clustering and density estimation (chapters 9 and 2 from [15]). Generative models can be applied for classification and regression, as they can model the joint probabilities between the data x and the labels y , $p(x, y)$, by first computing the marginal probability of the data given a label: $p(x|y)$ (done via Bayes' rule).

Traditionally, neural networks have been used as discriminative models. However, the discovery of Restricted Boltzmann Machines and later on, Deep belief nets has shown that ANNs can be successfully used as generative models.

For a detailed description of generative and discriminative models, as well as a detailed account of their advantages and disadvantages, refer to chapter 1.5.3 in [15]

2.4 Discriminative learning in neural networks

This section will describe how neural networks can be trained for discrimination, by describing backpropagation, a powerful algorithm that has been used successfully for more than 30 years and is applicable to deep belief networks.

2.4.1 The error function

An important step in the learning process is correcting the network when it makes mistakes. We need to be able to show the network what it needs to learn and by how much it is mistaken, so that it can adjust its current beliefs of the target function. For that a measure of the error is required. The choice of error function is application dependent, and has many consequences in the learning process. When presenting a data instance to the network, we can compare the output it produces (\mathbf{y}) with the target output (\mathbf{t}). A common choice is to use the square of the L_2 norm between the two vectors:

$$E(\mathbf{t}, \mathbf{y}) = \|\mathbf{y} - \mathbf{t}\|^2 \quad (2.2)$$

When computing the error on the entire dataset it is customary to use the mean square error, which is the average the error on each individual training cases:

$$MSE = \frac{1}{N} \sum_{i=1}^N \|\mathbf{y}_i - \mathbf{t}_i\|^2 \quad (2.3)$$

The unit of the mean squared error is not the same as the data, but square unit, and in order to moderate that it is common to also use the root mean square error:

$$RMSE = \sqrt{MSE} \quad (2.4)$$

2.4.2 The backpropagation algorithm

The backpropagation algorithm uses the derivatives of the error function with respect to the weight matrix and the bias term to find a set of parameters that minimizes the value of the error function. As the name suggests, the algorithm backpropagates the derivatives from the output layer to the layers below, one layer at a time. After computing the partial derivatives, a first order or second order optimization method is employed to find values of the parameters that minimize the error function. The most common optimization technique used is *gradient descent*, but other algorithms such as *conjugate gradient* have been proven successful [16].

Algorithm 1 Backpropagation learning algorithm

```

Initialize the weights with random values between 0 and 1
while not done training do
  for d in data do
    FORWARDS PASS
    Starting from the input layer, use eq. 2.1 to do a forward pass through the network,
    computing the activities of the neurons at each layer.
    BACKWARDS PASS
    Compute the gradient of the error with respect to the output layer activities
    for layer in layers do
      Compute the gradient w.r.t. the linear input of the neurons in the layer above
      Compute the gradient w.r.t. the parameters of the current layer
      Compute the gradient w.r.t. the activities of the neurons in the current layer
    Update the parameters.

```

Historically, backpropagation was not used successfully on deep nets, due a problem called *vanishing gradient*. What generally tends to happen is that the propagated gradient is smaller than the gradient in the layer above, and by the time the gradient of the first layer is computed, it is so small that the change of the parameters is not significant. This problem especially affects recurrent neural networks, which can be seen as feed forward networks with infinitely many layers. Backpropagation does not guarantee to return the parameters that define a global minimum of the error function, but just a local minimum. It could easily be that there is a set of parameters that perform better but they are in a different “valley” of the function, and the optimization algorithm did not find them (this limitation comes from the optimization algorithm, not from the backpropagation itself).

For a more comprehensive insight on backpropagation see [16].

2.4.3 Computing the derivatives in backpropagation

We will now describe the mathematical derivation behind the backpropagation algorithm. This is to give an insight to the reader as well as to define a framework used to justify later decisions.

We want to be able to compute the derivatives of the error function with respect to the parameters: the weights and biases of the network. We will use an induction-like argument to prove that we can backpropagate the derivatives of the error function from the output layer to the first layer. The derivatives needed in order to perform the recursion argument are the derivatives with respect to the activation of the neurons of the network (y). The argument will go as follows:

If we assume that we know the derivatives of the error function with respect to the activation of the units in the layer above, we can compute the derivatives of the error function with respect to the parameters and with respect to the activations of the current layer. Since we know the derivatives of the error function with respect to the activations in the output layer, we can backpropagate them until we reach the first layer.

We have to prove that if we know the derivatives for a layer, $l + 1$, we can compute them for the layer below, l . Denote by w the weight matrix between the two layers, and by b the bias vector for the units at layer $l + 1$ and assume that the forward pass through the network has been performed, so we know the linear input of each neuron, as well as the activation. We used superscripts to denote the layer a unit belongs to and subscripts to specify the unit inside a layer. Figure A.1 shows the notation used.

Assumption: $\frac{\partial E}{\partial y_j^{(l+1)}}$ is known.

Let the linear input of a neuron to be z :

$$z_i^{(l+1)} = \sum_j w_{ij} y_j^{(l)} + b_i \quad (2.5)$$

$$y_i^{(l+1)} = \sigma(z_i^{(l+1)}) \quad (2.6)$$

By the chain rule we can compute the error derivatives with respect to the weight and the biases in between the two layers.

For the weights:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial w_{ij}} = \frac{\partial E}{\partial z_i^{(l+1)}} \cdot y_j^{(l)} \quad (2.7)$$

For the bias term:

$$\frac{\partial E}{\partial b_i} = \frac{\partial E}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i}{\partial b_i} = \frac{\partial E}{\partial z_i^{(l+1)}} \quad (2.8)$$

Notice that in order to compute the derivatives with respect to the parameters, we need the derivative with respect to the linear input z of the neurons in the layer above. Again applying the chain rule:

$$\frac{\partial E}{\partial z_i^{(l+1)}} = \frac{\partial E}{\partial y_i^{(l+1)}} \cdot \frac{\partial y_i^{(l+1)}}{\partial z_i^{(l+1)}} = \frac{\partial E}{\partial y_i^{(l+1)}} \cdot \frac{\partial \sigma(z_i^{(l+1)})}{\partial z_i^{(l+1)}} \quad (2.9)$$

We know the first term, by the assumption we made. The second term in equation 2.9 can be easily calculated by using the derivative of the activation function.

The only step left is to compute the derivative of the error function with respect to the activations of the neurons at layer l , $y_j^{(l)}$ (so that it can be used for the layer below). With the chain rule (the sum is required because each of the linear inputs at layer $l + 1$ depends on the state of neuron j in the previous layer, $y_j^{(l)}$, as shown by equation 2.10):

$$\frac{\partial E}{\partial y_j^{(l)}} = \sum_{i=1}^K \frac{\partial E}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial y_j^{(l)}} = \sum_{i=1}^K \frac{\partial E}{\partial z_i^{(l+1)}} \cdot w_{ij} \quad (2.10)$$

This completes the proof. Our derivations show that we need to be know how to compute the derivative of the activations with respect to the linear input of the neuron and the derivatives of the error function with respect to the activations of the neurons in the output layer.

Example 2.2 (Sigmoid: computing $\frac{\partial \sigma(z_i^{(l+1)})}{\partial z_i^{(l+1)}}$).

If the activation function used is the logistic sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.11)$$

$$\frac{\partial \sigma(z_i^{(l+1)})}{\partial z_i^{(l+1)}} = \sigma(z_i^{(l+1)}) (1 - \sigma(z_i^{(l+1)})) \quad (2.12)$$

Example 2.3 (The mean square norm: computing the error for the last layer).

In order to be able to start backpropagation we have to compute the derivative of the error function with respect to the activations of last layer of neurons. If the error function used is the square of the mean square error between the output layer activities and the vector label:

$$\frac{\partial E}{\partial y_j} = -\frac{2}{N} \sum_{d \in \text{data}} (t_j^d - y_j^{(d)}) \quad (2.13)$$

2.4.4 Using the error derivative

Once we know how to compute the error derivatives with respect to the weights of the network, multiple issues arise:

- a When should we update the weights
- b How and by how much should we update the weights

a. When should we update the weights: types of learning

When training a neural network, multiple passes through the data are required. Each pass is called a training **epoch**. Inside an epoch, the parameters can be updated at different frequencies:

Online

Correct the model after **each training case**. As the error function changes according to each data instance, the gradients with respect to the parameters can highly fluctuate between updates. Overall, this can result in less stable learning.

Full-batch

Use the **entire dataset** to compute the error and then perform the weight updates by using the sum of the gradients obtained from the individual cases. This method is wasteful, as when we start we have bad parameters and to improve them we have to go through the entire dataset multiple times.

Mini-batch

A better way is combine the above two approaches and to run only **a part of the training cases**, making the parameters reasonable before continuing with more data. As the updates are averaged over multiple cases, the parameters oscillate less than they would do with online learning. When using mini batch learning it is important to ensure that a mini batch has an equal number of instances of each class the model is trying to learn. This is to avoid high fluctuations between updates in different mini-batches: these updates will end up cancelling each other without any benefit for the training process.

b. How to update the weights: optimisation algorithms

The backpropagation algorithm describes a way to compute the derivatives of the error function with respect to the weights of the network. These derivatives can then be used in conjunction with various optimization algorithms.

It is common to use the *gradient descent* algorithm, that updates the weights in the direction of the negative of the gradient.

ϵ is called the learning rate.

Algorithm 2 Gradient descent(ϵ , $threshold$)

```

while  $|x_n - x_{n-1}| < threshold$  do
     $x_{n+1} = x_n - \epsilon \cdot \nabla f(x_n)$ 

```

2.4.5 Parameters and techniques: how to use the gradients**Learning rate**

Various experiments [17] have shown that the learning rate is a crucial parameter that influences the convergence of training.

Setting the learning rate of the model can be done in multiple ways:

- Try out values in the set $10^{-1}, 10^{-2}, \dots, 10^{-5}$ and perform cross validation. Choose the value that yields best results and keep it constant throughout training.
- During learning, monitor the error on a validation set. If the error is steadily decreasing, then increase the learning rate by a constant factor. If the error is increasing, then decrease the learning rate. Towards the end of training, it is best to decrease the learning rate. You can do that once the error stops decreasing steadily. This removes the fluctuations in the values of the weights between mini-batches, and helps towards keeping a steady set of weights for the final ones. In this model, it is also possible to keep a different learning rate for each of the weights, and adjust it during training.

Momentum

The momentum method is a widely used technique that can improve the speed of learning. It is fairly general and can be used with mini-batch and full batch learning, and combined with some of the techniques explained later on, such as Rprop and Rmsprop.

The idea behind the momentum method is to take into account the previous values of the weight gradients when computing the current update. This ensures that the gradient keeps going into the direction it was going previously, speeding up learning. If the current gradient and the previous gradient agree on the direction the weight should move to, a bigger step is performed in that direction. In order to avoid huge weights, it is generally a good practice to decrease the learning rate when using momentum.

$$\nabla(\theta, t+1) = \mu \cdot \nabla(\theta, t) - \epsilon \cdot \frac{\partial E}{\partial \theta} \quad (2.14)$$

Equation 2.14 describes the parameter updates defined by this method, where ϵ is the learning rate and μ is a new parameter, called *momentum*.

At the beginning of learning, the parameters (θ) are quite bad so it is obvious how to change them, hence the gradients will be quite big. This is why it is best to start with a low value for momentum (about 0.5). After a short amount of time, the parameters become sensible and smaller changes are required, in order to settle on the local minimum for the “valley” the weights are already in. That is why it is best to increase momentum (to values up to 0.99) in order to keep going in the right direction.

It is also common to multiply the learning rate by $(1 - \text{momentum})$ to achieve the effect of a decreasing learning rate, due to the fact that momentum gets increased during training:

$$\nabla(\theta, t+1) = \mu \cdot \nabla(\theta, t) - (1 - \mu) \cdot \epsilon \cdot \frac{\partial E}{\partial \theta} \quad (2.15)$$

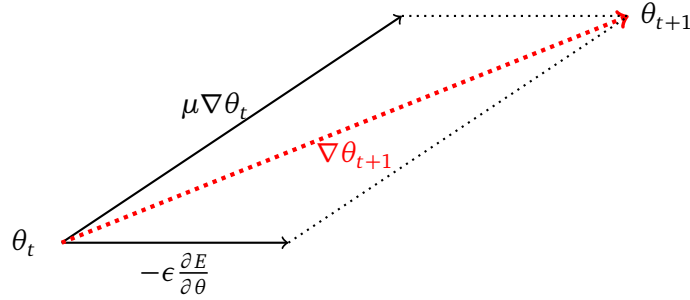


Figure 2.2: Parameter updates according to classical momentum.

Nesterov method for momentum

In the momentum method, we compute the gradient of the error function by doing a forward pass and then doing the cumulative parameter update (formed from the old update and the new gradient). In the Nesterov momentum method we first update the parameters according to the direction of the old update, after which we do a forward and a backward pass to compute the gradients, and then update (again) the parameters using the new computed gradients. It has been shown [18] that the Nesterov momentum can increase the performance of neural networks and tends to perform better than classic momentum. The parameter updates can be described as in equation 2.16.

$$\nabla(\theta, t+1) = \mu \cdot \nabla(\theta, t) - \epsilon \cdot \frac{\partial E}{\partial(\theta + \mu \cdot \nabla(\theta, t))} \quad (2.16)$$

Rprop

Due to variance in the magnitudes of the gradients for different layers and different weights, it is hard to choose one global learning rate that can adapt to all the possible values of the gradients. An idea is to not use a global learning rate, but one for each weight, without taking into account the *gradient* of the error, but just the sign. The idea is simple:

- If the signs of the last 2 gradients for the weight update agree, then multiply the learning rate for this weight by a constant factor (around 1.2)
- Otherwise decrease the learning rate for the weights by a factor. (around 0.5). It is best to keep this factor further away from 1 than the increasing factor, to ensure that the weights do not grow too much.

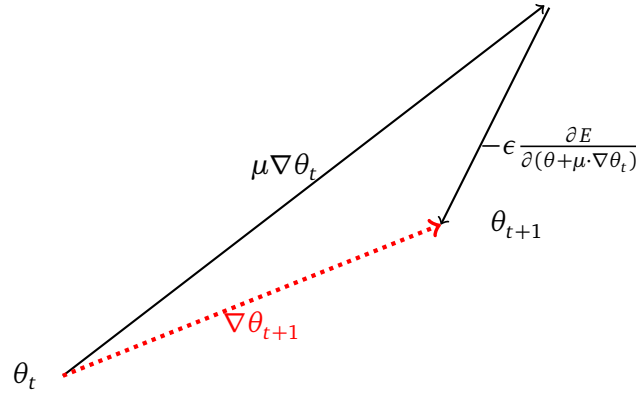


Figure 2.3: Parameter updates according to Nesterov momentum.

- Set the delta for each connection by multiplying the obtained learning rate with the respective gradient. When using Rprop it is useful to set upper and lower limits for the learning rates: too low learning rates do not improve the weights, and too high learning rates can lead to very big weights.

Rmsprop

Rmsprop is similar to Rprop, but for mini-batch learning. Rprop does not work well with mini-batches, and it is inherently a full batch learning method. The idea behind mini-batches is that the weight updates that come from mini-batches get combined in time leading to a suitable average learning updates. Consider for example a set of 10 mini-batches such that the gradients for a particular connections are 9 mini-batches give a gradient for a particular connection to be 0.2 and one mini-batch with gradient -1.8. We would expect that after learning the 10 batches the weights are pretty much unchanged. However, when we use Rprop this is not what happens: in this example, the weights explode, as we have 8 consecutive updates where the gradient keeps the same sign, so we multiply them 8 times by a factor bigger than 1. Hence, the new weights are $(1.2)^8$ times bigger than when we started. When we apply the negative gradient, the weights decrease by a factor of 0.5, making them $(1.2)^8 \cdot 0.5 = 2.15$ bigger than they initially were.

Rprop is equivalent to making an update using the gradient divided by its magnitude, so we divide by a different number for each mini-batch (namely, the gradient). In order to fix this, we ensure that we divide by similar numbers in adjacent mini-batches, by keeping a moving average of the square gradient:

$$\text{MeanSquare}(w, t + 1) = 0.9 \cdot \text{MeanSquare}(w, t) + 0.1 \left(\frac{\partial E}{\partial w} \right)^2 \quad (2.17)$$

$$\nabla(w, t + 1) = \frac{\partial E}{\partial w} \cdot \frac{1}{\sqrt{\text{MeanSquare}(w, t + 1)}} \quad (2.18)$$

Rmsprop can be used using both momentum and Nesterov momentum, even though simple momentum does not help as much as it usually does [19].

For momentum the equations for the updates become as follows¹:

$$\text{MeanSquare}(w, t + 1) = 0.9 \cdot \text{MeanSquare}(w, t) + 0.1 \left(\frac{\partial E}{\partial w} \right)^2 \quad (2.19)$$

$$\nabla(w, t + 1) = \mu \cdot \nabla(w, t) - \epsilon \cdot \frac{\partial E}{\partial w} \cdot \frac{1}{\sqrt{\text{MeanSquare}(w, t + 1)}} \quad (2.20)$$

For Nesterov momentum it has been shown that it is best to use the root of the mean square error to divide the correction made to the previous direction (using the gradient) rather than dividing the previous direction itself [19], leading to the following updates:

$$\nabla(w, t + \frac{1}{2}) = \mu \cdot \nabla(w, t) \quad (2.21)$$

$$\text{MeanSquare}(w, t + 1) = 0.9 \cdot \text{MeanSquare}(w, t) + 0.1 \left(\frac{\partial E}{\partial w} \right)^2 \quad (2.22)$$

$$\nabla(w, t + 1) = -\epsilon \cdot \frac{\partial E}{\partial w} \cdot \frac{1}{\sqrt{\text{MeanSquare}(w, t + 1)}} \quad (2.23)$$

The basic version of Rmsprop does not require adaptive learning rates, but adaptive rates can be used with it. A second order method similar to Rmsprop that is hyperparameter free is described in [20].

2.5 Softmax groups

Minimizing the mean square error of a dataset is equivalent to minimizing ϵ under the assumption that each of labels was drawn from a Gaussian distribution with mean at the true value function for the data point and variance ϵ (i.e. $y \sim \mathbf{N}(f(x), \epsilon)$). While this can be a suitable assumption for regression, for classification the output labels are discrete, so they are far from being Gaussian. This makes the mean square error function less suitable for classification problems.

Often when having to solve a classification task, it is common to have the network output class probabilities, instead of a concrete label. For learning, the labels are transformed into the basis unit vectors that span \mathbb{R}^K , where K is the total number of classes.

Example 2.4 (Why you should not use logistic units with the L_2 norm measure for classification).

Consider the case when a logistic neuron assigns to the correct label the probability 0.000001 (when it

¹As before, ϵ denotes the learning rate and μ the momentum.

should actually attribute 1). The neuron is very far off from the correct result, but as we will show now, there is almost no gradient to allow the logistic unit to change in the use of backpropagation.

Let E be defined as in equation 2.2.

We now compute the error derivative with respect to the last layer of the network:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j} y_i \quad (2.24)$$

Using the logistic function derivative:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) \quad (2.25)$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} = \frac{\partial E}{\partial y_j} y_j(1 - y_j) \quad (2.26)$$

Since E is the square of the L_2 norm:

$$\frac{\partial E}{\partial y_j} = -2(t_j - y_j) \quad (2.27)$$

$$\frac{\partial E}{\partial z_j} = -2(t_j - y_j)y_j(1 - y_j) \quad (2.28)$$

When $t_j = 1$ and $y_j = 0.000001$ then:

$$\frac{\partial E}{\partial z_j} = -2(1 - 0.000001) \cdot 0.000001 \cdot (1 - 0.000001) = 1.999996 \cdot 10^{-6} \quad (2.29)$$

Hence, when we want to propagate the error at the layer below, we get (by using 2.24):

$$\frac{\partial E}{\partial w_{i,j}} = 1.999996 \cdot 10^{-6} \cdot y_i \quad (2.30)$$

Considering that to obtain the weight update the derivative has to be multiplied with the learning rate (which is usually between 10^{-1} and 10^{-2}), the impact on the weights will be negligible, which is the opposite of what should happen, given that the network is completely wrong in its prediction.

We want the output layer to represent a probability distribution. Forcing the activities in the output layer to be between (0,1) can be done using the logistic function, but there is no guarantee that the probabilities add up to 1. The simple and elegant way to solve the above problems is by using a softmax group. In order to ensure that the sum of the output of the unit is 1, the output of a unit does not only depend solely on its input but also on the input of the other elements of the unit.

In the following, the input of one of the units (also called “the logit”) is denoted by z_i and the output of the unit is denoted by y_i .

Definition 2.1 (Softmax function).

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}} \quad (2.31)$$

The derivative of the softmax function is very similar to the one for the logistic function:

$$\frac{\partial y_i}{\partial z_j} = \begin{cases} y_i(1 - y_i) & i = j \\ -y_i y_j & i \neq j \end{cases} \quad (2.32)$$

The best error function to use for a softmax unit is the cross entropy cost function [21]:

Definition 2.2 (Cross entropy).

$$C = - \sum_j t_j \log y_j \quad (2.33)$$

where t_j is the target value of the unit.

Minimizing the entropy is equivalent to maximizing the log probability of the right answer, since for the labelled data t_j will be 1 only for the class corresponding to the training instance (we assume non-overlapping classes).

A softmax function has multiple properties which make it suitable for its use in machine learning:

- The units of a softmax group always form a discrete probability distribution
- A softmax group can be used to model the posterior probability distributions, thus making it suitable as a classification tool
- Any discrete probability distribution can be represented using a softmax unit.
- Used in conjunction with the cross entropy error function, has the property that it propagates big gradients when the correct answer is 1 and the network outputs a very small value (close to lower bound 0), making it is suitable for backpropagation.

We now revisit example 2.4 on a network with a softmax group and a cross entropy error function. Let $t_i = 1$ (hence $t_j = 0, \forall j \neq i$) and $y_i = 0.000001$

$$C = - \sum_j t_j \log y_j \quad (2.34)$$

$$\frac{\partial C}{\partial y_j} = -t_j \frac{1}{y_j} \quad (2.35)$$

Using the multi variate chain rule:

$$\frac{\partial C}{\partial z_k} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial z_k} = - \sum_j \frac{t_j}{y_j} \frac{\partial y_j}{\partial z_k} = - \frac{1}{y_i} \frac{\partial y_i}{\partial z_k} \quad (2.36)$$

By using the derivative of the softmax unit (equation 2.32):

$$\begin{aligned} \frac{\partial C}{\partial z_k} &= \begin{cases} -\frac{1}{y_i} y_i (1 - y_i) & k = i \\ -\frac{1}{y_i} (-y_i^2) & k \neq i \end{cases} \\ &= \begin{cases} y_i - 1 & k = i \\ y_i & k \neq i \end{cases} \end{aligned} \quad (2.37)$$

The weights corresponding to the unit which made the error will get penalized with the difference between the exact output and the target value. In this example the propagated error is $0.000001 - 1 = -0.999999$, which will have considerably more impact than $1.999996 \cdot 10^{-6}$, obtained in example 2.4 which used the squared L_2 norm on the same numbers.

2.6 Overfitting

The aim of a discriminative neural network is to learn regularities in the mapping from input to output. In the limited amount of data the network sees during training, there will also be accidental regularities, arising from **sampling error**. These accidental regularities can potentially make the network not generalize well to unseen testing data. It is impossible for a network to distinguish between real regularities that we aim to learn and the accidental regularities occurring in the data it sees. This important issue that arises when using machine learning techniques, and it is generally referred to as overfitting.

Example 2.5 (Simple overfitting example).

Assume that we want to teach children by example what a reptile is, and we show them only a lot of snakes, of different sizes and colours. If we then show them a turtle, they will probably not recognize it as a reptile, even though it is one. The children have learned that reptiles are much much longer than wide, and that they have no carapace. They have learned what a snake is, but have associated with it the label “reptile.”

