University of West Bohemia

Faculty of Applied Sciences

Department of Computer Science and Engineering

Master's Thesis

Image similarity assessment

Pilsen 2014

Vojtěch Frič

Declaration of Authenticity

I hereby declare that this master thesis is completely my own work and that I have used only the cited sources.

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

Pilsen, May 12, 2014

Vojtěch Frič

Abstract

The main goal of this work is to explore the methods usable for image similarity assessment. First part of this work is dedicated to a feature based approach utilizing the wavelet transform. The second part is dedicated to methods for extracting low dimensional codes from natural images. Particular emphasis is given to the use of neural networks. Secondary goal of this work is to explore the options of using the deep learning methods in an unsupervised machine learning field, especially in tasks of computer vision and image processing.

Keywords

image similarity, neural networks, deep learning, wavelet transform, content based image retrieval, restricted Boltzman machines, deep belief networks, RBM, DBN

Abstrakt

Hlavním cílem této práce je prozkoumat postupy využitelné pro analýzu podobnosti obrázků. První část práce je věnována příznakové metodě založené na waveletové transformaci. Druhá část se věnuje metodám pro extrakci nízko dimenzionálních kódů z přirozených obrázků. Zvláštní důraz je kladen na využití neuronových sítí. Dalším cílem této práce je prozkoumání možností využití hlubokého učení v oboru strojového učení bez učitele, především v oblasti počítačového vidění a zpracování obrazu.

Klíčová slova

podobnost obrázků, neuronové sítě, hluboké učení, waveletová transformace, vyhledávání na základě obsahu, restricted Boltzman machines, deep belief networks, RBM, DBN

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Kamil Ekštein, PhD. for his guidance and inspiration provided the early stages of this work.

I would like to thank my friend D.H. for sharing his language knowledge and bearing with my demands, under tight deadlines, thanks.

Contents

1 Introduction **1**

2 Brief Review of the Existing Methods **3**

2.1 Euclidean Distance 3

2.2 Image Hashing 5

2.2.1 Average Hashing 5

2.2.2 Perceptual Hashing 6

2.2.3 Difference Hashing 6

2.3 Feature-based Methods 7

3 Multi-resolution Image Querying **9**

3.1 Discrete Wavelet Transform 9

3.1.1 Extension to 2D 10

3.2 Image-querying Metric 11

3.2.1 Preprocessing of DWT Coefficients 11

3.2.2 Metric 12

3.3 Tuning the Metric 14

3.3.1 Logistic regression 15

3.3.2 Gradient Descent 17

3.4 Implementation Details 20

3.4.1 Haar Wavelet Decomposition 20

4 Low Dimensional Codes **23**

4.1 Principal Component Analysis 23

4.1.1 “Toy Example” 23

4.1.2 Idea Behind PCA . . . 25
4.1.3 Solving PCA . . . 29
4.2 Artificial neural networks . . . 32
4.2.1 Biological neuron . . . 32
4.2.2 Neuron Models . . . 33
4.3 Deep Learning . . . 35
4.3.1 Autoencoders . . . 35
4.3.2 Denoising autoencoders . . . 37
4.3.3 Stacked autoencoders . . . 37
4.3.4 Energy-Based Models . . . 38
4.3.5 Sampling in an RBM . . . 41
4.3.6 Deep Belief Networks . . . 42
4.4 Training Neural Networks . . . 44
4.4.1 Backpropagation Algorithm . . . 45

5 Experitments and Results 48
5.1 Datasets . . . 48
5.1.1 Imgur Dataset . . . 48
5.1.2 NUS WIDE . . . 49
5.1.3 CIFAR 10 . . . 49
5.1.4 MIRFLICKR . . . 50
5.2 Tools . . . 51
5.2.1 Python . . . 51
5.2.2 Theano . . . 52
5.2.3 Pylearn . . . 52
5.2.4 Theanets . . . 53
5.2.5 RBM Implementation . . . 53
5.3 Experiments . . . 54
5.3.1 How to read results . . . 56
5.3.2 Euclidean distance . . . 56
5.3.3 PCA . . . 57
5.3.4 Autoencoder and Denoising Autoencoders . . . 57
5.3.5 Deep Belief Network . . . 59

5.3.6 Multi-resolution Image Querying 60

6 Conclusion **64**

A MIRFLICKR Top 20 Tags **71**

B User Guide **72**

B.1 Requirements 72

B.2 Installation 73

B.3 Running experiments 73

B.4 Results 74

1 Introduction

In the days of Internet boom, social networks and affordable smart phones capable of taking high quality photos and videos, users have instantaneous access to millions of images across the Web. Given these circumstances the need to search, filter and organize the images is more and more crucial. In the case of small collections (i.e. hundreds of items) it is possible to search for the desired images or duplicates manually. This becomes infeasible if the number of items increases. Online photo banks are good manifestation of the need for easily searchable collection of images. The server `gettyimages.com` was recently[1] made available to public[6]. It hosts over 30 million images and stock photographs. The photographs are tagged but the tags can be rather misleading.