For improving generalization it is best to not try to fit the training set perfectly, as that will guarantee that the model has learned the accidental regularities in the data. Methods that aim to avoid overfitting by imposing a complexity penalty to the model are commonly referred to as *regularization techniques*.

Weight decay

It has been observed [22] that extreme values (very small or very big) for the parameters of a machine learning model are a symptom of overfitting: the model is trying to perfectly learn the regularities of the data. In order to avoid weights increasing too much, a *weight penalty* is imposed. This can be done by either subtracting a fraction of the weights at each update, or by imposing a size constraint on the L_2 norm of the incoming weights of a neuron (this is often called max-norm regularization). When the L_2 norm exceeds the allowed constant, the weight vector is scaled down such that the norm is in range. This type of constraint is particularly useful in conjunction with dropout [23].

Early stopping

The idea behind early stopping is to prevent the network from overfitting by halting training before convergence is achieved. This is done by keeping a validation set on which the error is computed during learning. Once the error stops decreasing on the validation set, stop the training, as the network has started to learn the regularities in the training set. This method is highly used to determine when to stop training a model.

Model averaging

Averaging the prediction of multiple models is better than using one single model. Even though for each individual test case one of the models will perform best, if we have varied models their performance will fluctuate from test case to test case, hence the combining them will be better on average. The trick is to find good models that err on different test cases².

We now prove that on average combining multiple models will result in a better performance than randomly chosen model (out of a set of N available different models), under the assumption that the error function used is the mean square error. Denote by y_i the value obtained using model i with $i \in \{1, 2, \dots, N\}$ and t the correct target value. We use $\langle x \rangle_i$ to denote sample mean of x over a population indexed by i .

$$\bar{y} = \frac{1}{N} \sum_{i=1}^N y_i \quad (2.38)$$

$$\begin{aligned} \langle (t - y_i)^2 \rangle_i &= \langle ((t - \bar{y}) + (\bar{y} - y_i))^2 \rangle_i \\ &= \langle (t - \bar{y})^2 \rangle_i + \langle (\bar{y} - y_i)^2 \rangle_i + 2 \cdot \langle (t - \bar{y}) \cdot (\bar{y} - y_i) \rangle_i \\ &= (t - \bar{y})^2 + \langle (\bar{y} - y_i)^2 \rangle_i + 2 \cdot (t - \bar{y}) \cdot \langle y - y_i \rangle_i \\ &= (t - \bar{y})^2 + \langle (\bar{y} - y_i)^2 \rangle_i \\ &\geq (t - \bar{y})^2 \end{aligned} \quad (2.39)$$

²The idea behind model averaging might remind the reader of a similar concept from finance: portfolio optimization. Instead of investing the entire principal in one asset, it is best spread our investment across multiple assets, to minimize risk and without decreasing the expected return.

Model averaging is also a good choice when using probabilistic models which try to increase the log likelihood of the probability for the correct class label (such as softmax groups in conjunction with the cross entropy error), due to the concavity of log:

$$\log\left(\frac{p_1 + p_2 + \dots + p_n}{n}\right) \geq \frac{\log(p_1) + \log(p_2) + \dots + \log(p_n)}{n} \quad (2.40)$$

Model averaging can be done by combining different types of machine learning techniques (neural nets, Bayesian nets, decision trees) or by combining neural networks with different characteristics:

- different architecture (number of layers and/or neurons per layer)
- different regularization constraints
- different activation functions

The reader might be familiar with the concept of model averaging from **Bayesian inference** (chapter 2 in [15]) in the form of the *posterior predictive distribution*. In a Bayesian setting, we can compute the posterior distribution of seeing a new data point \tilde{x} by marginalizing over all the parameters of the model:

$$p(\tilde{x}|X, \alpha) = \int_{-\infty}^{\infty} p(\tilde{x}|\theta) \cdot p(\theta|X, \alpha) d\theta \quad (2.41)$$

Here θ denotes the parameters of the model, α the hyper parameters and X the training dataset.

This means that the posterior of the probability of the new data point, \tilde{x} , is the expected value of the probability of seeing \tilde{x} given a set of parameters θ under the posterior distribution of θ given the entire dataset and the hyper parameters:

$$p(\tilde{x}|X, \alpha) = \mathbb{E}_{p(\theta|X, \alpha)} p(\tilde{x}|\theta) \quad (2.42)$$

Hence the prediction is the result of a weighted *average* of an infinite(!) number of models, one for each possible value of the parameters θ .

Bagging and boosting

Bagging uses multiple bootstrap datasets to train different classifiers, and at test time averages them in order to obtain a classification result. A bootstrap dataset is obtained by uniformly sampling with repetition from the original dataset.

Boosting is a similar technique in which the bootstrap sets are not obtained by uniform sampling, but rather by increasing the probability of obtaining a sample misclassified by the models trained with the previous bootstrap sets.

A comprehensive comparison between boosting, bagging and Bayesian model averaging is offered in [24], showing that model averaging can substantially outperform bagging and boosting in problems with where data has substantial amount of noise or in multi-class problems.

Generative pre-training

There are multiple advantages to generative pre-training, including reduction of overfitting. Generative pre-training will be discussed at length in this report, as it is one of the key advantages of deep belief networks. The core idea is to learn the structure of the data, without supervision, and then apply discriminative learning algorithms.

2.7 Dropout

Dropout is a technique which limits co-adaptations between neurons to increase the generalization capability of a network. It was introduced by Hinton [25] and has been successfully applied in both vision and speech tasks [14, 25]³.

The aim of dropout is to create features that do not rely on each other for producing a useful output. We want neurons to be effective on their own, each representing a feature in the data space without requiring coalitions with other neurons. This is achieved by randomly *dropping out* a percentage of the units in a layer and using only the remaining ones as input for the next layer. The percentage of (kept) active units is denoted by p and is called **dropout**. The dropout probability p can be made a hyperparameter of the model, found with cross validation. Dropping units from the visible layers significantly improves the performance of a network that uses dropout of hidden units [25]. Most experiments to date show best results when p is set to 0.5 for hidden layers and 0.8 for visible layers, and these values have been widely adopted.

Empirical results show dropout improves considerably when imposing a constraint on the L_2 norm of the incoming weights vector for each neuron [23].

As dropout is a technique used to mainly avoid overfitting, its effects can be seen especially on deep networks. This might make you think that the method is not applicable to simpler models (such as shallow networks) that tend to overfit less. There is a simple counter argument to that: dropout allows you to move from a simple model to a deeper, more powerful one that will better capture the features of the data without overfitting.

The mathematical explanations behind dropout are explained in detail in [26].

2.7.1 Dropout as model averaging

Averaging different machine learning models gives better results than just using one model (see subsection 2.6)⁴. Dropout can be seen as a form of model averaging: during training we define a new neural network for each data instance in the training set by omitting certain features in each of the hidden layers. During test time we combine all the learned networks, as the no units are dropped and all parameters are used.

Applying dropout in a network with one hidden layer and a softmax output unit is equivalent to averaging 2^H models [25], where H is the number of hidden units of the net. Each of the 2^H

³See [23] for comparison in error rates with and without dropout for various known benchmarks.

⁴The winners of the Netflix challenge averaged different models for their success in 2010 [27]

models can be obtained by probabilistically excluding units from the hidden layer. The geometric mean is used to average the outputs of the softmax group in order to provide a final result.

2.7.2 Biological intuition

Dropout also has a biological explanation and can be viewed from an evolutionary perspective: in order for an individual to be fit, its genes have to be well co-adapted together. Under a coarse view one of the outcomes of reproduction involves losing half of the genes of each parent. This means the offspring does not fully benefit from the co-adapted genes of its ancestors. Until recently it was not understood why this process can have a desired outcome in the evolution of species. Recent papers [28] show that in the long term this process leads to individuals that are more robust to change, as their genes learn to be more independent and form smaller co-adaptations that can perform different vital functions.

2.7.3 Forward pass in dropout nets

Given that we have to **drop** some of the units when we do a forward pass to the network, the mathematical equations have to change to reflect that. A comparison between a forward network pass with and without dropout can be seen in figure 2.4.

$$\begin{array}{c|c} \begin{array}{l} z_i = \sum_j w_{ij} y_j + b_i \\ y_i = \sigma(z_i) \end{array} & \begin{array}{l} r_j \sim \text{Bernoulli}(p) \\ z_i = \sum_j w_{ij} (y_j r_j) + b_i \\ y_i = \sigma(z_i) \end{array} \end{array}$$

Figure 2.4: Comparison between a forward pass without dropout (left) and with dropout p (right).

At test time, the classification weights need to be scaled by the dropout constant p . This is due to the fact that approximately p percent of the weights will not be active during training (as their corresponding input will be dropped out during learning, but not during test time). Multiplying the final weights by the dropout factor of the incoming layer ensures that the expected activation of a unit is the same during testing and training time.

$$\begin{array}{c|c} \begin{array}{l} r_j \sim \text{Bernoulli}(p) \\ z_i = \sum_j w_{ij} (y_j r_j) + b_i \\ \mathbb{E}(z_i) = \sum_j (p w_{ij}) y_j + b_i \end{array} & \begin{array}{l} z_i = \sum_j (p w_{ij}) y_j + b_i \\ \mathbb{E}(z_i) = \sum_j (p w_{ij}) y_j + b_i \end{array} \end{array}$$

Figure 2.5: Activation of a unit in a network trained with dropout during training (left) and testing (right).

3 | Deep belief nets

Neural nets have thrived since Hinton has shown [13] that Restricted Boltzmann Machines can be stacked together to form deep belief nets, a generative probabilistic model that can also be used for classification. The aim of this section is to provide an overview of models, techniques and algorithms that are used in the deep belief setting.

3.1 Restricted Boltzmann machines

3.1.1 Getting there: Hopfield networks and Boltzmann machines

A Hopfield net is a recurrent neural network, created from a complete and undirected graph built from binary neurons⁵.

Definition 3.1 (Activation function in a Hopfield net).

$$\sigma(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (3.1)$$

This makes the updates of the Hopfield net have the following form:

$$y_i = \begin{cases} 1 & \sum_{j=1}^N w_{ij}y_j + b_i \geq 0 \\ 0 & \sum_{j=1}^N w_{ij}y_j + b_i < 0 \end{cases} \quad (3.2)$$

Because the network does not have a layered structure, the role of the input layer is taken by the entire network: presenting an instance to the network entails setting the state of the network to that particular input pattern.

When we want to update the network, a random neuron is chosen, and its new value is computed from the states of all the other neurons according to equation 3.2. Updates can be done both synchronously and asynchronously.

The Hopfield network has an energy function associated with it, shown in definition 3.2. The form of the energy function guarantees that at each synchronous update the value of the energy function will decrease or stay the same. Moreover, under repeated updates the network will converge to a local minimum of the energy function. This point is called an attractor: upon further updates, the state of the network will not change.

⁵Neural activations of 1 and -1 are also common, as they simplify the learning rule.

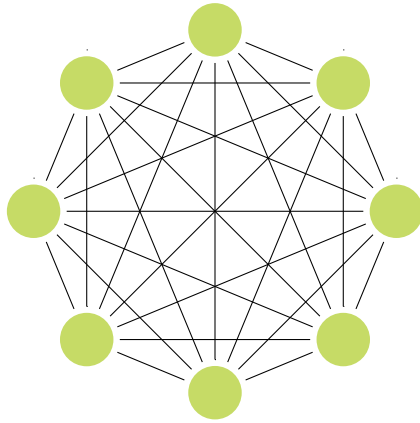


Figure 3.1: Hopfield network with 8 nodes.

Definition 3.2 (Energy of a Hopfield network ⁶).

$$E = - \sum_{i=1}^N \sum_{j < i}^N w_{ij} y_i y_j - \sum_{i=1}^N b_i y_i \quad (3.3)$$

Learning in Hopfield nets is done using a Hebbian rule:

$$\nabla w_{ij} = 4 \left(y_i - \frac{1}{2} \right) \left(y_j - \frac{1}{2} \right) \quad (3.4)$$

Hopfield nets can be understood as a probabilistic storage device: during training, the input data instances become local minima in the energy landscape, making the network likely to converge to one of them. This property can be used to recover a partially known pattern. This property has made them suitable for modelling associative memory [29].

However, during learning spurious minima occur: any linear combination of odd size between input patterns also becomes a local minimum. This is an issue with Hopfield networks, because it imposes a limit on the number of patterns that can be stored in a network of fixed size.

Example 3.1 (Spurious attractors in Hopfield nets).

Assume that we train a Hopfield net of size 5 with the patterns $p_1 = (0, 1, 0, 1, 1)$, $p_2 = (0, 0, 0, 0, 1)$, $p_3 = (1, 0, 1, 0, 0)$. Then $p_1 - p_2 + p_3 = (1, 1, 1, 1, 0)$ also becomes an attractor.

Hopfield nets can be generalized, by making their units stochastic and removing the constraint that the network graph has to be complete. In this form Hopfield networks are known as Boltzmann machines. The idea behind stochastic binary units is avoiding being stuck in a valley produced by spurious attractors. The valleys in the energy landscape created by *false* attractors are not as deep as the ones produced by real attractors (the training instances), so they are easier to escape,

⁶This type of energy functions is borrowed from physics and is generally known as the Ising model.

making it more probable for the network to settle to a real attractor. Due the probabilistic nature of Boltzmann machines, the notion of attractor changes, because updates are non-deterministic. The notion of *thermal equilibrium* replaces the simple notion of attractor. Thermal equilibrium is reached when the probability distribution over the set of states converges and the log probability of each state is linear w.r.t. to its energy. As for Hopfield nets, the energy of the Boltzmann machine can be described using the Ising ferro magnetism model, one in which the energy function is linear in the free variables (in this case the nodes of the network). This makes Boltzmann machines a particular case of Markov Random Fields (chapter 8.3 in [15]).

Definition 3.3 (Stochastic binary units).

A stochastic binary unit is activated (takes value 1) with probability:

$$p(y_i = 1) = \sigma \left(\sum_{j=1}^N w_{ij} y_j + b_i \right) \quad (3.5)$$

3.1.2 Restricted Boltzmann machines

Algorithms for training a general Boltzmann machine exist, but they have serious practical limitations, especially due to the fact that reaching thermal equilibrium for a network with multiple layers takes a large amount of time. The good news is, by restricting the number of layers and the connectivity in a Boltzmann machine we obtain a version easy to train, called *Restricted Boltzmann machine* (RBM).

Definition 3.4 (Restricted Boltzmann machines).

A *Restricted Boltzmann machine* is a neural network with 2 layers of stochastic binary units, with their connections forming an undirected bipartite graph. The layers of the network are called **visible** and **hidden**.

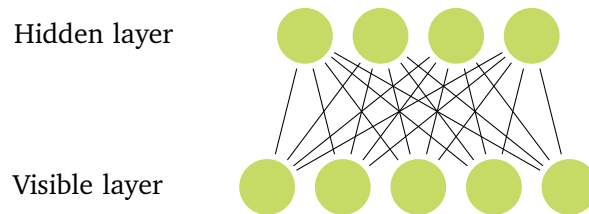


Figure 3.2: Restricted Boltzmann with 5 visible units and 4 hidden units.

RBMs are generative models: the hidden units are latent variables that generate the observable data (the visible units), thus defining a posterior probability distribution on the states of the visible units.

The energy function is the same as the one for a Boltzmann machine, but it is written in a different

form, to emphasise the structure of the network:

$$E(v, h) = - \sum_{i \in \text{visible}} a_i v_i - \sum_{i \in \text{hidden}} b_i h_i - \sum_{i \in \text{visible}} \sum_{j \in \text{hidden}} w_{ij} v_i h_j \quad (3.6)$$

$$= -\mathbf{a}^T \mathbf{v} - \mathbf{b}^T \mathbf{h} - \mathbf{v}^T \mathbf{W} \mathbf{h} \quad (3.7)$$

At equilibrium the network assigns a probability to each possible state of the network, depending on the energy:

$$p(v, h) = \frac{1}{Z} e^{-E(v, h)} \quad (3.8)$$

Definition 3.5 (Partition function).

In equation 3.8 Z is the normalizing constant, called **the partition function**:

$$Z = \sum_{v, h} e^{-E(v, h)} \quad (3.9)$$

The marginal probability of a visible vector is given by:

$$p(v) = \sum_h p(v, h) = \frac{1}{Z} \sum_h e^{-E(v, h)} \quad (3.10)$$

As RBMs are generative models, learning should increase the posterior probability of obtaining the given dataset by sampling the model. The aim of training is to decrease the energy of the data and increase the energy of other configurations of visible units, like in Hopfield nets. By looking at equation 3.10 we notice that increasing the energy of non-data configurations decreases the partition function (the denominator), and decreasing the energy of the data instances increases the nominator, having an overall effect of increasing the likelihood of the training set.

The derivative of the log probability of a data instance can be computed as follows:⁷

$$\frac{\partial \log p(v)}{\partial w_{ij}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \quad (3.11)$$

Equation 3.11 gives an idea for a learning algorithm: using gradient ascent with the following weight updates:

$$\nabla w_{ij} = \epsilon \left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \right) \quad (3.12)$$

⁷For a simple explanation of why the gradient takes this form, see [30]. For a more complete, but also complicated one, see [31, 32]

In order to be able to use equation 3.12 as part of a training algorithm we need:

- An unbiased sample of $\langle v_i h_j \rangle_{data}$
- An unbiased sample of $\langle v_i h_j \rangle_{model}$

Due to the structure of an RBM, *the hidden units are conditionally independent given the value of the visible units*. This property makes it simple to get an unbiased sample from the $\langle v_i h_j \rangle_{data}$ distribution, as the hidden units do not depend on each other given the visible unit (since there are no connection between them):

$$p(h_j = 1|v) = \sigma \left(\sum_{i=1}^N w_{ij} v_i + b_j \right) \quad (3.13)$$

Similarly, the visible units are conditionally independent given the hidden units:

$$p(v_i = 1|h) = \sigma \left(\sum_{j=1}^N w_{ji} h_j + a_i \right) \quad (3.14)$$

The proof of the above equations can be found in appendix B. It shows the tight connection between the activation function (in this case the logistic sigmoid - σ) and the energy function of the network.

This solves the problem of getting an unbiased sample of $\langle v_i h_j \rangle_{data}$. How about $\langle v_i h_j \rangle_{model}$? In theory, this can be obtained by starting with a random data vector and alternating steps of Gibbs sampling between layers until we have attained thermal equilibrium. However, practical issues arise: it is hard to know when thermal equilibrium is reached and to get to it a substantial amount of training time is needed.

Definition 3.6 (Positive phase).

Calculating the unbiased sample of $\langle v_i h_j \rangle_{data}$ is called the positive phase of a algorithm that trains an RBM.

Definition 3.7 (Negative phase).

Calculating an approximation of the unbiased sample of $\langle v_i h_j \rangle_{model}$ is called the negative phase of a algorithm that trains an RBM.

3.1.3 Training an RBM: Contrastive divergence

Contrastive divergence (CD) [32] is a training algorithm for RBMs that uses a simple approximation of $\langle v_i h_j \rangle_{model}$. Contrastive divergence is time efficient and gives good results in practice. It starts with a data vector from the training set and uses a step of Gibbs sampling⁸ to obtain the states of

⁸Gibbs sampling is a Markov Chain Monte Carlo algorithm for obtaining samples from a multi variate probability distribution. Details and a mathematical derivation can be obtained from [33].

the hidden units. From the states of the hidden units, visible units are sampled. The process is repeated multiple times, obtaining *reconstructions* for both the visible and hidden units.

An approximation to the unbiased sample of $\langle v_i h_j \rangle_{model}$ is obtained by using the values of the reconstructions:

$$\nabla w_{ij} = \epsilon \left(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{reconstruction} \right) \quad (3.15)$$

$$\nabla a_i = \epsilon \left(\langle v_i \rangle_{data} - \langle v_i \rangle_{reconstruction} \right) \quad (3.16)$$

$$\nabla b_j = \epsilon \left(\langle h_j \rangle_{data} - \langle h_j \rangle_{reconstruction} \right) \quad (3.17)$$

Positive phase in CD

Fix the data vector on the visible units and sample from the hidden units. Use $v_i h_j$ as an unbiased sample of $\langle v_i h_j \rangle_{data}$.

Negative phase in CD

Starting with the data vector on the visible units, perform alternating steps of Gibbs sampling. Use the reconstruction of visible and hidden states as an approximation for the unbiased sample of $\langle v_i h_j \rangle_{reconstruction}$.

For mini-batch learning, the samples obtained in the positive and negative phase above are averaged on the entire mini-batch before updating parameters.

CD_k denotes the contrastive divergence algorithm with k alternating Gibbs sampling steps performed to obtain the reconstructions. CD_1 , is most commonly used, as it is the most time efficient and gives good enough results.

3.1.4 Persistent Contrastive Divergence

Persistent contrastive divergence [34] is another algorithm used to train an RBM, introduced in 2008 by Tieleman.

Positive phase in PCD

Same as CD.

Negative phase in PCD

Keep a global set of *fantasy particles* - states of the network (pairs of visible and hidden units). For each parameter update, also update the particles by performing multiple steps of Gibbs sampling. Average out $v_i h_j$ over the set of particles to get an approximation for $\langle v_i h_j \rangle_{model}$ ⁹.

⁹We can notice here the difference that for the negative phase PCD has one single Markov chain that is ran throughout training, but CD starts a new chain for each data instance.

Persistent contrastive divergence relies on small weight changes at each update, such that fantasy particles can be considered to reach thermal equilibrium, as they are the result of repeated sampling since the start of training. This has a direct implication: the learning rate used in PCD has to be smaller than when using CD.

PCD tends to give better density models than CD [34], but has the disadvantage of requiring more time for training. It is also more sensitive to the value of the learning rate, hence it involves more tuning to obtain accurate results.

3.1.5 Initialization of parameters

At the beginning of learning, the weights are initialized to values sampled from a Gaussian distribution with mean 0 and standard deviation of about 0.01. The visible biases are initialized to $\log \frac{p_i}{1-p_i}$, where p_i is the percentage of training points that have the visible unit i on. The bias vector for the hidden units is initialized to 0 [35].

3.1.6 Monitoring learning: Free energy, reconstruction error and AIS

It is important to be able to monitor the error of an RBM on a validation set. The error can then be used to stop overfitting and increase the generalization of the model.

Reconstruction error

Computing the mean square error between the dataset and the visible reconstructions can give a rough idea on the progress of learning: it typically decreases rapidly at the beginning of learning and then steadily decreases (with minor fluctuations between mini-batches). Even though the reconstruction error might seem advantageous to use, it is a poor measure of the function CD is approximately optimizing, and should not be trusted [35].

Annealed Importance Sampling (AIS)

The main problem of optimizing the log likelihood of the training data for an RBM is computing the partition function (see definition 3.5). Consider a network binary units with 1000 visible neurons and 500 hidden neurons. The sum in definition 3.5 requires computing the energy for all possible configurations of visible and hidden units, in this case $2^{1000} \times 2^{500} = 2^{1500}$. Moreover, Z would need to be computed for *every update* of the parameters. This is simply infeasible.

AIS [36] combines simulated annealing [37] and importance sampling [38] to approximate properties of a desired intractable distribution by using a chain of intermediate distributions.

Applied to RBMs, AIS is used to approximate the value of the partition function (Z) by introducing a chain of distributions with known likelihoods (denoted here by p_k^*) but unknown partition constant (Z_k).

We define the chain of probability distributions as follows: let p_0, p_1, \dots, p_N such that p_k is very close to p_{k+1} , and $p_k = \frac{p_k^*}{Z_k}, \forall k \in \{0, \dots, N\}$.

By setting $p_0 = p_A$ to the distribution modelled by an RBM and $p_N = p_B$ to a distribution for which the partition function can be calculated, Z_B/Z_A can be computed as follows:

$$\frac{Z_B}{Z_A} = \frac{Z_N}{Z_{N-1}} \frac{Z_{N-1}}{Z_{N-2}} \dots \frac{Z_1}{Z_0} \quad (3.18)$$

Each Z_k/Z_{k-1} is computed using:

$$\frac{Z_k}{Z_{k-1}} = \frac{\int p_k^*(v) dv}{Z_{k-1}} = \int \frac{p_k^*(v)}{p_{k-1}^*(v)} p_{k-1}(v) dv = \left\langle \frac{p_k^*(v)}{p_{k-1}^*(v)} \right\rangle_{p_{k-1}} \quad (3.19)$$

$\left\langle \frac{p_k^*(v)}{p_{k-1}^*(v)} \right\rangle_{p_{k-1}}$ can be approximated for very close distributions by using Markov Chain Monte Carlo methods [17].

AIS also has problems, as it only gives a good approximation of Z_B/Z_A if p_A and p_B are close distributions. However, [17] shows that with AIS you can obtain a reasonably good approximation of the likelihood of the training data on multiple tasks, including the MNIST handwritten digit benchmark [39].

After having an estimate for the partition function, the probability of each network configuration can be easily approximated using equation 3.8, and from there the marginal probability of a data vector can be obtained. Displaying the approximate log probabilities during learning obtained from AIS can be a good indicator of what progress the network is making during learning, but it is considerably more computationally expensive than the reconstruction error, making it less used in practice.

Free energy

A simple and efficient way to check if the network is overfitting is by comparing the free energies of vectors from the training and validation set [35]. If the gap between the two energies is growing (the ratio between the free energy of the validation data and the free energy on the training set is significantly greater than 1), then the model is likely to have started overfitting. Note that the magnitudes of the free energies do not have any meaning, but only the ratio between them (in which the partition function cancels out).

Definition 3.8 (Free energy).

The free energy of a visible vector v is the energy required for a vector in order to have the same probability as all the (visible, hidden) configurations that contain v :

$$e^{-F(v)} = \sum_h e^{-E(v,h)} \quad (3.20)$$

A simple way to compute $F(v)$ [35] is:

$$F(v) = -v^T a - \sum_{j \in \text{hidden}} \log \left(1 + b_j + \sum_{i \in \text{visible}} v_i w_{ij} \right) \quad (3.21)$$

3.1.7 Convergence of training

Since RBMs have become popular tools for learning features of data, interest has risen in training them and in the theoretical guarantees that contrastive divergence can give. CD is a poor approximation of the gradient of the log likelihood of the data, and in fact it has been shown that it does not follow the gradient of any function [40].

Maximizing the log likelihood of the data (equation 3.11) is equivalent to minimizing the Kullback-Liebler divergence between the distribution of the data, P_0 , and the equilibrium distribution defined by the model, P_∞^θ , where θ are the parameters of the model: $KL(P_0||P_\infty^\theta)$. The Kullback-Liebler divergence gives a quantitative way to measure of the difference between two probability distributions:

$$KL(P||Q) = \int_{-\infty}^{\infty} \ln \frac{P(x)}{Q(x)} P(x) dx \quad (3.22)$$

Computing P_∞^θ is computationally expensive (it is intractable for real world problems), so what CD_n does instead is minimizing the difference between two divergence measures:

$$KL(P_0||P_n^\theta) - KL(P_n^\theta||P_\infty^\theta) \quad (3.23)$$

The updates for the RBM parameters (equations 3.15, 3.16 and 3.17) can be derived from equation 3.23¹⁰.

Notice that P_n^θ depends on the current parameters of the model (the weights), and that the weights change at each iteration, hence changing P_n^θ , but contrastive divergence ignores that.

Both of the RBM training algorithms described for (CD and PCD) can diverge from maximizing the log likelihood of the data, with the point of divergence being particularly dependent on the learning rate [17].

3.1.8 Real valued units

When using binary units, the input data has to be scaled to have values in between 0 and 1. This is not a problem for simple dataset, but imposes serious limitations when trying to represent properties present in real valued input. For example, it is hard to represent with binary units that the integer value of a pixel is very close to the average of the 8 pixels around it.

Using real valued visible units can solve this problem, and they can be achieved in multiple ways:

¹⁰The derivations can be found in [32]. The proof relies on that RBMs being at the intersection of 2 classes of probabilistic generative models: Products of Experts [41] and Boltzmann machines.

Gaussian visible units

A solution to replacing stochastic binary units for the visible layer is to use linear units with independent Gaussian noise. The energy function then becomes:

$$E(v, h) = \sum_{i \in \text{visible}} \frac{(v_i - a_i)^2}{2\sigma_i^2} - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i \in \text{visible}} \sum_{j \in \text{hidden}} \frac{v_i}{\sigma_i} h_j w_{ij} \quad (3.24)$$

where σ_i is the standard deviation of the Gaussian visible unit i .

Learning σ_i is possible, but it is difficult using CD. A common approach is to normalize the training data to have zero mean and unit variance on each dimension, and set the variances to 1 in equation 3.24.

When using Gaussian units, the learning rate has to be set smaller than with binary units [35], as there is no upper or lower bound on the values that the units can take, and no bounds on the update at each step in CD / PCD namely: $\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}}$ ¹¹.

Given that the input data has been normalized to have zero mean and unit variance, the reconstruction value of a Gaussian visible unit is given by the input from the hidden units plus its bias (i.e. its activation function is the identity function).

Gaussian visible and hidden units

The hidden units can be made Gaussian in the same way as the visible units. This makes learning substantially more unstable. The energy function adapts and becomes:

$$E(v, h) = \sum_{i \in \text{visible}} \frac{(v_i - a_i)^2}{2\sigma_{\text{vis},i}^2} - \sum_{j \in \text{hidden}} \frac{(b_j - h_j)^2}{2\sigma_{\text{hid},j}^2} - \sum_{i \in \text{visible}} \sum_{j \in \text{hidden}} \frac{v_i}{\sigma_{\text{vis},i}} \frac{h_j}{\sigma_{\text{hid},j}} w_{ij} \quad (3.25)$$

RBMs that have Gaussian visible and hidden units are very hard to train and unstable [35].

Rectified linear units

Requiring the values of the visible units to be roughly integers can be very useful (think of pixel values). Binomial units can be used to model noisy integer values [42], but more recently it has been shown that rectified linear units (ReLU) perform better at the same task [14, 43].

Definition 3.9 (Gaussian rectified linear units).

Noisy rectified linear units are neurons that use activation function:

$$\max(0, x + \mathbf{N}(0, \sigma(x))) \quad (3.26)$$

Training an RBM with rectified linear units does not impose any of the difficulties that using Gaussian units does. For computational reasons, it is common to replace the variance $\sigma(x)$ by a constant 1 [35]. Rectified linear units can also be used for training a neural network in a supervised fashion with backpropagation. This requires a deterministic version of the function, that can also be used for classification [43].

¹¹In the case of binary units, it is bound between -1 and 1.

Definition 3.10 (Deterministic rectified linear units).

Rectified linear units are neurons that use activation function:

$$\max(0, x) \quad (3.27)$$

When used with backpropagation the discontinuity at 0 is being ignored, so the derivative is taken to be:

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \quad (3.28)$$

It is suggested in [35] that if for an RBM both hidden and visible units are rectified linear units then the learning rate required for training might be smaller than the one used for binary stochastic units in order to avoid unstable weight updates (which can be caused by the now unbounded activation of the neurons). In [43] the authors report positive results by using Gaussian visible units and noisy rectified hidden units in when training Restricted Boltzmann machines.

3.1.9 Sparse hidden units

Forcing hidden units to be sparse can improve learning and helps with interpreting what features the hidden units have detected, by ensuring that the network does not learn trivial features (such as a unit learning exactly one pixel) [35]. The hidden units are forced to be sparse by adding a penalty that ensures that the actual probability that a unit is active (q) is close to the sparsity target p :

$$\text{Penalty} = \text{BinaryCrossEntropy}(p, q) = -p \log q - (1 - p) \log(1 - q) \quad (3.29)$$

The probability of being active is computed using a decaying average:

$$q_{\text{new}} = \lambda q_{\text{old}} + (1 - \lambda) q_{\text{current}} \quad (3.30)$$

where q_{current} is the mean activation probability on the current mini-batch and λ is a regularization parameters (usually around 0.9). Apart from the usual update of the parameters required by the training algorithm, the gradient of the penalty term is also used to update the parameters.

Sparse hidden activations can also be achieved using dropout when training a Restricted Boltzmann machine according to the experimental results reported in [23].

3.1.10 Other methods

Weight decay, momentum and Rmsprop apply to Restricted Boltzmann Machines as usual.

In the case of dropout the model changes to also include a binary vector of independent random variables \mathbf{r} . The length of \mathbf{r} is equal to the length of the vector of hidden units, and the value of the

random variable r_j determines if the hidden unit h_j is dropped out from the model: if $r_j = 1$ then h_j is kept in the model, otherwise it is dropped out.

The joint distribution of the RBM then becomes (p is the dropout parameter for the hidden layer, θ the other parameters of the RBM):

$$P(\mathbf{r}, \mathbf{v}, \mathbf{h}|p, \theta) = P(\mathbf{r}|p) \cdot P(\mathbf{v}, \mathbf{h}|\mathbf{r}, \theta) \quad (3.31)$$

$$P(\mathbf{r}|p) = \prod_j p^{r_j} \cdot (1 - p)^{1-r_j} \quad (3.32)$$

$$P(\mathbf{v}, \mathbf{h}|\mathbf{r}, \theta) = \frac{1}{Z'(\theta, \mathbf{r})} e^{-E(\mathbf{v}, \mathbf{h})} \prod_j \delta(h_j, r_j) \quad (3.33)$$

$$\delta(h_j, r_j) = \begin{cases} 0 & r_j = 0 \\ h_j & r_j = 1 \end{cases} \quad (3.34)$$

As usual, $Z'(\theta, \mathbf{r})$ denotes the partition function required for $P(\mathbf{v}, \mathbf{h}|\mathbf{r}, \theta)$ to be a probability distribution.

We can extend this and define the joint distribution to also include a dropout parameter for the visible layer (Z' and σ are defined as before):

$$P(\mathbf{r}_1, \mathbf{r}_2, \mathbf{v}, \mathbf{h}|p, \theta) = P(\mathbf{r}_1|p_1) \cdot P(\mathbf{r}_2|p_2) \cdot P(\mathbf{v}, \mathbf{h}|\mathbf{r}_1, \mathbf{r}_2, \theta) \quad (3.35)$$

$$P(\mathbf{v}, \mathbf{h}|\mathbf{r}_1, \mathbf{r}_2, \theta) = \frac{1}{Z'(\theta, \mathbf{r}_1, \mathbf{r}_2)} e^{-E(\mathbf{v}, \mathbf{h})} \prod_i \delta(v_i, r_{1,i}) \prod_j \delta(h_j, r_{2,j}) \quad (3.36)$$

3.2 Deep belief networks

Discovered by Geoffrey Hinton in 2006 [13], deep belief networks use the principle of greedy layer-wise training to initialise parameters before performing any discriminative or generative fine-tuning.

Like Restricted Boltzmann machines, deep belief networks are probabilistic generative models that use latent variables to learn features from the data. Unlike RBMs, they use multiple layers of hidden units, giving them a more hierarchical structure and allowing them to learn higher level representations (features of features). Hierarchical representations of objects are natural and reflect how humans perceive the world (example 3.2).

Deep belief nets were initially introduced using stochastic binary units, but the extensions of RBMs discussed in section 3.1.8 can be used stacked together to form DBNs.

Example 3.2 (Levels of representation in objects).

Looking at an image we distinguish multiple hierarchical levels:

- 1) the pixel level
- 2) the stroke level
- 3) the edge level
- 4) the object level

3.2.1 Greedy pre-training

We have seen how a Restricted Boltzmann Machine can learn features. In a hierarchical model, we now want to learn the features of these features. We can do this by creating another RBM, for which the input are the first set of learned features of the data (the state of the hidden units of the first RBM, when the input is a data vector). This process can be repeated multiple times, allowing learning of higher and higher layer of features. It can be proved that every time we add another layer, we improve the variational lower bound on the log probability of generating the data [13].

After creating these RBMs, we have to combine them together. The way the RBMs are combined together is not completely obvious: for the top layer RBM, we keep the undirected connections (such that they form an associative memory), but for the lower level ones, we only keep the top-down *generative* weights. We still keep the down-up *recognition* weights, but they are not part of the model. The recognition weights are the transpose of the generative weights, and will be used for inference. Figure 3.5 exemplifies the difference between recognition and generative weights.

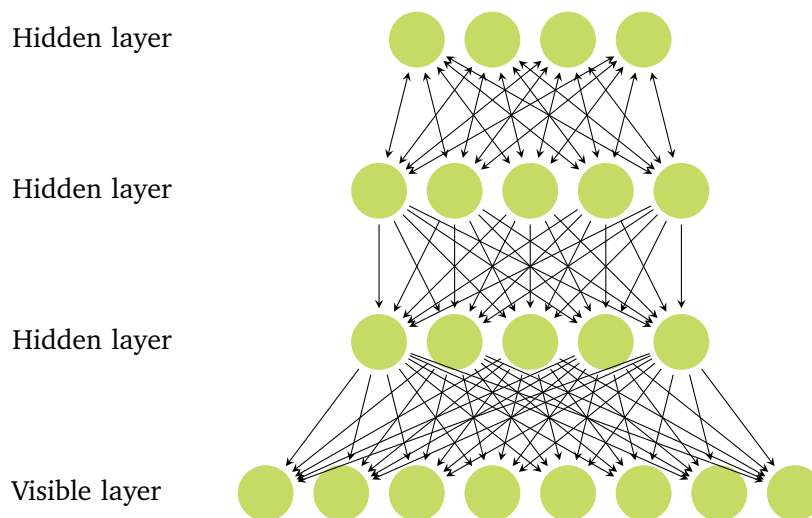


Figure 3.3: A deep belief network as a top down generative model.

The resulting model is a generative one, so we have to ask the question: what distribution is it modelling and how do we generate data from it?

DBNs model the joint distribution between the observable data (\mathbf{v}) and the latent hidden variables (feature vectors \mathbf{h}_k):

$$p(\mathbf{v}, \mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_n) = p(\mathbf{h}_n, \mathbf{h}_{n-1}) \cdot \prod_{k=0}^{n-2} p(\mathbf{h}_k | \mathbf{h}_{k+1}) \quad (3.37)$$

As an interesting observation, we note that DBNs are recursive: removing the visible layer of a deep belief net with more than 3 layers will result in another deep belief network, with one layer less.

Improving greedy pre-training for some network architectures

In order to be able to stack 2 RBMs on top of each other, the number of hidden units of the first RBM has to be equal to the number of visible units of the second RBM. This is required in order to be able to propagate the hidden activations of the first RBM as training data for the second RBM. What if we also know that the number of visible units of the first RBM is equal to the number of hidden units of the second RBM? Figure 3.4 shows such an example. If the RBM model is *symmetric* and the hidden activations of the first RBM are the input for the second one, the two networks are trying to model similar correlations, so the weights learned by the first one can be used to initialize the weights of the second RBM. This type of initialization also has a theoretical justification (see appendix C for the details).

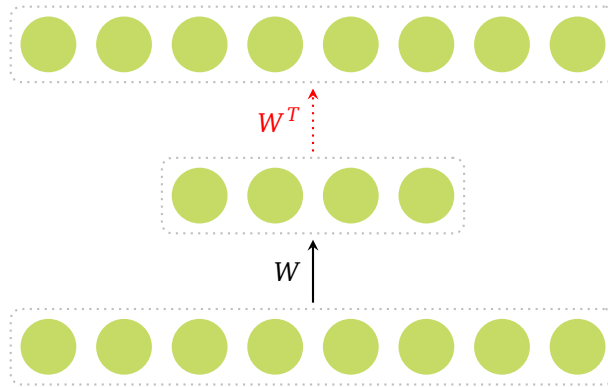


Figure 3.4: Required network architecture for initializing the weights of a network to the transpose of the weights of the network trained before it. The weights in black are the ones *after* training of the corresponding RBM, and the ones in red are *before* training the RBM.

As stated above, this initialization only if the RBMs are symmetric. In their initial formulation, RBMs are symmetric because both hidden and visible units used the same activation function, namely the logistic sigmoid. However, recent work [43] has shown that using Gaussian visible units and noisy rectified linear units can improve performance of RBMs. This type of RBM is not symmetric, due to the different activations functions. Moreover, it is common to scale the input

data to an RBM with Gaussian visible units to have zero mean and unit variance in order to not learn the variance of the visible units using CD. In that case, the input data for the second RBM is not given by the hidden activations of the first RBM, but by *scaled* hidden activations. Hence the second RBM models different correlations, meaning that we should not initialize the weights of the second RBM to the ones resulted from training the first network, even though the shapes of the two networks permits it.

Generating data from a DBN

Because DBNs are generative models, we would like to be able to sample from the distribution they define. This is done as follows:

- 1) Get an equilibrium sample from the top level RBM (by performing alternated Gibbs sampling between the two layers)
- 2) Starting from the hidden nets of the top layer RBM, use the top down generative weights to perform a pass through the network

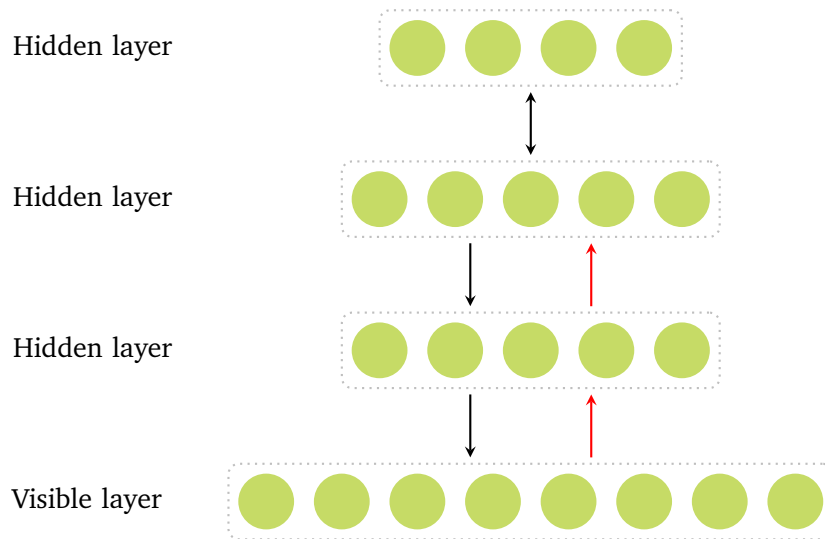


Figure 3.5: Generative versus recognition weights in a DBN. The recognition weights are depicted in red.

3.2.2 Theoretical justification of the greedy learning

We will now give some mathematical intuition behind layer-wise pre-training¹².

Consider a DBN built from 2 stacked Restricted Boltzmann machines. We denote by \mathbf{v} the input dataset, and by \mathbf{h}_1 and \mathbf{h}_2 the activities of the hidden layers which the network produces on a bottom-up pass through the network, when the visible layer of the bottom RBM is set to \mathbf{v} .

¹²A more detailed explanation in appendix C. The explanation here is based on [44].

$Q(\mathbf{h}_1|\mathbf{v})$ represents the conditional distribution imposed by the data onto the hidden units of the first RBM only. $p(\mathbf{h}_1|\mathbf{v})$ denotes the conditional probability between \mathbf{h}_1 and \mathbf{v} when the entire model is considered. H_p is the entropy of a distribution P with probability density function p :

$$H_p = - \int_{-\infty}^{\infty} p(x) \log p(x) dx \quad (3.38)$$

It can be shown that:

$$p(\mathbf{v}) = KL(Q(\mathbf{h}_1|\mathbf{v})||p(\mathbf{h}_1|\mathbf{v})) + H_{Q(\mathbf{h}_1|\mathbf{v})} + \sum_h Q(\mathbf{h}_1|\mathbf{v}) (\log p(\mathbf{h}_1) + \log(\mathbf{v}|\mathbf{h}_1)) \quad (3.39)$$

The probability distributions $Q(\mathbf{h}_1|\mathbf{v})$ and $p(\mathbf{h}_1|\mathbf{v})$ are different in general, but it can be shown that they are equal if we initialize the weight matrix of the second RBM to be the transpose of the weight matrix of the first RBM: $W_2 = W_1^T$.

$$W_2 = W_1^T \implies Q(\mathbf{h}_1|\mathbf{v}) = p(\mathbf{h}_1|\mathbf{v}) \implies KL(Q(\mathbf{h}_1|\mathbf{v})||p(\mathbf{h}_1|\mathbf{v})) = 0 \quad (3.40)$$

Under this assumption, the aim of training the second RBM is to minimize:

$$H_{Q(\mathbf{h}_1|\mathbf{v})} + \sum_h Q(\mathbf{h}_1|\mathbf{v}) (\log p(\mathbf{h}_1) + \log p(\mathbf{v}|\mathbf{h}_1)) \quad (3.41)$$

When isolating the terms that depend only on W_2 we obtain:

$$\sum_h Q(\mathbf{h}_1|\mathbf{v}) \log p(\mathbf{h}_1) \quad (3.42)$$

This is equivalent to maximizing the log likelihood of the distribution modelled by the second RBM, given that its input is given by the output of $Q(\mathbf{h}_1|\mathbf{v})$. But this is exactly what training an RBM does when its input is the output of the first already trained RBM!

3.2.3 Classification using deep belief nets

So far we have discussed deep belief nets as generative models. They can be adapted and used for classification and regression. In order to use deep nets for classification, another layer has to be added on top of the network. Usually this layer is with a softmax (subsection 2.5). Another option is to feed the last layer values as input into another classifier, such as a Support Vector Machine[11].

To train the network, we firstly perform the greedy pre-training, learning one layer of features at a time. Afterwards, we apply backpropagation to the entire network, in order to learn how to discriminate between class labels.

This approach eliminates a lot of the problems usually encountered with backpropagation:

- Backpropagation **does not have to learn the features of the data**. The task has been taken over by the greedy pre-training. This solves the problem of the vanishing gradient: the main aim of backpropagation is to learn the weights of the top (discriminative) layer, as the weights of the first layers already have sensitive values. If the gradient is too small to affect the first layers, the impact on learning is not as drastic.
- The algorithm is **less likely to get stuck in a bad local minimum** of the energy function, due to the sensible initialization of weights.
- **Less labelled data is needed**. The greedy pre-training does not require labelled data, as it is inherently unsupervised. Labelled data is a scarce resource, as obtaining it involves manual work. Requiring less labelled data is a plus for any algorithm, as it can be given as input bigger datasets.
- Greedy pre-training causes **less overfitting** than just using standard backpropagation, as a lot more information is obtained from the input data (namely the higher level features which are learned in the first phase of training).

3.2.4 Better generative models: Contrastive wake sleep

The wake-sleep algorithm described by Hinton in [45] can be adapted to DBN, allowing layers to influence each other, after greedy pre-training. The aim of this is to make the network better at data generation.

For this algorithm we start differentiating between the *generative* and *recognition* weights of the network.

The main steps of the algorithm are:

1. Use the recognition weights to do a stochastic bottom-up pass. From the layer activities obtained, adjust the generative weights.
2. Do a few iterations of sampling in the top level RBM and adjust its weights using contrastive divergence.
3. Use the generative weights to do a top-down pass and use the activities to adjust the reconstruction weights.

The updates in step 2) are done as explained in subsection 3.1.3.

For steps 1) and 3), the weights between any two layers are adjusted as in the standard wake sleep algorithm:

$$\nabla w_{ij} = \epsilon s_i(s_j - p_j) \quad (3.43)$$

where by s we denote the state of a neuron, and p is the activation probability of a neuron.

3.2.5 Improving training in deep belief nets

The methods discussed in section 2 also apply to deep belief nets. Most of them remain unchanged, as in the case of Restricted Boltzmann machines. However, when dropout is applied to pre-trained networks, the initialization of the weights of a layer in the DBN changes: instead of taking the exact value of the weights of the corresponding RBM (as usual with pre-training), it divides the weights by the dropout constant that will be used when fine-tuning this layer. This is to counter balance the fact that for testing the weights are multiplied by the dropout: the initial value of the weights should be of the same scale as the test weights [23].