The problem with tagging is, that the tag sometimes does not describe the object a user is interested in. Other typical problem is the ambiguity of the tags. When I ran a query *"black horse"* with the intent to retrieve images of dark coloured stallions, the first result was an athlete training on a pommel horse with a black background. To be fair, the search engine is smart enough to ask the user to resolve the ambiguities.

Tags bring another issue. Each image has to be manually tagged. Also the tags can be easily manipulated as shown by the *241543903*[2] tag phenomenon. Given the diversity of natural language it is possible to describe single image with multiple different tags with overlapping meaning.

A breakthrough in research in recent years allows us to classify images automatically. The problem is that systems capable of classi-

[1] March 5th, 2014

[2] http://241543903.com/about-241543903

fying natural images need to be trained on labelled data. That means the problem is not solved, it is just shifted to another domain.

For this paper I have decided to look into a different problem. To search and retrieve images using another images. This approach is probably the most helpful when locating duplicates in large collections of images or when searching for a picture with the same content but in different resolution. Those are the main selling points of this family of algorithms.

I see another potential use of the systems capable of searching using image query: augmented reality systems recently gained on popularity greatly. If the system will be backed by algorithms capable of interactive image search, it would be possible to provide helpful contextual information to the user. The possibilities are limitless. From simple tasks such as recognizing the logo of an unknown brand to confirming that the picked mushroom is actually edible.

2 Brief Review of the Existing Methods

In this chapter I will present a brief overview of the existing methods of solving the problem of retrieving similar images. Each section will cover the principles and the basic ideas of each method. Even though it is not explicitly stated for each method, all work under the assumption of normalized data. Each method uses different normalization but all the methods need image with identical size and colour space.

2.1 Euclidean Distance

Calculating Euclidean distance of two images to estimate their similarity is the simplest method in this review. That does not mean that it does not have its uses. The Euclidean distance of two points in N-dimensional space is defined as follows:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{N} (p_i - q_i)^2}, \tag{2.1}$$

where $\mathbf{p}$ and $\mathbf{q}$ are two points (vectors) and the indexed form denotes the respective elements of the vector.

This method brings several complications. At first, the metric is designed to work with points in Euclidean space, not with images. This can be overcome by treating each pixel of an image as an element of a vector and calculating the mutual distance element–wise. Other complication is the actual representation of the result. Given the nature of this metric it is not entirely clear what should be the threshold for discerning similar images from dissimilar. The initial di-

mensions of the image also play a crucial role. The bigger the image is, the bigger the resulting distance can be. Assume we are comparing two images of dogs. For one calculation the images will be 32 by 32 pixels large, for the other only 16 by 16 pixels big. The distance scales quadratically with respect to image dimensions. This effect can be partially eliminated by scaling the resulting distance by the factor proportional to the dimensions of the image

$$d(\mathbf{p}, \mathbf{q}) = \frac{\sqrt{\sum_{i=1}^{N}(p_i - q_i)^2}}{|p|}. \tag{2.2}$$

Euclidean distance possesses even more severe problem, it doesn't tell anything about the structure and content of the image. The loss of information due to the reduction to single a number is too massive to draw any conclusions from it. We can demonstrate this on the artificial example in fig. 2.1.

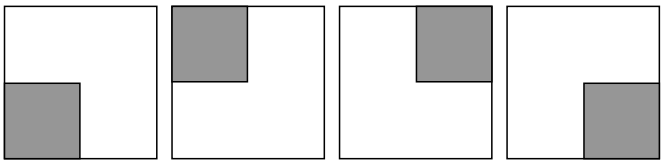


Figure 2.1: All images have the same Euclidean distance to each other

All shown images have the same Euclidean distance to each other but they are not similar. We will not consider the images being similar despite the fact that they all have one dark quarter.

As stated before, using Euclidean distance as an absolute metric is not practical. That does not mean it is an ineffective method though. Due to its simplicity it can be used to select potential matches quickly from a larger database. Albeit it is debatable what threshold to use to differentiate the matches.

2.2 Image Hashing

Perceptual hashing is, despite its name, quite different from its cryptographic counterpart. In cryptography the data works as a seed to a random function. In cryptographic hashing it is often also desirable that similar input data produce different hashes.

For the image comparison the need is exactly the opposite. Similar pictures should produce similar or same hash fingerprints. There is a number of hashing methods designed to work with pictures (or texts, audio, and other media) which fulfils the requirement.

All described hashing methods work on the same principle: At first, the image dimensions are reduced. The final size is arbitrary but the common choice is 8 by 8 pixels yielding 64 pixels in total. The main benefit of scaling is the removal of high frequencies from the image and improved performance of the algorithm. Scaling is followed by a conversion to greyscale. This step reduces the image to 64 values. The hash is then calculated from there.

2.2.1 Average Hashing

Average hashing is the simplest approach to generate the hash fingerprint. The resulting 8 by 8 matrix is converted to binary representation by thresholding. The assignment of 0 or 1 is based upon a comparison of the grey value g of the considered pixel with a predetermined threshold value t. The resulting image can thus be calculated with very little computational effort since for each pixel only a simple compare operation must be performed. The calculation (2.3) is

applied to each element of the matrix.

$$i_{bin} = \begin{cases} 0 & i_{grey} \leq t \\ 1 & \text{otherwise} \end{cases} \tag{2.3}$$

Value of parameter t is the average value of the input matrix.

2.2.2 Perceptual Hashing

The perceptual hashing[15, 22] uses a different approach for to obtain the desired 64 values. First step is again a size reduction and conversion to greyscale but the image is larger. The next step is to calculate Discrete Cosine Transformation (DCT) of the image. DCT expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. The common use of DCT is in JPEG compression algorithm and in various video and audio compression algorithms.

Only 64 values representing the lowest frequencies in the image are considered for further processing. These 64 values are then thresholded using the eq. (2.3) described in previous paragraph. There is one difference though: While calculating the average value for thresholding, the first coefficient should be omitted. This coefficient corresponds to 0th frequency and can significantly differ from the other values and thus skewing the average.

2.2.3 Difference Hashing

The difference hashing is another approach to image hashing based upon the same principle. Starting with scaled greyscale image, the binary values are based upon difference of neighbouring pixels.

$$i_{bin} = \begin{cases} 0 & i_{grey}[n] < i_{grey}[n+1] \\ 1 & \text{otherwise} \end{cases} \tag{2.4}$$

The resulting binary images are then arranged to form 64-bit number – the hash fingerprint. The arrangement method is of no consequence as long as it is consistent. The resulting fingerprints are then compared using Hamming distance where smaller distance means better match.

These hashing techniques are quite powerful with regard to finding matches in a large collection. The hashes are fairly robust to scaling and colour manipulations and can be used to narrow down the search space.

2.3 Feature-based Methods

Although this work is focused mostly on neural networks, I have to mention the feature based methods for the sake of completeness.

Many systems designed for the task of content-based image retrieval are based upon the colour and shape features. In the early years of the research there was a popular trend of using colour histogram only. The main flaw of this approach is that it leaves out the spatial information entirely. Techniques based upon the histogram only were later superseded by the colour correlograms, as seen in [11] and the edge histograms [19].

Advancements in the research later introduced new methods focused more on the overall scene representation[18, 5]. GIST and SIFT descriptors are able to extract interesting features from images utilising various signal processing techniques. These feature descriptors were designed with recognition in mind but can be exploited in simi-

larity assessment tasks as well.

I have dedicated one chapter to selected feature based algorithm which I have successfully used earlier.

3 Multi-resolution Image Querying

In [13] authors introduce an elegant method for fast retrieval of images based upon a low quality query image. The system performs well even when the query image is a rough hand-made sketch of the target image. The authors use features extracted from an output of a wavelet transform of the input image. The coefficients of the transform are processed and stored in a custom-designed data structure which allows a quick retrieval.

I have chosen this method as a feature-based counterpart to the neural network approach also presented in this work. From the many methods available I have chosen this one because I had previous experience with it.

3.1 Discrete Wavelet Transform

The wavelet transform is closely related to the Fourier transform. Its key advantage is that the wavelet transformation has a temporal resolution as well. The wavelet transform captures both frequency and location information. Mathematically, the wavelet transform is a representation of a function by orthonormal series generated by a wavelet.