4 | Putting it all together: our model

One of the aims of this project is to provide a comprehensive, *modular* and up to date implementation of Restricted Boltzmann Machines and Deep Belief Nets. This section describes the model we used.

We employed the techniques discussed in sections 2 and 3 and were able to compare their performance.

4.1 Restricted Boltzmann Machine

The training algorithm used for RBMs is Contrastive Divergence. We did implement Persistent Contrastive divergence but our results confirm that it less stable, as it is more sensitive to the parameters used for training.

4.1.1 Techniques

The implementation of Restricted Boltzmann machine applies the following methods:

- Mini batch learning
- Momentum / Nesterov momentum
- Dropout
- Rmsprop
- Weight decay
- Sparsity constraints

The training of an RBM is monitored using the reconstruction error on each batch. Despite the problems described in subsection 3.1.6, it is a useful guideline that can show when training goes wrong.

4.1.2 Sparsity constraints

Binary cross entropy and activation probability

Our model of Restricted Boltzmann machines allows setting a sparsity constraint, as described in subsection 3.1.9. In order to be able to use the sparsity cost, we need to calculate the activation probability of a unit. For binary hidden units it is simple (they are binomial random variables). Rectified units have been shown to be naturally sparse [43], but their sparsity could be improved and ensured via a sparsity penalty during training.

We have defined the sparsity penalty using the binary cross entropy cost (p is the desired activation probability and q is the running mean average of the activation probability for the neuron) :

$$q = \lambda q_{old} + (1 - \lambda) q_{current} \quad (4.1)$$

$$Cost = -p \log q - (1 - p) \log(1 - q) \quad (4.2)$$

$q_{current}$ is computed by averaging the probability of a hidden unit to be strictly active over a mini-batch.

In order to apply this penalty for an RBM with rectified noisy hidden units, we have to define the probability of a rectified linear unit to be active. We choose this to be the probability that the rectified linear unit is strictly positive:

$$P(\max(0, N(x, \sigma(x))) > 0) = P(N(x, \sigma(x)) > 0) = 1 - \text{GaussCDF}(0|x, \sigma(x)) \quad (4.3)$$

Mean square error and the expected value

Another alternative for a sparsity cost is similar to the one defined in [46]: the square error between the desired activation and the expected value of the activation, averaged over the instances in the mini batch. We extended it to use the running average over multiple mini-batches ¹³.

$$q = \lambda q_{old} + (1 - \lambda) \frac{1}{m} \sum_{l=1}^m \mathbb{E}(h_j^l | \mathbf{v}^l) \quad (4.4)$$

$$Cost = (p - q)^2 \quad (4.5)$$

Using this cost requires knowing the expected value of the hidden unit given the input from the visible layer. Appendix D shows the mathematical derivations for computing the expected value of a hidden unit given the value of the visible unit

$$\mathbb{E}(h_j | \mathbf{v}) = \frac{\sqrt{\sigma(x)}}{\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma(x)}} + x \text{GaussCDF}\left(\frac{x}{\sqrt{\sigma(x)}} \middle| 0, 1\right) \quad (4.6)$$

where x is the linear input received by unit h_j from the layer of visible units \mathbf{v} : $\sum_{i=1}^N w_{ij} v_i + b_j$.

The expected value of a rectified linear unit has been helpful also when stacking RBMs for creating deep belief nets. After training the first RBM, the activations of the hidden units are fed as input into the next RBM. In theory, the sampled activations should be set as input. However, practical experiments have shown that it is better to use a *deterministic* propagation (expected activations)

¹³This also allows us to apply the gradient to the weights of the RBM, not only to the hidden bias, as described in [46].

from the previous RBMs¹⁴. Defining the expected value for noisy rectified linear units allows us to use this heuristic when initializing a DBN after the RBMs have been trained with this type of unit.

Recipe for using sparsity constraints

We have mathematically defined how to use sparsity constraints for rectified linear units. But how to use them in practice? Our recipe comes from monitoring learning: start with no sparsity constraints. Monitor the average sparsity after training (or during each mini batch). If it is bigger than the desired sparsity, use the constraint. When using RBMs for creating a deep belief net, monitor the sparsity for each of them, and make an individual decision for each of the RBMs, depending on the average obtained sparsity.

4.2 Deep belief nets

Pre-training in deep belief networks is performed as explained in subsection 3.2.1: RBMs are trained sequentially, and if we use stochastic binary RBMs and their shapes permit it, the weights of the RBMs on top are initialized to the transpose of the learned weights of the RBM below. The number of unsupervised training epochs changes from experiment to experiment, but it is mainly set to 1.

4.2.1 Techniques

Supervised training is performed using backpropagation with gradient descent. The weight updates in backpropagation are modulated using:

- Mini batch learning
- Momentum / Nesterov momentum
- Dropout
- Rmsprop
- Weight decay

4.2.2 Pre-training heuristic

During the pre-training of a DBN, we use the same learning rate for each stacked RBM. This is the norm in the literature, mainly due to the computational demand of discovering the best learning rate for each RBM. However, we noticed that the optimal learning rate needed for the first RBM to provide a good reconstruction of the training data was about 10 times higher than the one that we obtained with cross validation for the unsupervised learning rate of the DBN. We tried to employ a

¹⁴See what Yoshua Bengio says here: <https://groups.google.com/forum/#!topic/pylearn-dev/cBNms1QEmXc>.

heuristic, in which after cross validation the learning rate of the first RBM is set to be 10 times bigger than the one cross validation suggested. This heuristic improved accuracy in some of our experiments, but not all. We found that it did not work well for the MNIST digits (subsection 5) but improved performance for emotion recognition (subsection 6).

4.2.3 Norm constraints

We did not use an L_2 norm constraint on the incoming weights of a hidden units when training deep belief nets. Rescaling the weights at each step would not allow the network to benefit from the features that it has learned during pre-training. We performed experiments to see how rescaling the weights during RBM training with Contrastive Divergence would affect the reconstruction error and the final classification results. The classification error obtained by training a DBN with MNIST data by imposing L_2 norm constraints during both unsupervised learning and supervised fine-tuning was 21% with a squared norm constraint of 8 (found out via cross validation)¹⁵. This shows that imposing norm constraints using CD training substantially affects the features learned. We believe that there are multiple reasons why norm constraints are not a good idea for RBMs but they work well for other nets with dropout: firstly, CD is a poor approximation of the log likelihood gradient on which we would like to perform gradient descent (this is not a problem when using backpropagation, as the exact gradient of the error function is used). Secondly, the features learned by each RBM are quite different (belonging to different abstraction levels), so it is possible that each of them requires a different norm constraint. This requires more investigation.

4.2.4 Momentum

We found that momentum plays a crucial role in determining classification performance. Momentum is increased linearly, from a rate of 0.5 to the maximum momentum, determined by cross validation. The rate of increase of momentum can also have a high impact on the performance of the network. It is important to try multiple values before settling on a value. Changing the linear step from 0.1 to 0.01 can boost classification rates by as much as 10%.

4.3 Types of units

Depending on the task, we employed either sigmoid neurons or deterministic rectified linear neurons for discriminative fine tuning. For training the Restricted Boltzmann machines we used both stochastic binary units (for the visible and hidden layer) or noisy rectified linear hidden units and Gaussian visible units for the visible layer. In the case where Gaussian units are used, the data is preprocessed to have zero mean and unit variance.

¹⁵Apart from the L_2 norm constraint, the experiment setting were the same as described in subsection 5. It is suggested in [25] that a decaying learning rate would be beneficial when using L_2 norm constraints as we can have a high learning rate at the beginning of learning without making the weights too big. Decaying learning rates showed an improvement, but not a substantial one in our case.

4.4 Practical considerations

So far we have discussed the theoretical considerations and insights behind the techniques used. It is however useful to discuss some practical details as well, for completeness.

Dropout

When training a Restricted Boltzmann machine, for each data instance two dropout masks are created: one for the visible units and one for the hidden units. These masks are applied to the activations of the visible and hidden units and stay the same during contrastive divergence, thus ensuring that for this data instance only the selected units get updated. The update equations of the parameters (equations 3.15, 3.16 and 3.17) use the expected values of activations of units. These expected values have to be 0 for units that have been inactive for this data instance (as they have been dropped out).

After training, RBMs are used to obtain hidden activations for a specific input (as part of DBN pre-training) or for reconstructions. The visible pattern or the hidden activations are not dropped out during reconstruction, so on average the input of a neuron will be higher by a scale equal to the dropout factor than the input received during training. In order to balance that, we need to use a new set of weights, that have to take into account that the network was trained with dropout, so a percentage of the weights were not active when the features were learned. This is why for RBMs trained with dropout we have 2 reconstruction weights: one for the hidden units and one for the visible units. Usually, the weight matrix for the visible vector is just the transpose of the weight matrix for the hidden units. With dropout, the two weight matrices will differ because the incoming weights to a hidden unit have to be multiplied by the visible dropout factor, while the incoming weights to a visible unit have to be multiplied by the hidden dropout factor. These two factors usually differ, as the hidden dropout is usually set to 0.5 while the visible one is set to a higher value (0.8), in order to preserve more of the input features.

A similar method has to be applied to the weights of a deep belief net when used for classification. In order to benefit from the input image, dropout is not used at test time. Hence weights have to be scaled by the dropout factor corresponding to their input layer: the weights incoming of the first hidden layer have to be multiplied by the visible dropout, while the rest have to be multiplied by the hidden dropout factor.

Rmsprop

The learning rate used with Rmsprop has to be one order of magnitude smaller than the one without using Rmsprop. Training time can be substantially decreased by using Rmsprop when updating the network parameters. Comparing figures 4.1 and 4.2 we can see how using Rmsprop helps in speeding up training: without Rmsprop, the network converges in about 5000 epochs, with Rmsprop, less than 1000 epochs are required.

Softmax

Softmax units are often numerically unstable, especially on float32 architectures, used

for GPU code. The mathematically equivalent but numerically stable implementation of a softmax is given by:

$$m = \max_i x_i \quad (4.7)$$

$$y_i = \frac{e^{x_i - m}}{\sum_{j=1}^n e^{x_j - m}} \quad (4.8)$$

Hyper parameters

The hyper parameters were obtained using cross validation. When searched tried learning rates we tried from the range 10^{-1} to 10^{-5} . After finding a good interval for the learning rate of the form $(10^{-k-1}, 10^{-k})$, we also tried $5 \cdot 10^{-k-1}$. For maximum momentum, we tried 0.9, 0.95, 0.99. With mini-batches, we tried sizes of 10, 20, 50 and 100.

Early stopping

The idea behind early stopping is elegant and simple: use a validation set to check when the model is overfitting and when it stops training.

However, this is not easy to check in practice. Our initial implementation stopped training when the validation error increased 5 consecutive epochs. This technique resulted in the training ending too soon. We exemplify this using a deep net with 5 layers, trained on the Cropped Kanade data (as described in subsection 6). The network was trained with an unsupervised dataset of size 1429 and a supervised set of size 406. Our early stopping criteria suggested using 618 epochs, but further experiments (shown in table 4.1) suggest that training more is better. Figure 4.1 shows the validation error displayed by the network on a validation set¹⁶ after each epoch of training when training for 10000 epochs. You can notice how the in the first 200 epochs the error is decreasing steadily after which it fluctuates considerably.

It has been suggested [25] that dropout removes the need of early stopping, as it stops overfitting. It is true that the stopping of learning is no longer “early”, but a stopping criterion is required. Our experiments find that even though dropout allows the network to be trained more without overfitting happening, training for too many epochs can still cause overfitting (this can be seen in table 4.1, where the network has been trained with a hidden dropout of 0.5 and a visible dropout of 0.8).

The problem is usually dealt with by trial and error and plotting the training versus validation error. However, due to the multiple configurations that one can try when optimizing a deep belief net, we wanted to avoid having to manually oversee and check the number of training epochs required for supervised fine tuning. For this we tried multiple approaches. The first one was to train the network for a fixed number of epochs and record the best weights on the validation set, and at the end of training keep these weights as the network weights. This very greedy approach did not perform as well as expected. We found that in this case

¹⁶The training set size to validation set size ratio is 9:1

the network overfits to the validation set, without maximizing performance on the test set. Another method we could have tried is to increase the number of consecutive epochs on which the validation set error is increasing. We decided to not do this for 2 reasons: firstly, we needed to decide on a number of consecutive epochs of increased validation error after which to stop training. Making this choice had the same disadvantages as choosing the number of epochs to train the network for, and seemed arbitrary. Secondly, as figure 4.1 shows, the validation error can fluctuate considerably. The fact that it did not decrease for a given number of consecutive epochs does not mean that we should not stop training, because on average the prediction accuracy is getting worse.

We also tried the technique based on [44] which doubles the maximum number of iterations performed during training every time the validation score improves the best validation score found so far. We found that this works best out of all the heuristics that we tried. However, this technique has a significant downside: it requires checking the validation error after each mini batch, not only after each epoch. This increases the computational demand significantly, adding an overhead to the training time. Moreover, any technique that requires a validation set in order to decide when to stop training decreases the size of the training set, hence affecting accuracy by using less training data.

In the end we settled for a hybrid approach: we used the validation technique that doubles the training epochs when the accuracy on the validation set is the biggest obtained so far to give us an idea of when training could be stopped. We record the number of epochs the validation technique suggests for training, we double it and train the network for that number of epochs. If the test accuracy decreases, we stop. If not, we double the number of epochs again and repeat the procedure.

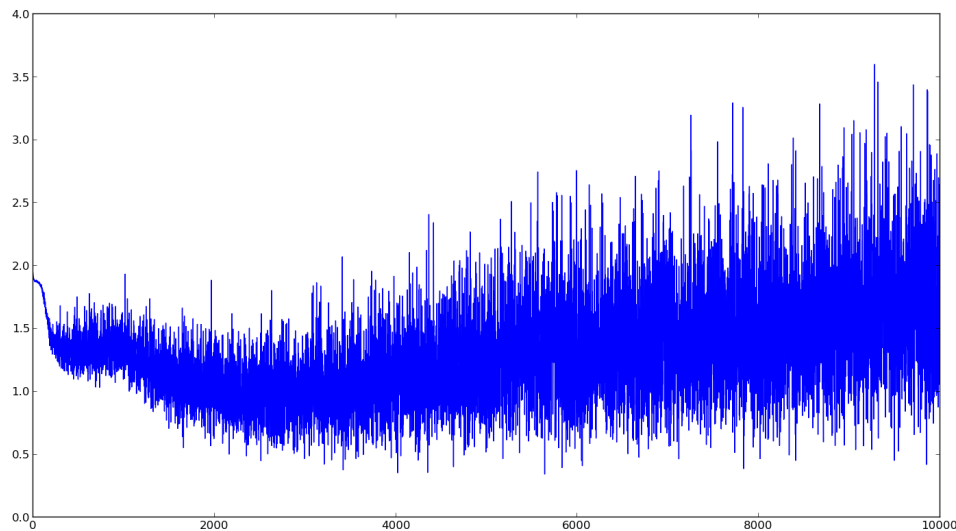


Figure 4.1: Validation error on 10000 epochs on a network trained without Rmsprop.

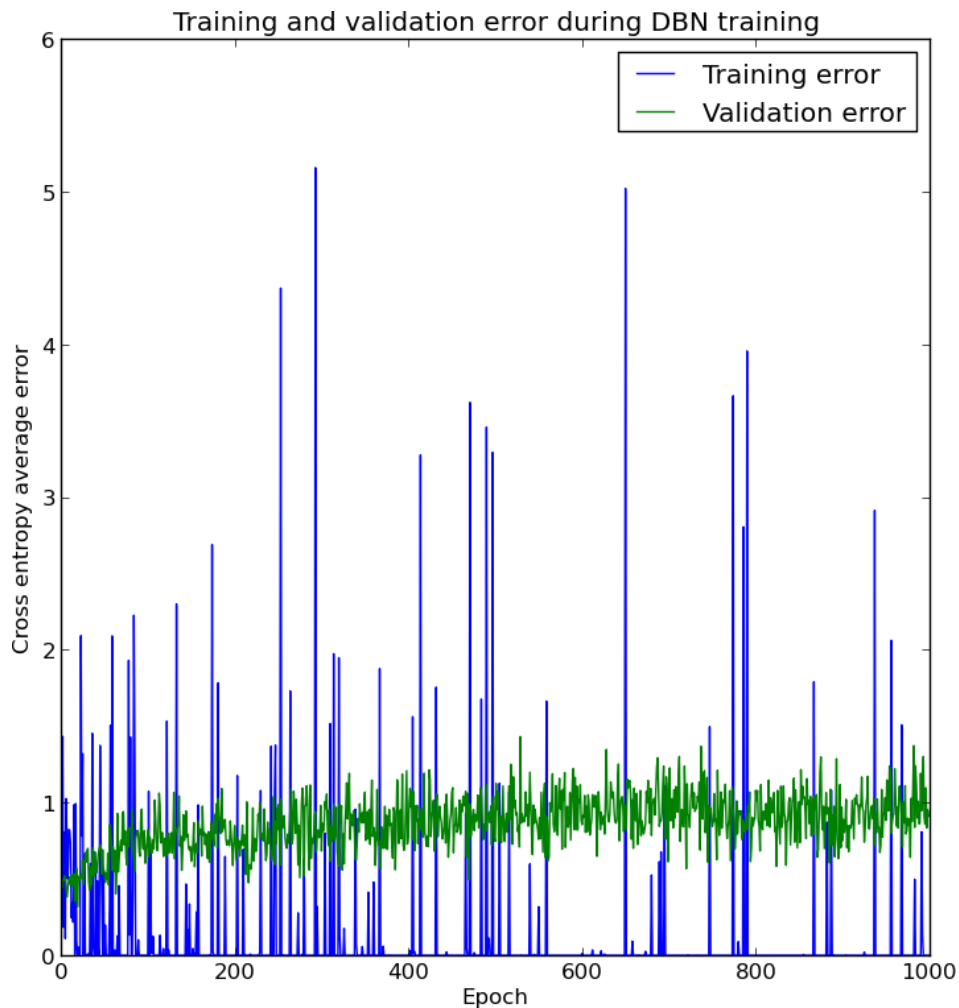


Figure 4.2: Validation error on 1000 epochs on a network trained with Rmsprop.

Number epochs	Test accuracy
618	40.2 %
1000	54.9 %
2000	62.2 %
5000	65.8 %
10000	63.4 %

Table 4.1: Relation between test accuracy and number of training epochs when not using Rmsprop on a network built to classify emotions.

5 | A standard benchmark: recognizing handwritten digits

We will now verify our model through a standard benchmark for machine learning algorithms: the MNIST dataset of handwritten digits¹⁷. This allowed us to make graduate improvements to our model and compare to well known results. The input images are given in a 28×28 resolution. The dataset is quite big, with 60000 training and 10000 testing cases, permitting us to focus on the core algorithms developed without having to worry about the quality and the size of the data given.

Figure 5.1¹⁸ shows examples from the dataset, in order from 0 to 9. Looking at the digits it becomes clear that classification is non trivial, as the digit 5 could easily pass as a 3.



Figure 5.1: Digit examples from the MNIST dataset.

5.1 Learning features

Figure 5.3 exhibits 100 of the weights that are learned by a RBM when trained with 2s in the MNIST dataset. All units learn the general figure of a 2, but each unit is specialized for a specific feature that can be seen in the slight differences between the intensities of the weights for different features. For example looking at the first weight vector we can observe that the hidden unit is more likely to be active when the lower stroke of the 2 is curly and has a specific shape (determined by the high intensity white pixels). Figure 5.2 exemplifies how the incoming weight vector is used to determine the probability of a hidden unit being active. The black pixels correspond to an input of magnitude 0 and the white pixels corresponding to a magnitude of 1. The dot product in equation 3.13 can be seen as an element-wise product (Hadamard product) between the weight vector and the input, followed by a sum. Only the non-zero elements of the input contribute to the sum that will be the input of the sigmoid function. The overlap between these non-zero elements and the strong incoming weights (corresponding to the feature the neuron has learned) determine if the unit will be active or not, as a high value of the weight will have a large effect on the sum. Hence if the input vector contains the feature the hidden unit has learned during training then there is a higher probability of the unit to get active when creating the hidden representation of the input. This is why the units of RBMs are often called “feature detectors”.

While figures 5.3 and 5.4 show the weights learned by an RBM trained with CD, figures 5.5 and 5.6 show the weights learned by an RBM trained with Persistent contrastive divergence. We note

¹⁷The dataset can be found here [39] together with the results obtained by various techniques.

¹⁸All the figures presented in this section have been smoothed using interpolation for presentation purposes.

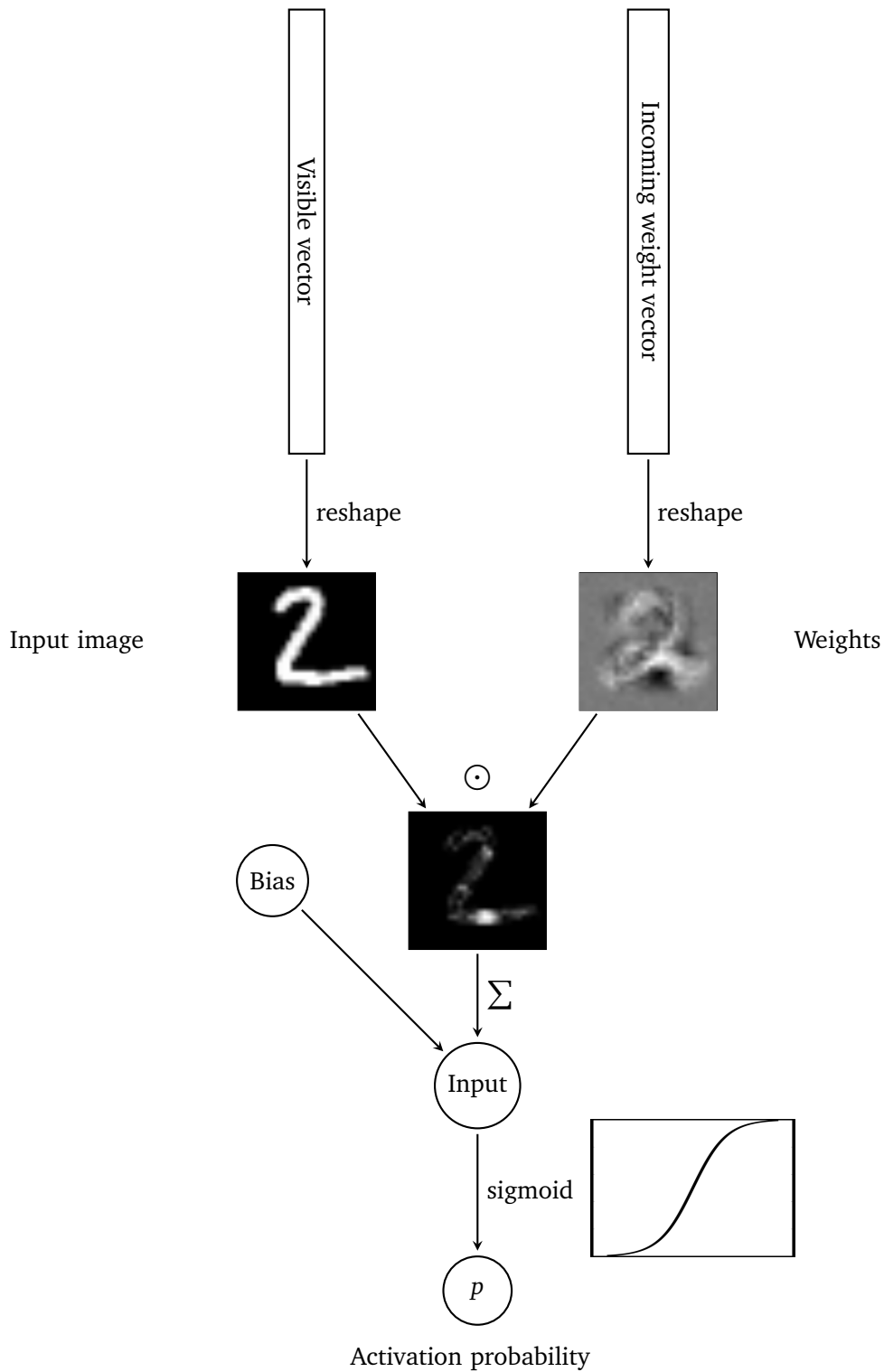


Figure 5.2: Pictorial representation of how to obtain the activation of a hidden unit in a RBM. \odot denotes the Hadamard (elementwise) product between two matrices.

that since weights can have negative entries, the pixel intensity is not a direct representation of the magnitude of the weight, as the weight sign influences the pixel as well.