There are many wavelets. The simplest one is the Haar wavelet designed by Alfréd Haar in 1910. For an input represented by a list of 2^n numbers, the Haar wavelet transform may be considered as pairing the input values, storing the difference and passing the sum. This process is repeated recursively pairing up the averages to provide

the next scale and finally resulting in $2^n - 1$ differences and one final sum.

The Haar discrete wavelet transform can be expressed in a matrix form. The transformation matrix (3.1) has $N \times N$ shape and is composed of basis vectors derived from the Haar mother wavelet. In the previous example the transform matrix (3.2) is only orthogonal. Equation (3.3) shows the smallest possible basis for the Haar transformation.

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0^T \\ \mathbf{h}_1^T \\ \vdots \\ \mathbf{h}_n^T \end{bmatrix} \tag{3.1}$$

$$\mathbf{H} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.2}$$

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.3}$$

The Haar wavelets were chosen because they are easiest to implement and fastest to calculate.

3.1.1 Extension to 2D

The Haar wavelet transform in its form presented in the previous section is defined for a 1D signal. The extension to 2D images is straightforward – first the rows are decomposed, then the columns. However, there are two options how to proceed.

Standard decomposition fully decomposes the rows, then the columns. This produces rectangular artefacts when used for image compression.

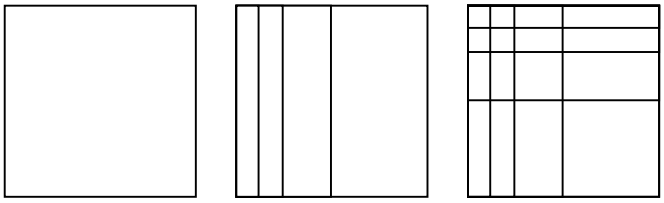


Figure 3.1: Process of standard Haar decomposition

Nonstandard decomposition alternates rows and columns, decomposing one level at a time. In this case, the compression artefacts are squares.

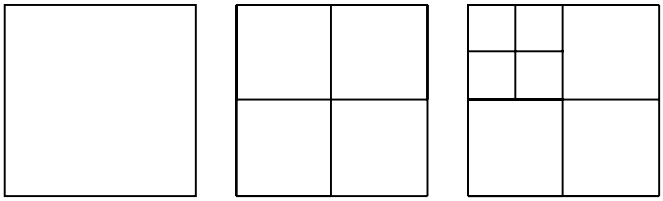


Figure 3.2: Process of nonstandard Haar decomposition

3.2 Image-querying Metric

The discrete wavelet transform (DWT) on its own does not help with searching for similar images. The resulting transformed image has the same size as the original data, thus there is no reduction of data to process. The image-querying metric described in this section demands preprocessed DWT coefficient to work properly.

3.2.1 Preprocessing of DWT Coefficients

The first step in preprocessing the transformed image is the truncation of the coefficients. An image of the size 32 by 32 pixels yields 1024 coefficients per colour channel. Certain number of the coefficients is

usually zero or very close to zero, in the case of processing natural images. This fact enables to employ the DWT in various data compression scenarios – probably the best known is the JPEG 2000 image compression algorithm. The image-querying metric takes advantage of this fact to greatly simplify the results of the transformation.

In the truncation step of the preprocessing, only m coefficients with the greatest magnitude are selected; rest of the coefficients is set to zero. This truncation both accelerates the search for the query and reduces storage for the database. Truncating the coefficients also greatly improves the discriminatory power of the metric, probably because it prioritizes the most important features of the image. The number of preserved coefficients is to be found experimentally.

The next preprocessing step is the quantization of the truncated coefficients. The quantization – alike the truncation – reduces the search time and the storage space for database and improves the discriminatory power of the metric. The quantized coefficients retain little or no information about the original magnitudes of the major features in an image. However, the presence or absence of these features is sufficient to decide about similarity of the measured samples. The coefficients are quantized using the *signum* function:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{3.4}$$

3.2.2 Metric

The image querying metric is defined as follows:

$$\|Q, T\| = w_{0,0} \left| Q[0,0] - T[0,0] \right| + \sum_{i,j} w_{i,j} \left| \tilde{Q}[i,j] - \tilde{T}[i,j] \right|. \tag{3.5}$$

The Q and T represent wavelet decompositions of the query and the target image. The $Q[0,0]$ and $T[0,0]$ are coefficients corresponding to the average intensity of the decomposed image. Further, let $\tilde{Q}[i,j]$ and $\tilde{T}[i,j]$ be the $[i,j]$-th truncated quantized coefficient of Q and T; its value is either -1, 0 or +1.