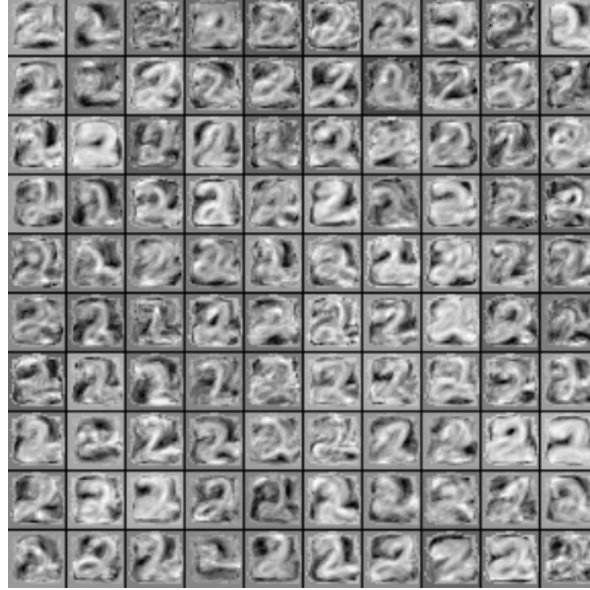


Figure 5.3: The incoming weights of 100 of the 500 hidden units of an RBM trained using Contrastive Divergence with instances of the digit 2 from the MNIST dataset. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.

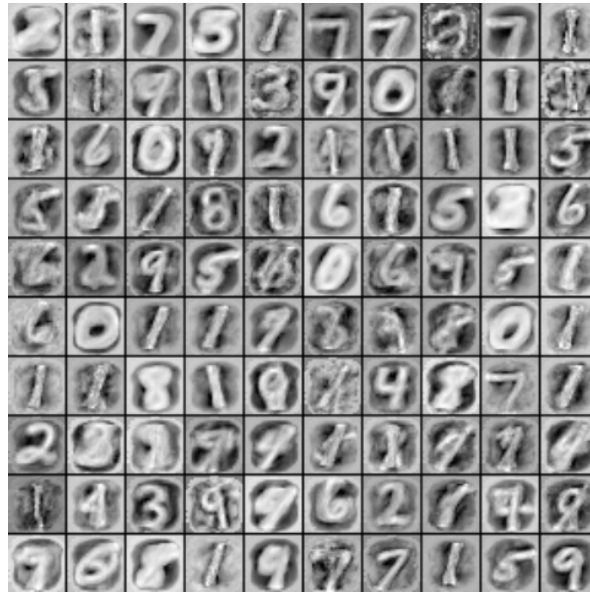


Figure 5.4: The incoming weight vectors of 100 of the 500 hidden units of an RBM trained using Contrastive Divergence with instances of all 10 digits in the MNIST dataset. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.

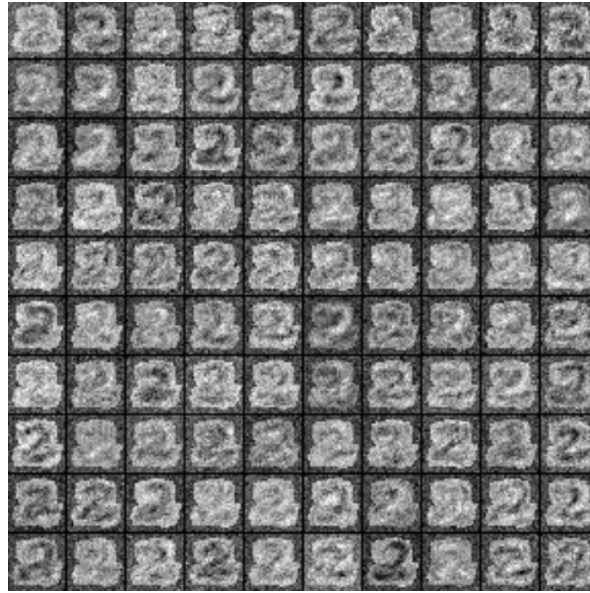


Figure 5.5: The incoming weight vectors of 100 of the 500 hidden units of an RBM trained using Persistent Contrastive Divergence with instances of the digit 2 from the MNIST dataset. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.

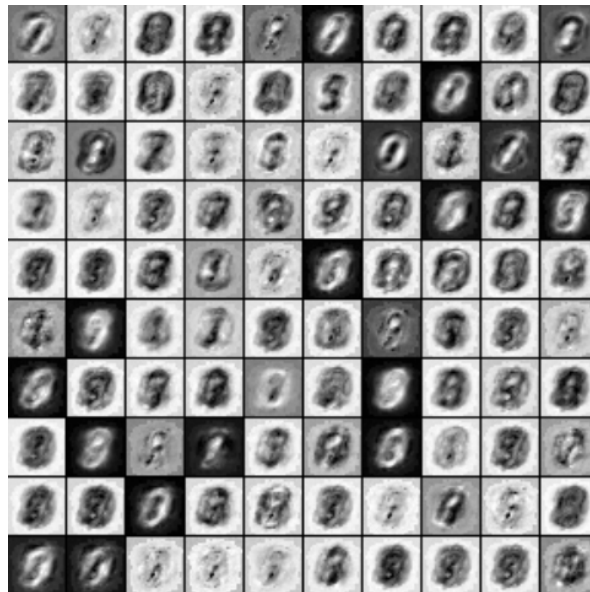


Figure 5.6: The incoming weight vectors of 100 of the 500 hidden units of an RBM trained using Persistent Contrastive Divergence with instances of all 10 digits. Weights are reshaped to have the same shape as the image inputs for visualisation purposes.

As RBMs are generative models, we can use them to sample from the distributions they model. After training, we can start with a random pattern and let a Markov Chain run through the network. We first show the results obtained after training only with only digit the 2, and then we exemplify the same results when training the RBM with all 10 digits.

Figure 5.7 shows how a Restricted Boltzmann Machine trained only with 2s will reproduce the only digit it has been exposed to during learning even from a random pattern. This reconstructed digit is very similar to the average (or even close to ideal) representation of a 2. Figure 5.8 exhibits what happens when a random pattern is fed as input to an RBM that was trained with all digits. The reconstructed image seems to be a fuzzy overlap between the digits the network has been exposed to.

Reconstructions of digits appear to be very accurate: from figure 5.9 we see that the original shape of the digit persists but its prominent features are “softened”, because this particular instance of the digit 7 is slightly different than the average one the model has learned when trained with images of the 10 digits in the dataset. We note that the input fed to the network (displayed in the left subfigure of figure 5.9) was not part of the training set of the RBM. Figure 5.10 shows how the 7 gets distorted into a 2 when fed as input to a network trained only with the 2s in the dataset. The network has never been exposed to a 7 before, and it thinks that the “world” is only made of 2s, because that is what it has learned. It then tried to fit the input to what it knows, so the reconstruction has the shape of an incomplete 2. This is similar to a human experience during learning: when humans have to deal with something unknown to them, they try to find similarities with what they have seen before in order to understand what they could do with the new encountered object, problem, situation.

5.2 Learning labels

For the discriminative task on MNIST we used a network with 5 layers, the last one being the softmax classification layer. The dimensions of the layers were as follows: 784 (dictated by the size of the input images), 1000, 1000, 1000 and 10 (required by the number of classes).

Table 5.1 shows classification results obtained on MNIST. The experiment setting was as follows: Nesterov momentum was used for all experiments. Momentum always starts at 0.5 and increases in steps of 0.01 until it reaches 0.95 (both for pre-training and fine tuning). The mini batch size is 20^{19} . The learning rate was multiplied by $(1.0 - \text{momentum})$. All networks were trained using Rmsprop, to increase the speed of learning. Dropout was supervised fine tuning, but not for pre-training. For supervised training we used a dropout of 0.8 for the visible units and 0.5 for the hidden units. No weight decay was used in any of the experiments. When the units of the RBM were stochastic binary, the activation function used was sigmoid, both for RBM training and for supervised training. When Gaussian units were used for the visible units of RBMs, we scaled the data to have zero mean and unit variance and used the identity activation function. For the noisy rectified linear units we used the activation function described in definition 3.9. When

¹⁹The mini-batch size is an important parameter that can highly influence classification accuracy.



Input pattern.

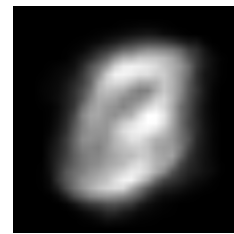


Visible reconstruction.

Figure 5.7: The visible reconstruction of a random pattern using an RBM trained using Contrastive Divergence with 2s from the MNIST dataset.



Input pattern.



Visible reconstruction.

Figure 5.8: The reconstruction of a random pattern using an RBM trained using Contrastive Divergence with samples from all 10 digits.



Input pattern.

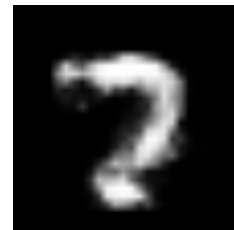


Visible reconstruction.

Figure 5.9: The reconstruction of an instance of the digits 7 using an RBM trained using Contrastive Divergence with instances of all digits.



Input pattern.



Visible reconstruction.

Figure 5.10: The reconstruction of an instance of the digits 7 using an RBM trained using Contrastive Divergence with only instances of the digits 2.

noisy rectified linear units are used for training an RBM, the deterministic version is used in the supervised training (see definition 3.10).


Figure 5.11 shows 10 digits which were misclassified by our model, together with the predicted and correct labels. While some of the digits can be easily classified by a human (for example, the first digit is clearly a 9²⁰), some of them are hard to label. It is unclear what the third and fourth digits should be, and the guess of the network is not far from the guess of a human.

Our results are comparable with similar results obtained when performing classification on MNIST. We note that convolutional networks perform better on this task, but convolutional nets directly encode information about the image topology in their architecture.

We conclude by saying that these results could be improved by spending more time tuning the parameters via cross validation (learning rates, momentum, mini batch size and training epochs), but due to the limited time we decided to focus on emotion recognition, described in next section.

Unsupervised learning rate	Supervised learning rate	RBM unit types	Supervised epochs	Unsupervised epochs	Error
0.01	0.05	Stochastic binary	100	1	1.4%
0.01	0.05	Stochastic binary	100	10	1.3%
0.01	0.05	Stochastic binary	500	1	1.2%
0.005	0.005	ReLU	100	1	1.55%
0.005	0.005	ReLU	200	1	1.15%
0.005	0.005	ReLU	300	10	1.1%

Table 5.1: Classification results obtained on MNIST. Details about the experiments are found in text.



Actual label	9	4	2	5	3	6	2	8	8	1
Predicted label	8	2	7	3	7	0	1	2	2	8

Figure 5.11: Examples of misclassified digits from MNIST.

²⁰The network probably sees it as an 8 due to the little gap between the bottom left curl of the 9 and the circle on top.

6 | Emotion recognition

We move on to discuss the application of deep belief networks for emotion recognition in images of humans²¹.

We performed the emotion recognition experiment with various settings. We first tried to combine multiple databases together in order to achieve a substantially large dataset (subsection 6.2). We then focused on one bigger and more comprehensive database, namely Multi PIE. We used it to perform emotion classification in various settings, described in detail in subsection 6.3.

6.1 Data preprocessing

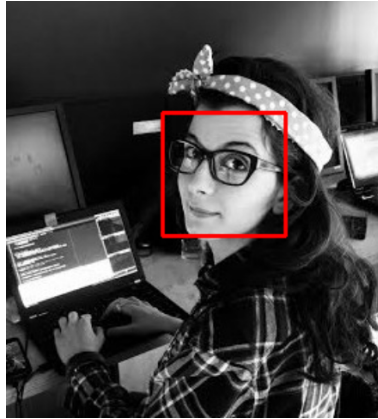
Due to the various origins of the databases, creating one single dataset required preprocessing, including **face detection** and **histogram equalization**. For face detection we used the method known as the Viola Jones classifier [47]. We used the implementation found in the OpenCV library [48]. The algorithm not only allows you to see if a face is present in the picture, but also detects a rectangle where it thinks the face might be (according to the facial features in the area). We used this rectangle to crop the images in certain datasets, to remove the noisy background information, which is irrelevant for emotion classification. Another preprocessing technique used ensures that the histogram of pixels is spread instead of very skewed, hence improving the image contrast²². We found that equalization substantially improves results (table 6.2). An overview of the equalization methods that we used in this project can be found in [49]: depending on the dataset, we used either global equalization or Contrast Limited Adaptive Histogram Equalization (CLAHE). Figure 6.1 displays sample faces from the used datasets before and after histogram equalization together with an example of a face detected by the Viola Jones classifier. An important preprocessing step in facial emotion recognition is **alignment**. For the network to easily learn what features correspond to different emotions, it is best to ensure that different facial parts are aligned, and present in the same area of each image. A way to visualise this is by thinking of overlapping all the images in the dataset and ensuring that the nose, eyes, mouth will also overlap between all posing subjects. We did not perform face alignment ourselves, but the faces in the Multi PIE dataset and Cropped Kanade dataset were already aligned before we used them for our experiments.

6.2 Cohn-Kanade, Jaffe and other databases

The Cropped Cohn-Kanade database contains 407 images labelled with 7 basic emotions: anger, disgust, fear, happiness, sadness, surprise and neutral. All subjects are depicted from frontal pose. We used a 40×30 resolution of the data, which is small enough in order to make training feasible

²¹Emotion recognition can also be done from sound, and that in itself is an interesting topic but not in the scope of this paper.

²²The idea of histogram equalization came from the problem face detection. It is recommended that images are equalized before given as input to the Viola Jones classifier.



Face detection using the Viola Jones classifier.



Histogram equalization of pixels. The original image is presented on top of the equalized image. The images are taken from the following datasets (left to right): Multi-PIE, Cropped Kanade, Jaffe and YaleB.

Figure 6.1: Visualization of image preprocessing techniques used before training a classifier to label emotions.

(in terms of computational capacity) but big enough for the network to be able to extract features from it. We used it in conjunction with other unlabelled datasets. An overview of the datasets is provided in table 6.1.

The Cohn Kanade database can be used to exemplify the power of image reconstruction by RBMs (figure 6.3). It is interesting to notice that some particular features of the face are ignored: it is uncommon for a person to have hair on their forehead (fringe), so most of the training data did not have that feature. As a consequence, in the reconstruction the fringe does not appear and the forehead is empty. Figure 6.4 emphasises the effects of Rmsprop on feature learning: features are substantially more prominent when using this technique. As a consequence we employed Rmsprop for both supervised and unsupervised learning in the experiments below.

Table 6.2 determines the classification accuracies obtained when training a deep belief net with the labelled Cropped Cohn Kanade dataset. The network is capable of classifying emotions at an accuracy of 80% given a dataset of size of only 407 images. The experiments were performed on a deep belief net with 3 hidden layers of 1500 hidden units and a softmax layer of 7 units. The dropout rates were 0.8 and 0.5 for the visible and hidden layer, respectively²³. Rmsprop and Nesterov momentum were used both for RBM and DBN training. The learning rate is multiplied by $1 - \text{momentum}$, thus achieving an effect similar to a decaying learning rate, because momentum grows linearly before achieving a maximum of 0.95. Rectified linear units were used for all reported experiments. The number of pre-training epochs is 20 and the number of supervised epochs is 2000. We used the heuristic described in 4.2.2 for this set of experiments, thus the learning rate of the first RBM is set to be 10 times bigger than the one of the other RBMs, but capped to 1.0. When we used only the Cohn-Kanade dataset the supervised learning rate was 0.01

²³Without dropout the accuracy drops by 20%.

and the unsupervised learning rate was set to 0.05. For the Cohn-Kanade + Jaffe experiments we used different unsupervised learning rate: 0.01 and we used the mean square error sparsity constraints we introduced in 4.1.2 with a desired sparsity of 0.01 and a regularization parameter of 0.001. We note that the Viola Jones classifier did not detect all faces from the Jaffe dataset, so we were able to use only 200 of them when we trained the classifier with the images cropped around the face. We did not use the labels from the Jaffe dataset, we used it only for unsupervised training. Adding the Nottingham and YaleB datasets for unsupervised pre-training gave similar results as Jaffe together with Att, so we do not report them here.

Database	Labelled	Sexes	Reference
Cropped Cohn-Kanade	✓	Both	[50]
Jaffe	✓	Female	[51]
Cropped Yale	✗	Both	[52]
Nottingham	✗	Both	[53]
Att	✗	Both	[54]

Table 6.1: Databases used for emotion recognition.



Figure 6.2: Images from the Cropped Kanade database. Emotions displayed (left to right): anger, disgust, fear, happiness, sadness, surprise and neutral.



Face from Cropped Kanade database.



Visible reconstruction.

Figure 6.3: Reconstruction of a face from the Cropped Kanade database, using a Restricted Boltzmann machine.

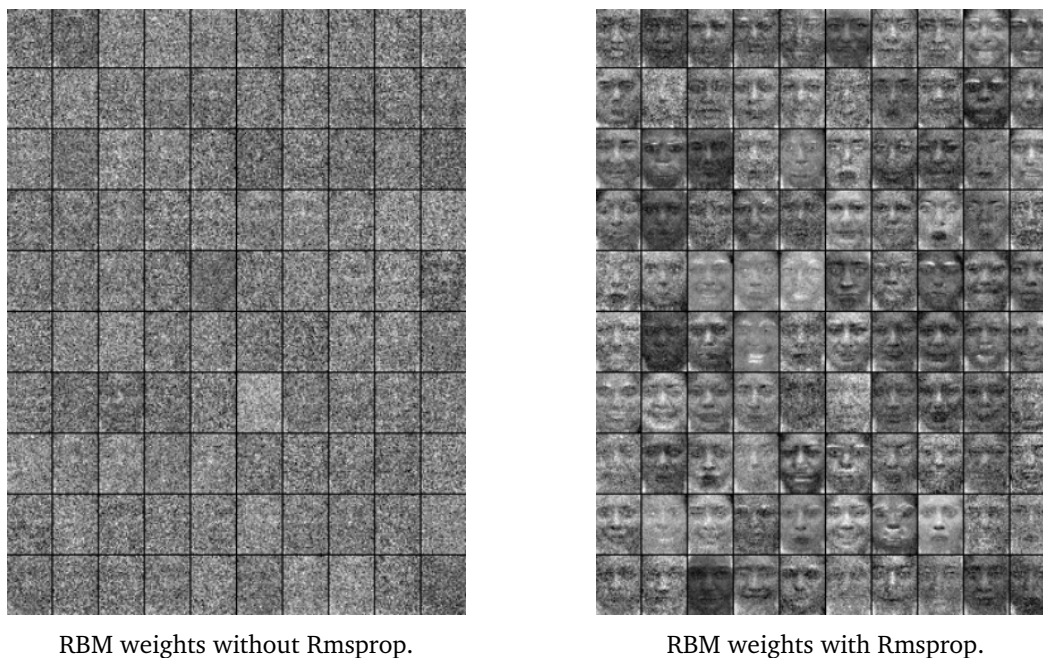


Figure 6.4: Visual comparison of weights of an RBM trained with and without Rmsprop.

Databases for unsupervised training	Preprocessing	Accuracy
Kanade	None	72.0 %
Kanade	Equalization	81.3 %
Kanade, Jaffe,	None	70.8%
Kanade, Jaffe	Face detection	76.8 %
Kanade, Jaffe,	Equalization, Face detection	81.7 %
Kanade, Jaffe, Att	Equalization, Face detection	81.7 %

Table 6.2: Accuracy obtained when training a deep belief net with various unlabelled datasets. The only labelled dataset is the Cropped Kanade dataset. Details found in text.

6.3 Multi PIE

We will now move to describing another set of experiments, involving the Multi PIE database. The database comes with 22050 images, labelled with 6 emotions: neutral, surprise, squint, smile, disgust and scream. Moreover, the database has labelled the subjects (147 in total), the pose (5 in total) and the illumination conditions (5 in total) in which the images were taken. Figure 6.5 displays 3 different subjects, displaying each of the labelled emotions. In order to make it clear how emotions are depicted, we kept the same pose and illumination condition within subjects, but changed them in between different subjects.



Figure 6.5: Images of subjects from the Multi PIE database. Emotions (from left to right): neutral, surprise, squint, smile, disgust, scream. Each subject is shown in a different pose and different illumination.

We performed emotion classification using the Multi Pie database in different settings:

- **Random data split:** training and test data contain all subjects, poses and illuminations
- **Different subjects:** test the network with subjects that were not in the training set
- **Different illuminations:** train the network with 4 of the 5 illumination conditions, test with the 5th
- **Different poses:** train the network with 4 of the 5 poses, test with the 5th
- **Missing data:** test the network with images containing contiguous patches of missing data.

The network architecture for these experiments had 3 hidden layers of 1500, 1500 and 1500 units and a softmax layer of 6 units. The unsupervised learning rate was 0.005 and the supervised one was 0.001²⁴. Dropout was used, with 0.5 in the hidden layers and 0.8 in the visible layers²⁵. Rectified linear units were used for the hidden units. During pre-training Gaussian visible units were used, with the data being scaled to have zero mean and unit variance. Unless specified in the description of a particular experiment, equalization was not used. Rmsprop was used both for RBM and DBN training. Nesterov momentum was used, with momentum increasing linearly after

²⁴We tried different learning rates for each experiment, but these ones performed best in all of them. We noticed a high sensitivity of the network to the learning rate.

²⁵For this problem, hidden dropout did not perform well when using sigmoid units instead of rectified linear units. However, adding hidden dropout to rectified linear units improved performance

T \ P	neutral	surprise	squint	smile	disgust	scream
neutral	728	1	4	0	0	0
surprise	1	729	0	1	0	2
squint	1	0	729	2	0	0
smile	0	0	1	728	4	0
disgust	0	0	0	3	729	0
scream	0	0	0	0	2	732

Table 6.3: Average confusion matrix obtained by training splitting data in folds with a ratio 4 to 1 (train to test). Average classification rate: **99.3%**.

each epoch of training until it reaches 0.95, after which it is kept constant. Both L_1 and L_2 weight decay regularization were used, with a penalty hyper parameter of 0.001 in both cases.

6.3.1 Random data splits

This experiment randomly partitions the dataset into the training and testing data such that the ratio between the number of instances in the two sets is 4 to 1. The confusion matrix is displayed in table 6.3. The classification accuracy obtained was **99.3%**. The only preprocessing technique used was global histogram equalization.

6.3.2 Different subjects

To test the prediction accuracy of the network on a subject it has never seen before, we performed an experiment which ensures that the subjects which appear on the training data will not be in the testing data. With no preprocessing on the input, the classification accuracy for this experiment was **91.3%**. Doing the same experiment with global histogram equalization on the input images, the accuracy increased, but not substantially, reaching **91.7%**.

6.3.3 Different illuminations

We tested the network with instances under a different illumination setting than the ones it was exposed during training. We did this for each of the 5 illuminations and averaged the results: the classification rate was **89%**. Most folds gave very good performance (such as 94%), apart from one (corresponding to illumination setting number 3, which gave an accuracy of 60%). Table 6.4 shows the confusion matrix obtained by averaging out the 5 confusion matrices obtained for each test-train illumination pair. Table 6.5 shows the confusion matrix obtained when training with illumination settings 1, 2, 4 and 5 and testing with illumination setting 3. When equalizing input images the different illuminations play less of a role, resulting in a substantial increase in accuracy: from 89% with no equalization to 94.8% with global equalization performed on the input data.

The average confusion matrix obtained is shown in table 6.6. We note that even with equalization, the performance obtained by illumination condition 3 is decreased compared to others, giving a classification score of 88%.

T \ P	neutral	surprise	squint	smile	disgust	scream
neutral	671.8	5.2	49.2	2.6	6.2	0
surprise	34.8	651.0	23.6	5.8	13.2	6.6
squint	12.4	1.6	708.0	4.4	8.4	0.2
smile	30.6	1.4	84.2	587.2	30.8	0.8
disgust	22.2	3.8	55.6	8.2	642.2	3.0
scream	20.2	13.0	26.8	6.2	17.2	651.6

Table 6.4: The classification accuracies obtained when testing with an illumination to which the network was not exposed to during training. Average classification rate: **89%**.

T \ P	neutral	surprise	squint	smile	disgust	scream
neutral	642	1	92	0	0	0
surprise	171	432	105	9	6	12
squint	35	0	700	0	0	0
smile	151	2	404	126	49	3
disgust	109	9	265	6	339	7
scream	101	52	134	30	51	367

Table 6.5: Confusion matrix obtained when testing with illumination type 3 and training with illuminations 1, 2, 4, 5. Experiment classification accuracy: **59.8%**.

T \ P	neutral	surprise	squint	smile	disgust	scream
neutral	702.0	7.0	23.8	1.0	1.2	0.0
surprise	4.0	716.8	4.2	1.0	5.4	3.6
squint	14.8	7.0	701.2	6.0	5.2	0.8
smile	5.8	10.0	19.8	665.0	34.0	0.4
disgust	10.4	13.6	4.4	22.6	679.8	4.2
scream	1.6	5.4	1.6	0.2	8.6	717.6

Table 6.6: Average confusion matrix obtained when equalizing the input. Average obtained accuracy is **94.8%**.

6.3.4 Different poses

Another robustness test involves testing the network with a pose it has not been presented during training. Figure 6.6 displays the 5 different poses we used from the MultiPie dataset. Table 6.7 gives the classification results for each of the 5 folds, in 2 different experiments: in the first one,

unsupervised pre-training is done as usual, with only the training data but in the second one we used the testing data in the unsupervised pre-training. This increased the accuracy of our classifiers.

	Pre-training with train data	Pre-training with train and test data
Pose 0	16.8%	47.9%
Pose 1	56.3%	44.2%
Pose 2	45.7%	46.2%
Pose 3	47.3%	43.6%
Pose 4	16.8%	32.7%
Average	36.5%	42.9 %

Table 6.7: The classification accuracies obtained when testing with a pose to which the network was not exposed to during training.



Figure 6.6: The 5 different poses displayed in the Multi PIE database.

6.3.5 Missing data

In order to assess the robustness of our emotion recognition network we tested it with images in which we added 5×5 squares of black pixels at random positions in the image. The training was performed with images from the original Multi PIE dataset. The train to test ratio was set to 3:1. No hidden dropout was used for this experiment. With unsupervised training for 20 epochs and supervised 2000 epochs, we obtained an accuracy of **91.5%**. When we used 10×10 the accuracy dropped to **65%**. This is expected giving that we are depriving the network of 1/12 of the input image during test time. Figure 6.7 shows example test images used. We note that there was no specific trend in the error performed by the network, the errors were equally distributed to all the labels.



Figure 6.7: Test images from the missing data experiment.



Figure 6.8: Facial areas according to contribution to the classification error in the missing data experiments. A stronger colour indicates a higher error when the network was deprived of the image pixels in the corresponding area (by making the pixels black). We notice that the most important area is around the mouth, followed by the region between the eyes (which is helpful to distinguish frowning and screaming).

Figure 6.8 visually illustrates the results from another set of experiments. Instead of randomly choosing a 10×10 square from which to omit the data, we chose one of the 12 squares for which both coordinates are multiples of 10, thus forming a partition of a 40×30 image. We tested the network with these images and created a heat map according to the average classification error obtained when that particular area was omitted from the input pixels (by setting the pixel value to 0). This allows us to see the areas which contribute most to the emotion recognition process. As expected, the mouth contains the most information used in the classification process. Figure 6.8 shows that the area between the eyes also exhibits features required for classification and so do the cheekbones.

6.4 A wild dataset

The labelled datasets we have used so far (Cropped Cohn Kanade and Multi PIE) depict humans in highly confined environments and are aligned. We wanted to assess our method on another type of dataset. For this we used one available on a website which hosts machine learning competitions, Kaggle ²⁶. The dataset used ²⁷ had input of size 48×48 with a training set of size (details about

²⁶<http://www.kaggle.com/>

²⁷<https://inclass.kaggle.com/c/facial-keypoints-detector/data>

Experiment	Equalization	Accuracy
Random data splits	✓	99.3 %
Different subjects	✗	91.3 %
Different subjects	✓	91.7 %
Different illuminations	✗	89.0 %
Different illuminations	✓	94.8 %
Different poses	✓	36.5 %
Different poses, unsupervised pre-training with test data	✓	46.3 %
Missing data 5×5	✗	91.5%
Missing data 10×10	✗	65 %

Table 6.8: Comparison of experiments performed on the Multi PIE database.

the dataset can be found in [55]. The training instances are labelled with one of 7 emotions: angry, disgust, fear, happy, sad, surprise, neutral. This dataset is substantially harder to classify than the ones we have used before, as it is not aligned, and sometimes the images do not contain the entire face of the subject. Moreover, some of the input images do not contain a face at all or contain a distorted face. We cannot compare directly with the results reported, as we do not have access to the labels of the testing set provided for the competition (they were not made public by the organizers)²⁸. However, we used the training data that is publicly available for both training and testing by partitioning it such that the train to test ratio is 4. We obtained an accuracy of **69.6%**, by training a deep belief net of 3 hidden layers each of size 1500 (the visible layer had an input size of 2304), with a supervised learning rate of 0.01. We used rectified linear units, rmsprop and Nesterov momentum (increased linearly up to 0.95). The percentage of hidden units dropped out was 50%, and the percentage of visible units dropped was out 20%. For pre-training we used Nesterov momentum, Gaussian visible units and Noisy rectified linear units, with a learning rate of 0.05, for only 1 epoch. The mini-batch size used was 20. When we equalized the images, the accuracy increased to **79.3%**.

We note that while training with this dataset leads in a decrease in performance accuracy it is more suitable to use for a real life application (such as detecting emotions from faces presented in a web cam).

6.5 Emotion similarity

This section describes the experiments we have performed in order to find out if we can train a network to distinguish between emotions without having to label them. It was shown in [43] that Restricted Boltzmann machines can detect features of faces in images and that a twin network architecture can be used to determine if the two images represent the same person. We will use a similar architecture for our emotion experiments. Firstly we replicate their experiment on the

²⁸<https://inclass.kaggle.com/c/facial-keypoints-detector/forums/t/4053/test-set-labels>



Figure 6.9: Images from the dataset used in the Kaggle competition for emotion recognition.

aligned Multi-PIE database and then describe the emotion experiments that we have performed. To our knowledge no one else has performed such an experiment before.

When computing data features with the RBM for the cosine distance, we used the activations of the hidden units not the sampled values, to avoid sampling error. As the network is made of rectified linear hidden units, we used the expected value we defined in appendix D.

The network architecture we used is similar to the one described in [43]: we train a Restricted Boltzmann Machine with the dataset of faces. We then replicate the Restricted Boltzmann machine and use it in order to find the hidden representations (features) of both our inputs. The cosine similarity between the two vectors is used to decide if they represent features of the same face by creating a simple learning unit that we can train via backpropagation. This learning unit has one input x , a weight w and one bias b and its output is $\frac{1}{1+e^{wx+b}}$.

The entire architecture is shown in figure 6.10. The RBM used has 1000 hidden units (noisy rectified or sigmoid) and 1200 visible units (Gaussian or sigmoid). We tried 500, 700 and 1000 and 2000 hidden units, but using 1000 units gave the best results. We trained the network using Nesterov momentum and Rmsprop. When we performed the discriminative fine tuning with backpropagation, we set the parameters of our model to be both w and b ²⁹ and the parameters of the RBM that affect the error (the weights and hidden bias vector). We then used stochastic gradient descent with classical momentum to update them. We found that Rmsprop does not increase performance for this type network, so we did not employ this technique during training. The error used to compute the gradients with respect to the parameters was the square of the difference between the predicted output and the true label (namely 0 if the two depicted people are different, 1 otherwise). As an error measure we used binary cross entropy, as it performed slightly better than the square difference between the correct labels and the output probability. No dropout was used in the RBM training³⁰.

We used trained the RBM for 50 epoch and the supervised network for 500 epochs (we also tried 1000 epochs, but the precision did not increase substantially as to justify the extra computational power used). Maximum momentum was set to 0.95. It is interesting to notice that in some cases noisy rectified linear hidden units were outperformed by sigmoid units (table 6.9). For networks

²⁹At the beginning of training, w and b are initialized to 0.

³⁰We experimented with using dropout in this setting, but it decreased performance.

with sigmoid units the supervised learning rate was 0.001 and the unsupervised learning rate was 0.005. For networks with noisy rectified linear units the rates were 0.005 and 0.0005, respectively.

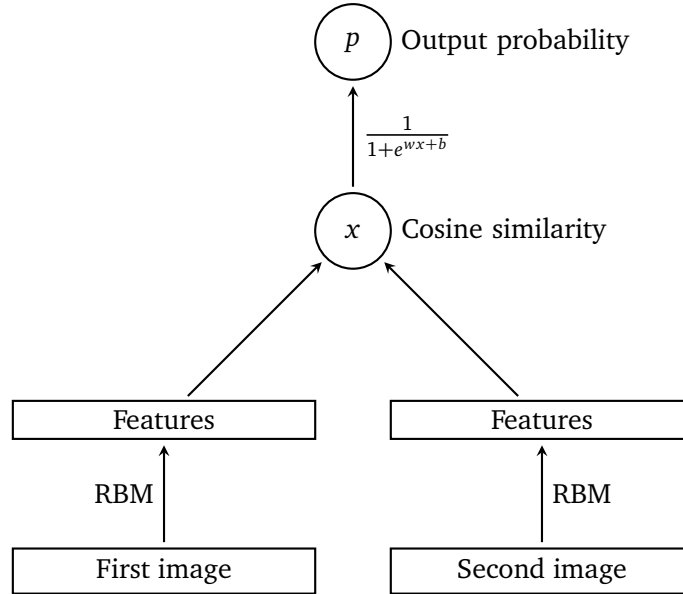


Figure 6.10: Network architecture used for similarity detection.

6.5.1 Same person

As a first step into our experiments, we made a model capable of distinguishing between subjects depicted in images. The applications of such a model vary from judicial systems to social networks: it can be used for automatic subject matching in criminal databases, as well as face tagging in a social network (some social sites provide this functionality today via suggestions, but the accuracy can be improved).

In order to determine if two faces are the same or not, we decided to use the threshold of 0.5: hence if the probability outputted by the network is bigger than 0.5, we classify the two faces as the same, if not as different. By looking at the confusion matrices we saw that this threshold is the correct one to take, as the network does not tend to err in one particular direction (ie. the number of faces *wrongly* classified is the same for both labels - different or same).

Experiment a

The Multi PIE dataset comes as a set of pictures, without them being split into pairs, which is what we required for these experiments. When we performed the splitting, we decided to do it such that each image from Multi PIE is part of exactly 2 pairs in our constructed dataset. Before constructing the pairs, we randomly shuffled the data, as to avoid more similarity in the images caused by how the Multi PIE dataset is structured (the images of subjects in the same poses and illuminations are grouped together). We trained the network by splitting

the data into training and testing making a 4 to 1 ratio. When constructing the training and testing sets we ensured that the network is exposed to the same number of instances in which the subjects are same person and in which they are not. This ensures that the network is not biased towards any of the classes due to increased exposure during training. There were no other special arrangements made, and the network sees all the subjects at least once. When training the RBM, we used the concatenation of training pairs from the constructed dataset.

Experiment b

In this experiment we ensured that the testing set did not contain any image of a subject that was in the training set. For that we split the images according to subjects they depict, and define the two sets such that no subject appears in both. We partitioned the training data as in the first experiment, to ensure that half of the training and testing sets contained positive examples (of pairs of pictures which represent the same person) and half of the images represented negative examples (of pictures which represent different individuals). The train to test ratio was kept to 4, in order to be able to make a fair comparison between experiments. As before, a particular image was part of two image pairs as part of either a training pair instance or a test pair instance.

Experiment c

In this experiment we compared the networks test ability when the test subjects are not part of the same database as the database the network was trained with. In order to do so we trained the network as before with the Multi PIE database, but tested it with subjects from the Cropped Yale database [52].

Table 6.9 shows the classification results of the experiments described.

Experiment	Train Database	TestDatabase	Hidden Units	Accuracy
Experiment a	Multi PIE	Multi Pie	ReLu	86.0 %
Experiment b	Multi PIE	Multi Pie	Sigmoid	82.5 %
Experiment c	Multi PIE	Cropped Yale	ReLu	65.0 %

Table 6.9: Results from the classification task of determining if the people displayed in 2 images are the same or not.

6.5.2 Same emotion

Taking the experiments one step further we decided to train the network to detect if two emotions are the same. The network we have built is able to determine with high accuracy if 2 people are displaying the same emotion.

A legitimate question to ask would be: why do this? We already have a good classifier that is able to detect emotions. In order to see if two people display the same emotion we could train the classifier, classify both images and see if the predicted labels are the same. There are multiple reasons for which we decided to build this model and construct this experiment:

- **Robustness:** in the model specially designed to differentiate emotions adding another emotion does not require changing the network architecture, while doing so in the DBN recognition network would require changing the last softmax layer by increasing the number of units by 1. This requires reassessing the number of hidden layers and the number of hidden units in each layer. Changing the learning rates and momentum would require cross validation.
- The model we employ is probabilistic, it gives a meaningful probability that reflects its belief that the two displayed emotions are the same. Just checking if a classifier predicts the same label for both images does not define a confidence measure in the result. However, if the classifier used is probabilistic (such as DBN with a softmax layer), we can give a confidence score. Assuming that the events of the network making a mistake on two different images are independent, the confidence score can be defined as the product of the two classification probabilities given by the emotion recognition network. Nonetheless, the resulting accuracy will not be meaningful, because a deep belief net with high prediction accuracy (like the one described in subsection 6) tends to be very confident with in the results it gives, even if it makes a mistake. The product of such two numbers will always be very close to 1, providing almost no information.
- Train / test time is substantially decreased for this smaller network.
- As we used the same model to distinguish between subjects (subsection 6.5.1) we can compare the performance on the two tasks.

Experiment a

This experiment tests the ability of the network to distinguish between emotions if the input images represent the same subject.

Experiment b

No constraints on the subjects identity was set in this experiment. Example input images are shown in figure 6.11. It is interesting to note that the accuracy did not decrease substantially in this experiment.

The classification results obtain in these experiments can be seen in table 6.10. All reported results use rectified linear hidden units.

Experiment	Accuracy
Experiment a	92.6 %
Experiment b	90.5%

Table 6.10: Results from the classification task of determining if the people displayed in 2 images are the same or not.



Figure 6.11: Example of inputs presented when detecting if the emotions are the same. 7 pairs are shown, aligned vertically.

6.5.3 Same subjects, different emotions

A new experiment shows how the accuracy of the network we defined in subsection 6.5.1 depends on the emotions depicted by the subjects. We trained the network to differentiate between subjects, as before. However, now we are interested to see how different emotions affect the network's ability to determine if the two presented subjects are the same. So we test the network by giving it two images of the same subject but we vary the emotions, and record how the output probabilities change. As expected, when the two images presented to the network display the same emotion, the accuracy of the network increased. The predicted probability is also determined by how much an emotion requires change of the facial features. We notice this by looking at the average predicted probability when the two presented images depict a person in neutral expression and the average predicted probability when the two presented images depict a person who screams. This gives us a key insight to how to perform subject detection. If we have only two images and we want to determine if the subjects in the images are the same, we have to account for bias in case both subjects are widely smiling.

Table 6.11 provides numerical results obtained by training a network for 300 epochs, with 5 epochs for RBM training. We used sparsity constraints as we have defined them in 4.1.2, with the cost used being binary cross entropy. Both learning rates were 0.005. We see that the network is a lot less secure when the subject depicts different emotions (regardless of which pair of emotions, the probability is below 0.7) and more confident when there are the emotions are the same (always above 0.8).

First emotion	Second emotion	Average predicted probability
Neutral	Neutral	0.80
Neutral	Smile	0.68
Neutral	Scream	0.66
Smile	Smile	0.84
Smile	Scream	0.61
Scream	Scream	0.83

Table 6.11: Average predicted probabilities by a similarity network when it is presented with two instances of the same subject, but under different emotions.

7 | Implementation

As part of this project we provide a high performance open source library for deep belief networks³¹.

The code was implemented in the Python programming language. There were multiple factors taken into account when making this important decision:

- It has an excellent computations library, Numpy [56], that provides not only vast functionality, but also speed, as the Python code is translated and runs in C.
- It is highly used in the scientific community, making the project easily accessible for other researchers upon completion.
- Code is easy to write, read and understand.
- The Theano library [57] designed for machine learning allows you to write Python and compiles part of the code to CUDA, allowing increased speeds by using the GPU. Theano integrates tightly with Numpy, as its core data structure are Numpys *ndarrays*.

7.1 Setup

The experiments described have been performed with the following set up:

- Python version 2.7
- Numpy version 1.8
- OpenBlas [58], a numerical library which allows multi-threaded matrix operations
- Theano version 0.6 [57]
- Sklearn version 0.14.1 [59] a machine learning library used for cross validation.
- OpenCV version 3.0.0-dev [48], for face detection
- Matplotlib version 1.2.1 [60], for creating some of the plots presented in the report.
- Scitkit-image version 0.9.3 [61], for image processing.

³¹You can find it at <https://github.com/mihaelacr/pydeeplearn>

7.2 Runtime speed concerns

The initial experiments were performed using Python and Numpy, on a CPU. However, due to the computational demand required by training deep belief nets, the time spent on one experiment was becoming a problem. For example, training a 4 layered deep belief net with the MNIST dataset for 100 epochs of backpropagation took approximatively 15 hours.

The initial guess was that the issue is Python, which tends to be substantially slower than statically typed languages that compile to native code. A profiler analysis revealed that most of the time was not spent on Python code, but on matrix operations, which are defined using Numpy and hence are translated to C.

Options considered for speeding up the code were Cython [62], an extension of Python with embedded types which translates the code to C, and Theano, which lets you run code on a graphics card via CUDA. Due to the output of the profiler and the comparisons between the two shown in [57], we opted for Theano. The disadvantage of this choice was the time spent learning the tool: while Cython is easy to master for someone who has knowledge of Python and C, Theano has a much steeper learning curve, due to the need to define expression graphs which assemble program trees at run-time. Figures 7.1 and 7.2 show the difference between writing Python code with Numpy and Theano for matrix multiplication and raising a matrix to a given power.

Another measure taken to obtain speed up was a change in the underlying matrix operations library that Numpy and Theano are linked with. By default, they are linked against ATLAS [63], but other libraries such as MKL and GotoBlas2 have shown to perform better [64]. OpenBlas is a library built on top of GotoBlas2, giving a considerable speed up over ATLAS. OpenBlas allows multi-threaded operations, while ATLAS does not.

7.2.1 Theano

Theano is a library designed for fast mathematical computation, built by the Lisa lab at the University of Montreal³². The aim of the library is to provide a clear way of defining expression graphs that can be then translated into CUDA code. Theano combines aspects from an optimizing compiler and a computational algebra library.

As a computational library, Theano has multiple benefits. One that was particularly appealing for this project was the support for *symbolic variables* (which are required for creating expression graphs) and *symbolic differentiation*. Symbolic differentiation comes in handy when implementing backpropagation. Our initial CPU implementation required hard coding of the derivatives for each function. This has the obvious disadvantage of lack of flexibility: when the cost function of the network is changed, a new derivative has to be manually computed and coded. Symbolic differentiation also increased code clarity and removed clutter: the same functionality (the updating of the parameters) becomes clear, delegating to the library code. Moreover, symbolic differentiation is more numerically stable: finding out the error derivative at the first layer is just a matter of replacing the layer activations in the mathematical formula for the derivative of the

³²http://lisa.iro.umontreal.ca/index_en.html

error with respect to the layer weights and biases (the parameters of the model for that layer), instead of using the numerical propagated error from the layer above. This avoids propagation of floating point errors from the higher layers to the lower ones.

One of the key advantages of our implementation is the clean transparent interface that it provides to the user. Due to the learning overhead that Theano has, we have decided to not impose it on the user of our library. Hence the interface to the deep belief network that we provide is Theano-free. The input and output are required and given using Numpy arrays, and the user does not have to know or understand how to code on the graphics card.

```
import theano.tensor as T
from theano import function
from numpy.random import randint

x = T.matrix('x')
y = T.matrix('y')

sc = T.dot(x, y)
mydot = function([x,y], sc)

a = randint(0, 100, (1000, 1000))
b = randint(0, 100, (1000, 1000))
print mydot(a,b)
```

```
import numpy as np
from numpy.random import randint

a = randint(0, 100, (1000, 1000))
b = randint(0, 100, (1000, 1000))
print np.dot(a,b)
```

Figure 7.1: Comparison between Theano (left) and Numpy (right) code for matrix multiplication.

7.2.2 OpenBlas

We decided to link our Numpy and Theano implementation against OpenBlas, in order to be able to benefit from the multi threaded implementation. As expected, we found that this helps most when using the CPU code. When using Theano for the GPU, very little flow occurs on the processor (for example, a loop which delegates work to the GPU), so multi threading is of little help in that case.

Table 7.1 shows the speed improvements obtained by using the GPU graphics card Quadro 6000, and using Open Blas. Further experiments performed on a Titan Black graphics card showed another factor 3 improvement in speed.

```

k = T.iscalar("k")
A = T.vector("A")

def accumulate(prior_result, A):
    return prior_result * A

# Symbolically describe the result
result, updates = theano.scan(
    fn=accumulate,
    outputs_info=T.ones_like(A),
    non_sequences=A,
    n_steps=k)

final_result = result[-1]

# Compiled function returning  $A^k$ 
power = theano.function(
    inputs=[A, k],
    outputs=final_result,
    updates=updates)

```

```

result = 1
for i in xrange(k):
    result *= A

```

Figure 7.2: Comparison between Theano (left) and Numpy (right) raising to matrix power. Adapted from [65].

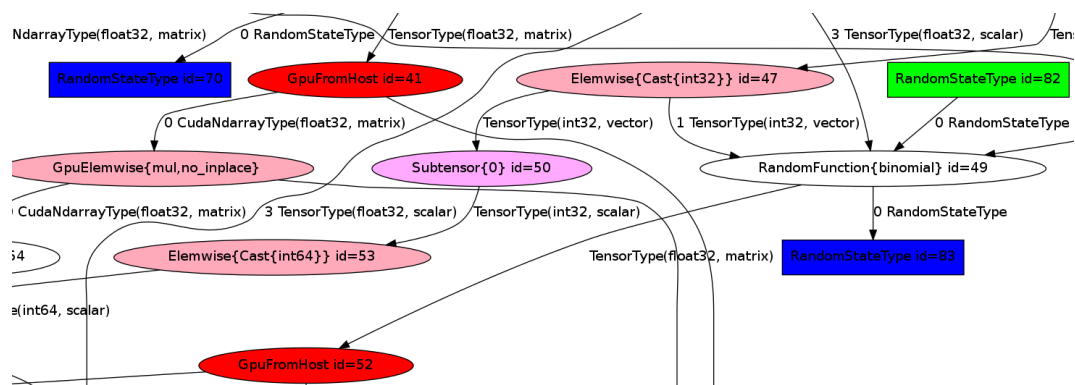


Figure 7.3: A part of the Theano expression graph of the training function used for the deep belief network implementation in this project.