We can simplify the metric in several ways. The term $\left|\tilde{Q}[i,j] - \tilde{T}[i,j]\right|$ can be replaced by $\left(\tilde{Q}[i,j] \neq \tilde{T}[i,j]\right)$ where the expression $(a \neq b)$ is evaluated as shown in eq. (3.6).

$$(a \neq b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

Next, the terms are grouped together into *buckets*. The weights are grouped using the function $bin(i,j)$ which is defined as

$$bin(i,j) = \min\{\max\{i,j\}, B\}, \tag{3.7}$$

where B is a parameter chosen manually. The idea behind grouping is that the probability of coefficient being on the same scale level as its neighbour is increasing with the distance from $[0,0]$, and that information they carry has identical impact. As shown in fig. 3.2, three quarters of the coefficients correspond to the same scale level.

Finally, to speed up the search in database, only the coefficients of the query $\tilde{Q}[i,j]$ which are non-zero are considered. The final form of the metric is as follows:

$$\|Q,T\| = w_{0,0}\,|Q[0,0] - T[0,0]| + \sum_{i,j:\tilde{Q}[i,j]\neq 0} w_{i,j}\left(\tilde{Q}[i,j] \neq \tilde{T}[i,j]\right). \tag{3.8}$$

It is worth noting that the last step disqualifies this formula as a metric because it violates symmetry. In the rest of this work the term

metric will be used to avoid confusion.

Computational form of the metric

The *metric* (3.8) is further refined to allow for simple computations. It is assumed that the final database will contain much more mismatches than matches, therefore we can adjust the metric to reflect this. The summation in eq. (3.8) can be rewritten in terms of equality operator

$$(a = b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

Using this operator the term

$$\sum_{i,j:\tilde{Q}[i,j]\neq 0} w_{i,j} \left(\tilde{Q} \neq \tilde{T}\right)$$

in eq. (3.8) can be rewritten as

$$\sum_{i,j:\tilde{Q}[i,j]\neq 0} w_k - \sum_{i,j:\tilde{Q}[i,j]\neq 0} w_{i,j} \left(\tilde{Q} = \tilde{T}\right).$$

Since the first term $\sum w_k$ is independent of $\tilde{T}$, we can ignore it for the purpose of ranking the images in database with respect to the metric. To rank the images, it is sufficient to calculate the expression

$$w_0 \left|Q[0,0] - T[0,0]\right| - \sum_{i,j:\tilde{Q}[i,j]=0} w_{bin(i,j)} \left(\tilde{Q}[i,j] = \tilde{T}[i,j]\right). \tag{3.10}$$

3.3 Tuning the Metric

The final metric (3.10) involves a linear combination of terms. In this section, I will discuss how to find a good set weight to parametrize the metric. It is possible to use some multidimensional optimizations

but it would be difficult to find the appropriate cost function. The regression models based upon the least squares fit are not suitable for this task either. If the database contains 100 samples, then for each match there are 99 mismatches. This would cause the regression model to shift towards the mismatches. In contrast, using equal-sized sets of matches and mismatches means leaving useful data out.

Another way of how to think of the problem of finding proper weights for the metric is to consider it a classification problem. The matches form one class and the mismatches second one, while the parameters of the classification model will be used in the metric. The simplest classifier – despite its name – is logistic regression which is used to find the good set of weights.

3.3.1 Logistic regression

Logistic regression is a method capable of modeling a random variable with Bernoulli distribution ($B(p)$) which is a special case of binomial distribution $Bi(n, p)$ where n – number of trials – is equal to 1 and p is the probability of success.

In our case the dependent variable Y can only have two values 0 in the case of a mismatch and 1 in the case that the query and target image matches. The random variable $Y \sim B(p)$ can be described as follows

$$P(Y = y) = p^y(1 - p)^{1-y}. \tag{3.11}$$

It is not possible to model the random variable directly in the form

$$Y = \Theta_0 + \Theta_1 x_1 + \cdots + \Theta_n x_n$$

because the variable Y is discrete while the coefficients $\boldsymbol{\Theta}$ and values $\mathbf{X}$ are from $\mathbb{R}$. We can model the probability that Y will get a specific

value, though. Modelling the probability of Y getting value

$$P(Y = y) = \Theta_0 + \Theta_1 x_1 + \cdots + \Theta_n x_n, \quad y \in \langle 0; 1 \rangle$$

still does not guarantee that the data $\mathbf{X}$ will not generate a value outside the interval $\langle 0; 1 \rangle$. This can be solved by introducing the *odds* function:

$$odds(P(Y = 1)) = \frac{P(Y = 1)}{P(Y = 0)} = \frac{P(Y = 1)}{1 - P(Y = 1)} \tag{3.12}$$

The *odds* function represents how many times it is more likely that the value of Y will be 1 as opposed to the value of Y being 0. The range of the *odds* function is $(0, \infty)$.