Atlas	Open Blas	Open Blas threads	Theano	Time (hours:mins: seconds)
✓	✗	✗	✗	2:36:31
✗	✓	4	✗	2:11:02
✓	✗	✗	✗	8:53
✗	✓	1	✓	8:14
✗	✓	2	✓	7:56
✗	✓	3	✓	8:09
✗	✓	4	✓	11:34

Table 7.1: Speed comparison between multiple libraries when training a deep belief net with 5 layers with MNIST data. Training set size 10000. Testing set size 10. Experiments run on a GPU used a Quadro 6000 graphics card, and the ones on an Intel Core i7-2600 CPU with 3.40GHz, 8 cores and 8GB RAM. The experiments were performed by fixing the seed of the random generators, in order to accurately compare results. Due to the limited computation performed on the CPU, adding too many threads decreases performance, due to context switch overhead.

8 | Evaluation

We are now ready to analyse our work, having in mind other techniques, results and available resources.

Due to the relative novelty of the field and the rapid discovery of new methods, there are few complete references on deep learning. This report provides a vast description of techniques from the area, with particular aim in covering deep belief nets. Learning deep architectures for AI (2009, [31]) provides an excellent overview of the methods available, including auto encoders and convolutional networks, which are not touched upon here. However, since its publication multiple approaches have been proven to improve classification accuracy and speed of training, including dropout, rectified linear units and Rmsprop (which can also be applied when training convolutional neural nets). We have covered them here combining both mathematical intuition and implementation pragmatism.

8.1 Handwritten digits

Section 5 described the experiments we have performed for hand written digit classification on the MNIST dataset. This is the standard benchmark used for testing and comparing machine learning algorithms, so numerous results are available. Table 8.1 provides a summary of techniques used for this task and the results that were obtained.

Technique	Error
Linear classifier (1-layer NN)	12.0 %
40 PCA + quadratic classifier	3.3 %
K-nearest-neighbours, L3 norm distance	2.83 %
Estimated human performance	2%
Our best performance	1.1%
NN, 784-500-500-2000-30 + nearest neighbour, RBM + NCA training	1.0 %
Maxout nets	0.94 %
Deep convex net, unsup pre-training	0.83 %
Virtual SVM, deg-9 poly, 1-pixel jittzered	0.68 %
Large convolutional net, unsup pre-training	0.53 %
Convolutional maxout, dropout	0.45 %

Table 8.1: Classification results obtained on the MNIST dataset by various techniques. Results which involve no preprocessing, in order to be able compare with our results. The values were obtained from the MNIST official page and [7]. The estimated human accuracy is obtained from [66].

8.2 Emotion recognition

One of the main goals of this thesis was to perform emotion recognition from images. We have achieved a performance of **99.3%** on the Multi PIE dataset, with **91.7%** when the sets of training and testing subjects are disjoint. We will now evaluate other techniques used for facial emotion recognition and report their performance.

Support Vector Machines can be successfully used for discrimination after features have been learned with convolutional neural nets [67]. This method was used to win a Kaggle competition³³ on emotion recognition from images. Details about the dataset used in the competition can be found in [55]. The authors describe how back propagating the error with respect to the cost of a L_2 Linear SVM can outperform using a softmax layer as a discriminative method.

Local binary patterns have been used for feature extraction from faces of images, in conjunction with support vector machines for discrimination [6]. This model was trained and tested on the Multi Pie dataset. The images in the reported experiments are not aligned and the face is detected using the Viola Jones classifier. The training and test sets were divided in a 4 to 1 ratio and testing data is taken from subjects that were not present in the training data. For our experiments, the Multi PIE images were aligned before used for feature extraction and classification. We obtained an accuracy of **91.7%**. Moreover, the dataset we used only has 5 poses, instead of 7 (we did not use the 75% and 90% rotation angles). The emotion recognition accuracy for each of the 7 poses is reported in the paper, with the best one being 87.5%, for the 15% angle pose.

A new method is proposed in [68]: a deep network that uses a convolutional layer and a max pooling layer to detect over complete representations from which they defined receptive fields. The receptive fields are filtered in order to minimize the common information between them. The remaining receptive fields are fed into a stack of RBMs, and the resulting features are classified using linear SVMs. The reported accuracy on the extended Cohn Kanade database³⁴ (CK+) is 92%. While this method does not directly use FACS action units, the authors claim that the receptive fields in their model is consistent with the interpretation of FACS.

A method based on head-pose optimisation has been evaluated on the Multi-PIE dataset [69]. The technique used requires multiple steps: first, linear discriminant analysis (LDA -subsection 4.1.4 in [15]) and mixture of Gaussians (GMM) probabilistically determine the pose in the image. Then, Coupled Scaled Gaussian Process Regression (CSGPR) [69] is used to map angled poses to the frontal pose. From the frontal pose the 39 facial features are manually extracted and set as input to a support vector machine that classifies emotions. The results reported on the Multi PIE dataset use 50 subjects and 4 poses and detect only 4 emotions (surprise, disgust joy and neutral). We used 147 subjects, 5 poses and 6 emotions. This method has the advantage of being able to detect emotions from poses it was not exposed during training, highly improving generalization in unconstrained environments.

Table 8.2 gives an overview of the methods described for emotion recognition, together with the obtained classification results.

³³<http://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/>

³⁴This dataset is a superset of the one we used in subsection 6.2.

Dataset	Feature extraction technique	Classification technique	Nr. emotions	Ref.	Accuracy
Kaggle	Convolutional NNs	SVM	7	[67]	71.2 %
Multi PIE	Local binary patterns	SVM	6	[6]	80.17%
Multi PIE	LDA, GMM, Gaussian Process Regression, Manual point extraction	SVM	4	[69]	94.8%
Cohn Kanade+	AU-aware deep networks	SVM	7	[68]	92.05%

Table 8.2: Overview of techniques for emotion recognition from images and their results on different datasets. Details about each method found in text.

8.2.1 Maxout nets

In order to compare our results we decided to employ a new technique in the field of neural networks and compare it with our results. This technique is described in detail in [7], and has been show to give good results in different well known tasks (including 0.94% error on the MNIST permutation invariant task). Maxout nets are designed to be used with dropout, as it ensures that dropout acts as model averaging. This new type of neural net introduces a new activation function, which can be seen as the extension of rectified linear units: each hidden unit h is taken to be as the maximum of a set of multiple intermediate units (denoted here by z) (which is the same for each hidden unit).

$$h_i = \max_{j \in [1, k]} z_{ij} \quad (8.1)$$

$$z_{ij} = \sum_m x_m W_{mij} + b_{ij} = x W_{...ij} + b_{ij} \quad (8.2)$$

where by $W_{...ij}$ we mean the vector of size of the input obtained by accessing the matrix $W \in \mathbb{R}^{m \times n \times k}$ at the second coordinate i and third coordinate j . The number of intermediate units (k) is called the number of pieces used by the maxout net. A pictorial representation of a maxout layer is denoted in figure 8.1. Maxout nets also use a softmax layer on top for classification. As described in subsection 2.6, a weight constraint can be imposed on the L_2 norm of the incoming weight vector of a hidden unit, for regularization. This technique is also often used with maxout nets, as described in [7]. A heuristic for setting the norm constraint is to monitor the average norm during training (without constraint) and then set the constrained norm to be 80% of that average.

The authors of maxout nets maintain the library `pylearn2` [70] and have open sourced their code. We used it to build and train maxout nets for emotion recognition. As far as we know this is the first application of maxout nets for facial emotion recognition from images. Table 8.3 describes the results we obtained together with the architectures used. All the experiments used a learning rate of 0.1 and trained the network with mini-batch gradient descent with mini-batches of size 100.

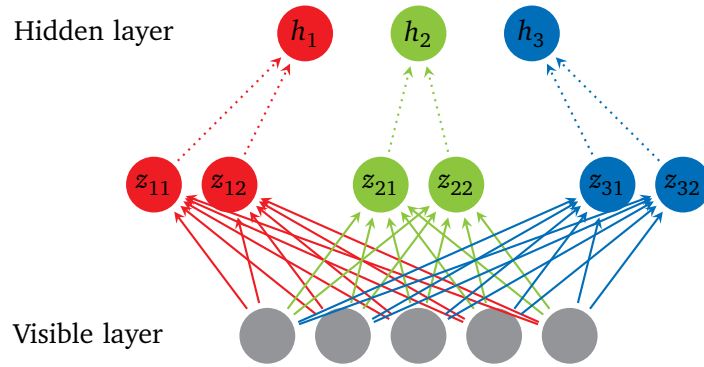


Figure 8.1: A maxout net with 5 visible units and 3 hidden units and 2 pieces for each hidden unit.

Architecture	Number of pieces	Momentum	Max norm constraint	Accuracy
400, 400, 400	3	✗	✗	94.1%
500, 500, 500	3	✗	✗	95.4%
700, 700, 700	3	✗	✗	94.6%
500, 500, 500	3	✗	1.0	94.1%
500, 500, 500	3	✓	✗	89.0%

Table 8.3: Accuracy of maxout nets when classifying the emotions in the Multi Pie dataset.

8.2.2 Commercial applications

Due to the high applicability of emotion recognition, new technology companies have been built around the concept of providing clients with the ability of detecting emotions from *video*. We are aware of Emotient (US)³⁵ and Realeyes (UK)³⁶. Emotient focuses on detecting a wide range of emotions, from basic ones to more advanced (such as frustration and confusion) as well as combinations of emotions. Realeyes uses action units [3] to determine the emotions from live web cam recordings.

8.3 Similarities

8.3.1 Same subjects

The experiments detailed in subsection 6.5.1 are performed in the spirit of the ones described in [43]. Their best reported classification results achieve a performance of 81% with unsupervised pre-training and rectified linear units. There are some substantial differences in how our experiments are performed and we would like to clarify them here. We compare our results obtained with

³⁵<http://www.emotient.com/products>

³⁶<http://www.realeyesit.com/emotions>

experiment b, in which the network does not see a subject used for testing during training, as in [43]. Table 8.4 summarises these differences and compares accuracies.

The input data

The database used in our experiments is Multi PIE. In their experiments they used Labelled Faces in the Wild, a database of 13233 images of 5749 people. Exposure to a substantially bigger number of subjects (5749 vs 147) can increase performance, because the network has more instances of different subjects from which to learn. Our experiments present the network with 147 subjects in different poses, rather than different subjects. Our input data are black and white 40×30 images, while their data is coloured, and has dimension: $32 \times 32 \times 3$.

Hidden units

The best values report in their experiments use 4000 rectified linear units. We used 1000 sigmoid units.

Preprocessing

Our inputs are aligned, but the faces used in the experiments in [43] are not pre aligned.

Data splits

The data splits we used are random in order to keep a train to test ratio 4:1. The splits used in the experiments described by [43] are predefined by the designers of the dataset, into 10 splits of 5400 training pairs and 600 test pairs.

	Ours	Theirs
Data size	40×30	$32 \times 32 \times 3$
Nr. hidden units	1000	4000
Hidden units	Sigmoid	ReLU
Aligned faces	✓	✗
Train to test ratio	4	9
Total nr. subjects	147	5749
Number of pre-training epochs	50	300
Accuracy	82.5%	81%

Table 8.4: Comparison of experiments performed for detecting if two images represent the same person.

8.4 Implementation

As part of this project, a Python library with focus on deep belief nets was created. We named it **pydeeplearn**. We have to assess it against other available Python libraries, such as:

- Sklearn [59]
- Theano [57]
- Pylearn2 [70]
- Nolearn [71]

Sklearn

One of the most popular python machine learning libraries available, Sklearn offers a variety of algorithms, as well as useful utilities such as cross validation, confusion matrices and precision tables. While Sklearn has an implementation of RBMs, is not a complete or up to date one: it does not have support for real valued units, dropout or Rmsprop.

Theano

Theano is a machine learning library with GPU support (discussed at length in subsection 7). While there are available tutorials of how to build a DBN with it, there is no modular support in the library for higher level models. Theano can be seen as a building block for other libraries and implementations (such as the one presented here and Pylearn2).

Pylearn2

Pylearn2 is a library made by researchers, for researchers. While it does not offer an implementation of a Deep belief net (but contains up to date implementations of convolutional nets, multi-layer perceptrons and maxout neural nets [7]), one can be constructed from the functionality it provides. Using Pylearn2 requires knowledge of Theano and extensive knowledge of Pylearn2 itself. This contrasts to one of our goals: build an easy to use interface to DBNs and RBMs, that does not ask the user to know GPU programming via Theano .

Nolearn

Nolearn provides an easy to use and simple interface to a deep belief network that runs fast on the GPU. However, it does not provide numerous features our implementation (pydeeplearn) does. Nolearn does not allow specifying unlabelled data that can be used for pre-training, hence depriving the user of one of the main benefits of deep belief nets: increased performance with pre-training on a bigger dataset than the available labelled instances. Since Nolearn does not depend on Theano, it does not benefit from some of its key resources, such as symbolic differentiation. Symbolic differentiation allows us to have flexible arguments to both RBMs and DBNs: we can specify any activation function, without having to manually compute any derivatives. That is why instead of hardcoding the types of units with binary flags (such as having a flag that determines if rectified linear units should be used), we have arguments that specify the activation functions. This feature has helped us experiment with new activations functions for discriminative training, such as the expected

value of a noisy rectified linear unit (as described in appendix D). Our library also provides interruptible training: a keyboard interrupt will stop training at the current stage and start testing³⁷. This feature came in handy during the experiments described in this paper. A comprehensive comparison is reported in table 8.5.

Library feature	pydeeplearn	nolearn
Allow unlabelled data for unsupervised training	✓	
RBM module	✓	
Rmsprop	✓	
L_1 weight decay	✓	
Early stopping options	✓	
Activation functions as arguments	✓	
Sparsity regularization for pre-training	✓	
Dropout for pre-training	✓	
Nesterov momentum for pre-training	✓	
Different momentum update functions	✓	
Flexible arguments	✓	
Interruptible training	✓	
Dropout for fine tuning	✓	✓
Momentum	✓	✓
Rectified linear units	✓	✓
Fast computation via GPU usage	✓	✓
L_2 weight decay training	✓	✓
L_2 weight decay pre-training	✓	✓
API compatibility with sklearn	✓	✓
Different learning rates for different layers		✓
Decaying learning rate		✓
Supervised weights scale		✓
Pre train call back		✓
Train call back		✓

Table 8.5: Comparison between the library presented in this paper (**pydeeplearn**) and **nolearn**. Details found in text.

³⁷If the user wants to abort the program completely, they can do so with another keyboard interrupt

9 | Future work

Our investigation into emotion recognition using deep learning is far from being over. We need to be able to create models that work better in the “wild”, giving a high accuracy on any image depicting a human face, without confining the subject into an artificial setting (like the images which come from most databases). Our model performs best when the input faces are aligned. This limitation can be solved by using convolutional neural networks, which achieve translational invariance.

The field of deep learning is advancing fast, creating more accurate and robust models that can be adapted and enhanced to our problem statement. Keeping this in mind, we propose various directions in which the work presented here can be expanded and improved.

9.1 Improving the current model

In this paper we have focused on a special technique from the deep learning family: deep belief networks and we have presented an up to date implementation of this model. However, there are further improvements that can be brought to our current implementation. These would be useful mainly for automating some of the tedious tasks that come with training a neural network, namely finding the right parameters:

- Adaptive learning rates (using methods such as Yan LeCun’s recipe in [20].)
- Better methods for early stopping evaluation.
- Use second order methods for optimization, instead of plain gradient descent.
- Model averaging on different architecture of the current model (different number of layers, number of units per layer, etc.)

9.2 Data preprocessing

One of the lessons we have learned with this paper is the importance of data preprocessing before classification. Further preprocessing could be helpful, under the constraint that it can be done online (as opposed to requiring all available data at once) and in reasonable speeds (having in mind a real time application). We have seen that equalization performs an important role in determining the accuracy of our classifiers. One of the methods used is Contrast Limited Adaptive Histogram Equalization (CLAHE), has multiple parameters that need can be changed to improve performance. Due to time constraints, we did not pursue that further, but used the default options coming with the computer vision library used. Another option for preprocessing could be extracting action units from the images (using the method described in [5]). We could then append the action units at the end of the image vector, allowing the network to benefit from the information

of the localized features as well. Similar techniques to the Viola Jones classifier can be used to detect eyes and mouths, allowing the network to benefit from more specialized information.

9.3 Different models for unsupervised pre-training

We propose trying two other models for emotion recognition: convolutional deep belief nets and deep Boltzmann machines. We chose these models as they both allow learning features from data in an unsupervised fashion.

Convolutional RBMs and Convolutional deep belief nets

Convolutional RBMs [72] are an extension of RBMs that rely on convolution and weight sharing, as they are inspired by convolutional neural networks. Convolutional RBMs can also be stacked together in order to form a deeper model, called Convolutional Deep belief nets. Unlike deep belief nets, Convolutional DBNs cannot be trained using backpropagation, as they are solely a generative model. The usual classification method feeds the input image into the network and the last learned layers of features into a support vector machine (SVM). An advantage of using this method comes from the flexibility of the input shape: they allow the network visible layer to be a matrix, instead of a flattened vector. This enables the network to learn from the spacial proximity of pixels, improving performance. Using convolutional RBMs also give the benefit from more robust feature learning: the performance of the network will be less affected if the input images are not aligned.

Deep Boltzmann machines

Deep Boltzmann machines [73] are a very similar model to deep belief nets: they also work by greedy layer-wise pre-training using Restricted Boltzmann machines, with modifications on how the pre-training is performed for the top and bottom layers. Unlike DBNs, they are undirected probabilistic models. A softmax layer can be added on top of the Deep Boltzmann machine and then discriminative fine tuning can be performed as usual ³⁸.

9.4 Emotion recognition from video

We are interested in improving our model to work in real life conditions and also be competitive against other implementations. We plan on entering the Facial expression in the wild challenge ³⁹ to compare how our model does in an unconstrained environment. In order to do so we need an end to end robust pipeline of preprocessing and classification, that needs to be designed for coping with noise. Part of the challenge is also to do emotion recognition from videos. Recently it has been shown [74] that deep belief nets can be successfully used to detect emotions from multi-modal data. We plan on augmenting our model to cope with audio and image sequences as input data.

³⁸The authors of the papers describe a model in which the last layer of features are augmented to the visible layer in order to improve classification performance.

³⁹For details of the challenge see: <http://cs.anu.edu.au/few/ChallengeDetails.html>

10 | Conclusion

The presented work proves that deep belief nets are a good model for performing facial emotion recognition. We have shown that unsupervised pre-training in generative neural networks provides an excellent framework for feature extraction from images of faces. We assessed classification performance on three different labelled datasets. The first dataset, Cropped Cohn-Kanade, contains a small number of images which depict people from frontal position. The MultiPIE dataset contains a large number of images of subjects captured in different poses and illumination conditions. Finally, we used a medium sized dataset of unaligned images from a Kaggle competition. We obtained state of the art results on all three datasets. Part of our success is due to new techniques such as dropout and rectified linear units.

Through a novel experiment, we created a probabilistic model that is able to distinguish if two people are displaying the same emotion, without requiring emotion classification. This work has shown that emotion plays an important role in recognizing if two images depict the same person, giving new insight of how face matching should be performed.

Our empirical results show that input preprocessing (face alignment, face detection and histogram equalization) still have a crucial role in emotion recognition. Machine learning algorithms need to become more robust to noise and perform better in an unconstrained environment, either by incorporating these preprocessing techniques or by creating more powerful models.

On the theoretical side, we have extended two types of sparsity constraints for Restricted Boltzmann machines to work with noisy rectified linear units, and have provided guidelines for when and how to use them.

Through our high performance open source implementation, we fill a gap in the community for an accessible and complete software library for deep belief nets and Restricted Boltzmann machines.

A | Notation

When a variable is written with bold face (\mathbf{x}), it is to represent that it is a vector.

Standard notation

We adopt the standard conventions and notations as follows:

- Throughout the report, we use the names gradient and derivative interchangeably.
- Unless otherwise stated, $\sigma(x)$ represents the standard logistic function.
- $\|x\|$ denotes the L_2 norm of x .
- $N(\mu, \sigma^2)$ is Gaussian distribution with mean μ and variance σ^2 .
- $\text{GaussCDF}(x|\mu, \sigma^2)$ is the cumulative distribution function at value x for the Gaussian random variable with mean μ and variance σ^2 .
- KL stands for the Kullback-Lieber divergence between two distributions.
- H_P denotes the entropy of a probability distribution P .

Neural networks specifics

In the discussion about neural networks, activations and structure we use:

- N denotes the size of the training set.
- x denotes a input data vector
- y denotes the activity of a neuron
- z denotes the total linear input of a neuron receives.
- t denotes the target vector associated with an instance.

Abbreviations

Throughout this report we use various abbreviations, summed up in the table below.

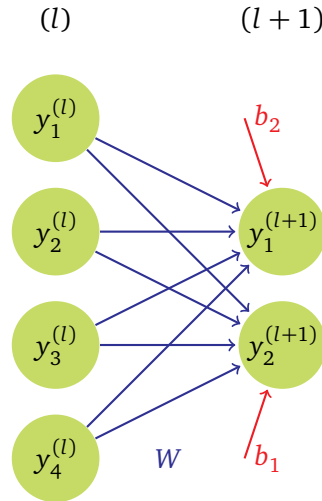


Figure A.1: A layer in a neural network. Used to exemplify the notation in this thesis.

Abbreviation	Meaning
NN	Neural Network
ANN	Artificial Neural Network
MSE	Mean square error
DBN	Deep Belief Network
RBM	Restricted Boltzmann Machine
CD	Contrastive Divergence
LDA	Linear Discriminant Analysis
GMM	Gaussian Mixture Models
PCD	Persistent Contrastive Divergence
AIS	Annealed Importance Sampling
SVM	Support Vector Machine
ReLU	Rectified Linear unit

Table A.1: Abbreviations used in this thesis.

B | Deriving the activations of a unit in a RBM

This section does the derivations which lead to the activation probabilities defined in equations 3.13 and 3.14, based on [75].

We will do the derivation for a visible unit, but since the Restricted Boltzmann machine is a symmetric network, a similar proof can be done for a hidden unit.

Consider a Restricted Boltzmann Machine with m visible units and n hidden units. Take unit v_l .

$$\begin{aligned}
 E(v, h) &= - \sum_i a_i v_i - \sum_i b_i h_i - \sum_i \sum_j w_{ij} v_i h_j \\
 &= -a_l v_l - \sum_j w_{lj} v_l h_j - \sum_{i \neq l} a_i v_i - \sum_i b_i h_i - \sum_{i \neq l} \sum_j w_{ij} v_i h_j \\
 &= -v_l \underbrace{\left(a_l + \sum_j w_{lj} h_j \right)}_{z_l} - \underbrace{\sum_{i \neq l} a_i v_i - \sum_i b_i h_i - \sum_{i \neq l} \sum_j w_{ij} v_i h_j}_{\alpha_l} \\
 &= -v_l z_l + \alpha_l
 \end{aligned}$$

Denote the set of visible units without unit v_l by v_{-l} . By using the conditional independence of visible units:

$$\begin{aligned}
 p(v_l = 1|h) &= p(v_l = 1|v_{-l}, h) = \frac{p(v_l = 1, v_{-l}, h)}{p(v_{-l}, h)} \\
 &= \frac{p(v_l = 1, v_{-l}, h)}{p(v_l = 0, v_{-l}, h) + p(v_l = 1, v_{-l}, h)} = \frac{e^{-E(v_l=1, v_{-l}, h)}}{e^{-E(v_l=1, v_{-l}, h)} + e^{-E(v_l=0, v_{-l}, h)}} \\
 &= \frac{e^{-(-1z_l + \alpha_l)}}{e^{-(-0z_l + \alpha_l)} + e^{-(-1z_l + \alpha_l)}} = \frac{e^{-\alpha_l} e^{1z_l}}{e^{-\alpha_l} e^{0z_l} + e^{-\alpha_l} e^{1z_l}} \\
 &= \frac{e^{z_l}}{1 + e^{z_l}} = \frac{1}{1 + e^{-z_l}} = \sigma(z_l)
 \end{aligned}$$

This justifies the usage of the logistic sigmoid function as the activation function used in stochastic binary RBMs.

The same approach can be taken when using Gaussian units, by changing the energy function to the one in equation 3.25.

C | Why greedy learning works: detailed mathematical explanation

We will now give a more detailed explanation of the mathematical intuition behind the greedy pre-training in DBNs. DBNs are generative models that try to increase the likelihood of obtaining the observed data, $P(x)$.

We can rewrite the log probability of the data as follows:

$$\begin{aligned}
 \log P(x) &= \left(\sum_{h_1} Q(h_1|x) \right) \log P(x) \\
 &= \sum_{h_1} Q(h_1|x) \log \frac{P(x, h_1)}{P(h_1|x)} \\
 &= \sum_{h_1} Q(h_1|x) \log \frac{P(x, h_1) Q(h_1|x)}{P(h_1|x) Q(h_1|x)} \\
 &= \sum_{h_1} Q(h_1|x) \log Q(h_1|x) + \sum_{h_1} Q(h_1|x) \log P(x, h_1) + \sum_{h_1} Q(h_1|x) \log \frac{Q(h_1|x)}{P(h_1|x)} \\
 &= H_{Q(h_1|x)} + KL(Q(h_1|x) || P(h_1|x)) + \sum_{h_1} Q(h_1|x) \log P(x, h_1) \\
 &= H_{Q(h_1|x)} + KL(Q(h_1|x) || P(h_1|x)) + \sum_{h_1} Q(h_1|x) (\log P(x|h_1) + \log P(h_1))
 \end{aligned}$$

By network construction, $Q(x|h_1) = P(x|h_1)$ (when the model is given h_1 it only uses the first RBM to generate v). Another way of observing this is by looking at the distribution modelled by a DBN given in equation 3.37.

It is important to note that $Q(h_1|x)$ and $P(h_1|x)$ are generally different distributions, because the upper layers of the DBN impose a different prior on h_1 , affecting $P(h_1|x)$. However, $Q(h_1|x)$ is only affected by the parameters of the first layer RBM. However, under certain circumstances the two distributions are equal, making the KL term null.

We will now illustrate that if the second RBM is initialized such that its weights are the transposed of the weights of the first RBM (ie. $W_2 = W_1^T$)⁴⁰, and if the biases of the hidden units are initialized to the biases of the visible units (and conversely for the visible units) for the first RBMs then the two distributions are the same. This easily follows from the symmetry of a Restricted Boltzmann Machine: the probability of the second RBM to generate the values for h_1 . Under this

⁴⁰This also adds the constraint that the number of hidden units of the second RBM has to be equal to the number of visible units of the first RBM

initialization, the RBMs model the same distributions (because an RBM does not care what we call the *visible* and *hidden* layers).

Under the assumption, when using the second RBM, we want to optimize:

$$H_{Q(h_1|x)} + \sum_{h_1} Q(h_1|x) (\log(P(x|h_1) + \log P(h_1))) \quad (\text{C.1})$$

$H_{Q(h_1|x)} + \sum_{h_1} Q(h_1|x) \log(P(x|h_1))$ do not depend on the second RBM, but only on the first one: the first term is the entropy of the conditional distribution modelled by the first RBM of the hidden units given the visible units, and the second term can be written as $\sum_{h_1} Q(h_1|x) \log Q(x|h_1)$, by using $Q(x|h_1) = P(x|h_1)$.

Hence, when we want to optimize with respect to the second RBM, we only use:

$$\sum_{h_1} Q(h_1|x) \log P(h_1) \quad (\text{C.2})$$

This is equivalent to training the second RBM, using as input the hidden activations of the observed data produced by the first level RBM.

Since the *KL* divergence measure is always positive and we started with it at 0, during training of the second RBM it can only increase. At the same, $H_{Q(h_1|x)} + \sum_{h_1} Q(h_1|x) \log P(x|h_1)$ will not change, as it does not depend on the second RBM, so we can only improve on our maximum likelihood estimation during training.

D | Expected value of a noisy rectified unit

We remind the reader that the noisy rectified linear units have the following activation function:

$$f(x) = \max(0, x + \mathbf{N}(0, \sigma(x))) = \max(0, \mathbf{N}(x, \sigma(x))) \quad (\text{D.1})$$

Where σ is the logistic sigmoid.

Looking at the cost of sparsity, we see that we have to be able to compute $\mathbb{E}(h_j^l | v^l)$.

$$\mathbb{E}(f(x)) = \mathbb{E}(\max(0, \mathbf{N}(x, \sigma(x)))) \quad (\text{D.2})$$

Let's compute the $\mathbb{E}(\max(0, \mathbf{N}(a, b)))$, for any real numbers a and b with $b > 0$.

$$\mathbb{E}(\max(0, \mathbf{N}(a, b))) = \int_{-\infty}^0 0 \mathbf{N}(x|a, b) dx + \int_0^{\infty} x \mathbf{N}(x|a, b) dx = \int_0^{\infty} x \mathbf{N}(x|a, b) dx \quad (\text{D.3})$$

Now we do a change of variable so that we move into using the standard normal:

$$x \rightarrow (x - a)/\sqrt{b}$$

$$\int_0^{\infty} x \mathbf{N}(x|a, b) dx = \quad (\text{D.4})$$

$$\int_0^{\infty} x \frac{1}{\sqrt{2\pi b}} e^{-\frac{(x-a)^2}{2b}} dx = \quad (\text{D.5})$$

$$\int_0^{\infty} (\sqrt{b}y + a) \frac{1}{\sqrt{2\pi b}} e^{-\frac{y^2}{2}} dy = \quad (\text{D.6})$$

$$\int_0^{\infty} (\sqrt{b}y + a) \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} \frac{1}{\sqrt{b}} dy = \quad (\text{D.7})$$

$$\int_0^{\infty} (\sqrt{b}y + a) \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} \frac{dy}{dx} dx = \quad (\text{D.8})$$

$$\int_{-\frac{a}{\sqrt{b}}}^{\infty} (\sqrt{b}y + a) \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} dy = \quad (\text{D.9})$$

$$\frac{\sqrt{b}}{\sqrt{2\pi}} \int_{-\frac{a}{\sqrt{b}}}^{\infty} y e^{-\frac{y^2}{2}} dy + \frac{a}{\sqrt{2\pi}} \int_{-\frac{a}{\sqrt{b}}}^{\infty} e^{-\frac{y^2}{2}} dy = \quad (\text{D.10})$$

$$\frac{\sqrt{b}}{\sqrt{2\pi}} \int_{-\frac{a}{\sqrt{b}}}^{\infty} y e^{-\frac{y^2}{2}} dy + a \left(1 - \text{GaussCDF} \left(-\frac{a}{\sqrt{b}} \middle| 0, 1 \right) \right) = \quad (\text{D.11})$$

$$\frac{\sqrt{b}}{\sqrt{2\pi}} \int_{-\frac{a}{\sqrt{b}}}^{\infty} -\left(-\frac{y^2}{2}\right)' e^{-\frac{y^2}{2}} dy + a \text{GaussCDF} \left(\frac{a}{\sqrt{b}} \middle| 0, 1 \right) = \quad (\text{D.12})$$

$$-\frac{\sqrt{b}}{\sqrt{2\pi}} \int_{-\frac{a}{\sqrt{b}}}^{\infty} \left(e^{-\frac{y^2}{2}}\right)' dy + a \text{GaussCDF} \left(\frac{a}{\sqrt{b}} \middle| 0, 1 \right) = \quad (\text{D.13})$$

$$-\frac{\sqrt{b}}{\sqrt{2\pi}} \left(e^{-\frac{y^2}{2}} \right) \Big|_{-\frac{a}{\sqrt{b}}}^{\infty} + a \text{GaussCDF} \left(\frac{a}{\sqrt{b}} \middle| 0, 1 \right) = \quad (\text{D.14})$$

$$-\frac{\sqrt{b}}{\sqrt{2\pi}} \left(-e^{-\frac{\left(-\frac{a}{\sqrt{b}}\right)^2}{2}} \right) + a \text{GaussCDF} \left(\frac{a}{\sqrt{b}} \middle| 0, 1 \right) = \quad (\text{D.15})$$

$$\frac{\sqrt{b}}{\sqrt{2\pi}} e^{-\frac{a^2}{2b}} + a \text{GaussCDF} \left(\frac{a}{\sqrt{b}} \middle| 0, 1 \right) \quad (\text{D.16})$$

$$\frac{\sqrt{b}}{\sqrt{2\pi}} e^{-\frac{a^2}{2b}} + a \text{GaussCDF} \left(\frac{a}{\sqrt{b}} \middle| 0, 1 \right) \quad (\text{D.17})$$

Going back to our original problem, we replace a by x and b by $\sigma(x)$, where x is the linear input received by unit h_j from the layer of visible units \mathbf{v} : $\sum_{i=1}^N w_{ij} v_i + b_j$.

$$\mathbb{E}(h_j^l | v^l) = \frac{\sqrt{\sigma(x)}}{\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma(x)}} + x \text{GaussCDF} \left(\frac{x}{\sqrt{\sigma(x)}} \middle| 0, 1 \right) \quad (\text{D.18})$$

Index

Annealed Importance Sampling, 29

Backpropagation, 8

Cohn-Kanade, 56

Contrastive divergence, 27

Contrastive wake sleep, 39

Cross entropy, 17

Deep belief networks, 34

Dropout, 21

Early stopping, 19

Free energy, 30

Full-batch learning, 11

Gaussian units, 31

Gradient descent, 11

Histogram equalization, 56

Hopfield net, 23

Logistic sigmoid, 6

Markov Random Fields, 25

Maxout nets, 79

Mini-batch learning, 11

MNIST, 49

Model averaging, 19

Momentum, 12

Multi PIE, 59

Nesterov Momentum, 13

Online learning, 11

Persistent Contrastive divergence, 28

Reconstruction error, 29

Rectified linear units, 32

Restricted Boltzmann machine, 25

Rmsprop, 14

Rprop, 13

Softmax, 15

Sparse hidden units, 33

Support Vector Machine, 38

Theano, 73

Unsupervised pre-training, 35

Viola Jones classifier, 56

Weight decay, 19

References

- [1] J. S. Garofolo, L. D. Consortium, *et al.*, *TIMIT: acoustic-phonetic continuous speech corpus*. Linguistic Data Consortium, 1993.
- [2] A. Martinez and S. Du, “A model of the perception of facial expressions of emotion by humans: research overview and perspectives,” *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 1589–1608, 2012.
- [3] P. Ekman and W. V. Friesen, “Measuring facial movement,” *Environmental Psychology and Nonverbal Behavior*, vol. 1, no. 1, pp. 56–75, 1976.
- [4] M. Valstar and M. Pantic, “Fully automatic facial action unit detection and temporal analysis,” in *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW’06. Conference on*, IEEE, 2006, pp. 149–149.
- [5] J. M. Susskind, G. E. Hinton, J. R. Movellan, and A. K. Anderson, “Generating facial expressions with deep belief nets,” *Affective Computing, Emotion Modelling, Synthesis and Recognition*, pp. 421–440, 2008.
- [6] S. Moore and R. Bowden, “Local binary patterns for multi-view facial expression recognition,” *Computer Vision and Image Understanding*, vol. 115, no. 4, pp. 541–558, 2011.
- [7] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, “Maxout networks,” *arXiv preprint arXiv:1302.4389*, 2013.
- [8] W. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943. [Online]. Available: <http://link.springer.com/article/10.1007/BF02478259>.
- [9] F. ROSENBLATT, “The perceptron: a probabilistic model for information storage and organization in the brain,” *Psychological review*, vol. 65, pp. 386–408, 1958, ISSN: 0033-295X. DOI: [10.1037/h0042519](https://doi.org/10.1037/h0042519).
- [10] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. 1969, vol. 165, p. 258, ISBN: 0262631113. DOI: [10.1109/T-C.1969.222718](https://doi.org/10.1109/T-C.1969.222718). [Online]. Available: <http://cdsweb.cern.ch/record/114106>.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, 1998, ISSN: 0018-9219. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791).
- [13] G. Hinton, S. Osindero, and Y. Teh, “A fast learning algorithm for deep belief nets,” *Neural computation*, 2006. [Online]. Available: <http://www.mitpressjournals.org/doi/abs/10.1162/neco.2006.18.7.1527>.
- [14] G. Dahl, T. Sainath, and G. Hinton, “Improving Deep Neural Networks for LVCSR using Rectified Linear Units and Dropout,” *Proc. ICASSP*, 2013. [Online]. Available: http://www.cs.toronto.edu/~gdahl/papers/reluDropoutBN_icassp2013.pdf.

- [15] C. M. Bishop, *Pattern Recognition and Machine Learning*, M Jordan, J Kleinberg, and B Schölkopf, Eds., ser. Information science and statistics 4. Springer, 2006, vol. 4, p. 738, ISBN: 9780387310732. DOI: [10.1117/1.2819119](https://doi.org/10.1117/1.2819119). arXiv:[0-387-31073-8](https://arxiv.org/abs/0-387-31073-8).
- [16] Y LeCun, L Bottou, G. Orr, and K. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*, 1998. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-49430-8_2.
- [17] H. Schulz, A Müller, and S. Behnke, “Investigating convergence of restricted boltzmann machine learning,” ... *2010 Workshop on Deep Learning* ..., no. December, pp. 1–9, 2010. [Online]. Available: http://www.ais.uni-bonn.de/~schulz/papers/nips10ws_schulz_mueller_behnke.pdf.
- [18] I Sutskever, J. Martens, G Dahl, and G Hinton, “On the importance of initialization and momentum in deep learning,” *cs.utoronto.ca*, no. 2010, 2012. [Online]. Available: http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf.
- [19] T Tieleman and G Hinton, “Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*, vol. 4, 2012.
- [20] T. Schaul, S. Zhang, and Y. LeCun, “No More Pesky Learning Rates,” *Journal of Machine Learning Research*, vol. 28, pp. 343–351, 2013. arXiv:[arXiv:1206.1106v2](https://arxiv.org/abs/1206.1106v2).
- [21] G. E. Nasr, E. Badr, and C Joun, “Cross entropy error function in neural networks: forecasting gasoline demand,” in *FLAIRS Conference*, 2002, pp. 381–384.
- [22] J. Moody, S. Hanson, A. Krogh, and J. A. Hertz, “A simple weight decay can improve generalization,” *Advances in neural information processing systems*, vol. 4, pp. 950–957, 1995.
- [23] N Srivastava, “Improving neural networks with dropout,” 2013. [Online]. Available: http://www.cs.toronto.edu/~nitish/msc_thesis.pdf.
- [24] I. Davidson and W. Fan, “When Efficient Model Averaging Out-Performs Boosting and Bagging,” no. 1,
- [25] G. Hinton and N Srivastava, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv preprint arXiv: ...*, pp. 1–18, 2012. arXiv:[arXiv:1207.0580v1](https://arxiv.org/abs/1207.0580v1). [Online]. Available: <http://arxiv.org/abs/1207.0580>.
- [26] P. Baldi and P. Sadowski, “Understanding dropout,” *NIPS 1*, 2013. [Online]. Available: http://www.editlib.org/INDEX.CFM?fuseaction=Reader.ViewAbstract&paper_id=4620.
- [27] R. M. Bell, Y. Koren, and C. Volinsky, “A perspective on the Netflix prize,” pp. 24–29,
- [28] A. Livnat, C. Papadimitriou, J. Dushoff, and M. W. Feldman, “A mixability theory for the role of sex in evolution,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 105, no. 50, pp. 19 803–8, Dec. 2008, ISSN: 1091-6490. DOI: [10.1073/pnas.0803596105](https://doi.org/10.1073/pnas.0803596105). [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2604923&tool=pmcentrez&rendertype=abstract>.
- [29] Hopfield networks. Accessed: 2013-12-27, [Online]. Available: http://www.scholarpedia.org/article/Hopfield_network.

- [30] Boltzmann machines. Accessed: 2013-12-19, [Online]. Available: <http://richardweiss.org/blog/?p=75>.
- [31] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009, ISSN: 1935-8237. DOI: [10.1561/22000000006](https://doi.org/10.1561/22000000006). [Online]. Available: <http://www.nowpublishers.com/product.aspx?product=MAL&doi=22000000006>.
- [32] G. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural computation*, 2002. [Online]. Available: <http://www.mitpressjournals.org/doi/abs/10.1162/089976602760128018>.
- [33] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 6, pp. 721–741, 1984.
- [34] T. Tieleman, "Training restricted Boltzmann machines using approximations to the likelihood gradient," *Proceedings of the 25th international conference on Machine learning - ICML '08*, pp. 1064–1071, 2008. DOI: [10.1145/1390156.1390290](https://doi.org/10.1145/1390156.1390290). [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1390156.1390290>.
- [35] G. Hinton, "A practical guide to training restricted Boltzmann machines," *Momentum*, 2010. [Online]. Available: <http://www.csri.utoronto.ca/~hinton/absps/guideTR.pdf>.
- [36] R. Neal, "Annealed importance sampling," *Statistics and Computing*, 2001. [Online]. Available: <http://link.springer.com/article/10.1023/A:1008923215028>.
- [37] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing.," *Science (New York, N.Y.)*, vol. 220, pp. 671–680, 1983, ISSN: 0036-8075. DOI: [10.1126/science.220.4598.671](https://doi.org/10.1126/science.220.4598.671).
- [38] I. Beichl and F. Sullivan, "The importance of importance sampling," *Computing in Science & Engineering*, vol. 1, 1999, ISSN: 1521-9615. DOI: [10.1109/5992.753049](https://doi.org/10.1109/5992.753049).
- [39] Mnistdigitsdataset. Accessed: 2014-03-5, [Online]. Available: <http://deeplearning.net/tutorial/DBN.html>.
- [40] I. Sutskever and T. Tieleman, "On the convergence properties of contrastive divergence," ... *on Artificial Intelligence and ...*, 2010. [Online]. Available: http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2010_SutskeverT10.pdf.
- [41] G. Hinton, "Products of experts," in *9th International Conference on Artificial Neural Networks: ICANN '99*, vol. 1999, 1999, pp. 1–6, ISBN: 0 85296 721 7. DOI: [10.1049/cp:19991075](https://doi.org/10.1049/cp:19991075). [Online]. Available: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=8195325> \backslash\$nhhttp://www.scholarpedia.org/article/Product_of_experts.
- [42] Y. Teh and G. Hinton, "Rate-coded restricted Boltzmann machines for face recognition," *Advances in neural information processing ...*, 2001. [Online]. Available: <http://www.csri.utoronto.ca/~hinton/absps/nips00-ywt.pdf>.
- [43] V. Nair and G. Hinton, "Rectified linear units improve restricted boltzmann machines," ... *on Machine Learning (ICML-10)*, no. 3, 2010. [Online]. Available: http://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_NairH10.pdf.

- [44] Theano tutorial on dbn. Accessed: 2014-01-5, [Online]. Available: <http://deeplearning.net/tutorial/DBN.html>.
- [45] G. Hinton, P. Dayan, B. Frey, and R. Neal, "The" wake-sleep" algorithm for unsupervised neural networks," *SCIENCE-NEW YORK THEN ...*, 1995. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.82.804&rep=rep1&type=pdf>.
- [46] H. Lee, C. Ekanadham, and A. Ng, "Sparse deep belief net model for visual area V2," *Advances in neural ...*, pp. 1–8, 2007. [Online]. Available: http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2007_934.pdf.
- [47] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, IEEE, vol. 1, 2001, pp. I–511.
- [48] Open cv official website. Accessed: 2014-04-12, [Online]. Available: <http://opencv.org/>.
- [49] Open cv equalization methods. Accessed: 2014-05-26, [Online]. Available: http://docs.opencv.org/trunk/doc/py_tutorials/py_imgproc/py_histograms/py_histogram_equalization/py_histogram_equalization.html.
- [50] Kanade database. Accessed: 2014-04-05, [Online]. Available: <http://www.pitt.edu/~emotion/ck-spread.htm>.
- [51] M. Lyons, S. Akamatsu, M. Kamachi, and J. Gyoba, "Coding facial expressions with gabor wavelets," in *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*, IEEE, 1998, pp. 200–205.
- [52] A. Georghiades, P. Belhumeur, and D. Kriegman, "From few to many: illumination cone models for face recognition under variable lighting and pose," *IEEE Trans. Pattern Anal. Mach. Intelligence*, vol. 23, no. 6, pp. 643–660, 2001.
- [53] Nottingham database. Accessed: 2014-04-12, [Online]. Available: <http://www.kasrl.org/jaffe.html>.
- [54] F. S. Samaria and A. C. Harter, "Parameterisation of a stochastic model for human face identification," in *Applications of Computer Vision, 1994., Proceedings of the Second IEEE Workshop on*, IEEE, 1994, pp. 138–142.
- [55] I. J. Goodfellow, D. Erhan, P. L. Carrier, A. Courville, M. Mirza, B. Hamner, W. Cukierski, Y. Tang, D. Thaler, D.-H. Lee, *et al.*, "Challenges in representation learning: a report on three machine learning contests," in *Neural Information Processing*, Springer, 2013, pp. 117–124.
- [56] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.
- [57] J. Bergstra and O. Breuleux, "Theano: A CPU and GPU math compiler in Python," ... of *9th Python in ...*, no. Scipy, pp. 1–7, 2010. [Online]. Available: http://www-etud.iro.umontreal.ca/~wardefar/publications/theano_scipy2010.pdf.
- [58] Openblas. Accessed: 2013-12-21, [Online]. Available: <http://www.openblas.net/>.

- [59] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [60] J. D. Hunter, "Matplotlib: a 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.
- [61] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "Scikit-image: image processing in python," *PeerJ PrePrints*, Tech. Rep., 2014.
- [62] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: the best of both worlds," *Computing in Science and Engineering*, vol. 13.2, pp. 31–39, 2011. [Online]. Available: <http://www.computer.org/portal/web/csdl/doi/10.1109/MCSE.2010.118>.
- [63] Atlas. Accessed: 2014-02-26, [Online]. Available: <http://math-atlas.sourceforge.net/>.
- [64] Comparison between matrix libraries. Accessed: 2014-02-26, [Online]. Available: <http://stackoverflow.com/questions/5260068/multithreaded-blas-in-python-numpy>.
- [65] Looping in theano. Accessed: 2014-04-03, [Online]. Available: <http://deeplearning.net/software/theano/library/scan.html>.
- [66] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, vol. 2011, 2011.
- [67] Y. Tang, "Deep learning using linear support vector machines," in *Workshop on Challenges in Representation Learning, ICML*, 2013.
- [68] M. Liu, S. Li, S. Shan, and X. Chen, "Au-aware deep networks for facial expression recognition," in *Automatic Face and Gesture Recognition (FG), 2013 10th IEEE International Conference and Workshops on*, IEEE, 2013, pp. 1–6.
- [69] O. Rudovic, M. Pantic, and I. Patras, "Coupled gaussian processes for pose-invariant facial expression recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 6, pp. 1357–1369, 2013.
- [70] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio, "Pylearn2: a machine learning research library," *arXiv preprint arXiv:1308.4214*, 2013.
- [71] Nolearn dbn library. Accessed: 2014-06-05, [Online]. Available: <https://pythonhosted.org/nolearn/>.
- [72] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pp. 1–8, 2009. DOI: [10.1145/1553374.1553453](https://doi.org/10.1145/1553374.1553453). [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1553374.1553453>.
- [73] R. Salakhutdinov and G. E. Hinton, "Deep boltzmann machines," in *International Conference on Artificial Intelligence and Statistics*, 2009, pp. 448–455.

-
- [74] Y. Kim, H. Lee, and E. M. Provost, “Deep learning for robust feature generation in audiovisual emotion recognition,” in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, IEEE, 2013, pp. 3687–3691.
 - [75] A. Fischer and C. Igel, “An introduction to restricted boltzmann machines,” in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, Springer, 2012, pp. 14–36.