The next step is to extend the range of the *odds* function so it covers the whole $(-\infty, \infty)$ interval. For this we define the *logit* function:

$$logit(P(Y = 1)) = ln(odds(P(Y = 1))) = ln\left(\frac{P(Y = 1)}{1 - P(Y = 1)}\right), \tag{3.13}$$

which can be modelled similarly to the linear regression:

$$logit(P(Y = 1)) = \Theta_0 + \Theta_1 x_1 + \cdots + \Theta_n x_n.$$

The probability is expressed as follows:

$$P(Y = 1) = \frac{1}{1 + e^{-\mathbf{X}^T \boldsymbol{\Theta}}} \tag{3.14}$$

which is a hypothesis for logistic regression model.

The next step is to find the appropriate cost function which can be used for calculating the optimal parameters of the model. Similarly

to linear regression we define the cost function as the least squares fit

$$J(\Theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\Theta(\mathbf{x}^{(i)}), y^{(i)})$$

$$Cost(h_\Theta(\mathbf{x}), y) = \begin{cases} -\log(h_\Theta(\mathbf{x})) & \text{if } y = 1 \\ -\log(1 - h_\Theta(\mathbf{x})) & \text{if } y = 0, \end{cases} \tag{3.15}$$

where $h_\Theta(\mathbf{x})$ is hypothesis (3.14) and m is a number of training samples. Because $y \in \{0; 1\}$ we can rewrite the *Cost* function as

$$Cost(h_\Theta(\mathbf{x}), y) = -y\log(h_\Theta(\mathbf{x})) - (1 - y)\log(1 - h_\Theta(\mathbf{x})),$$

then we can write the final cost function of the logistic regression model as

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} -y^{(i)}\log(h_\Theta(\mathbf{x}^{(i)})) - (1 - y^{(i)})\log(1 - h_\Theta(\mathbf{x}^{(i)}))\right], \tag{3.16}$$

which is a form suitable for minimization.

3.3.2 Gradient Descent

Gradient descent is a way to find a local minimum of function $f(\mathbf{x})$. We start with an initial guess and in each step we move in the direction of negative gradient $-\nabla f(\mathbf{x})$. We continue this process until we get to the point, where the function gradient is zero. That point is local minimum of the examined function. If our function is $f(\mathbf{x})$ and the initial guess is $\mathbf{x}_0$ then the gradient descent can be defined as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda\nabla f(\mathbf{x}), \tag{3.17}$$

where λ is a learning constant. The purpose of this parameter is to guarantee, that the algorithm will converge. The choice of λ is usually in a range 10^{-1}–10^{-6}. Properly chosen parameter λ ensures

that $f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k)$ hold for every k.

The main problem for gradient descent are function which has narrow valleys and plateaus. In case of narrow valleys, the gradient descent can start oscillating in zig-zag pattern. Cause of this is, that in each step the direction of gradient is almost perpendicular in each step. The gradient descents progress slows down significantly.

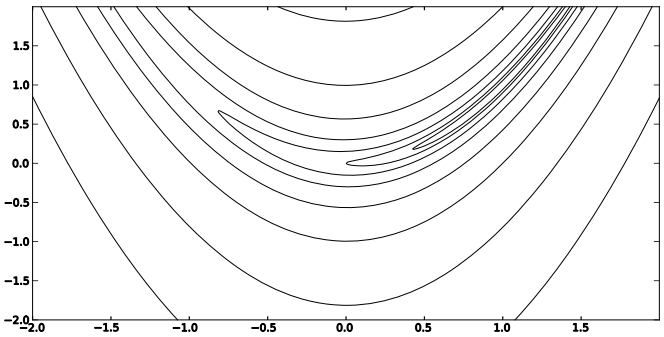


Figure 3.3: Rosenbrock function

Another problem are the plateaus. On the plateau the gradient of function is small. The progress is then slowed down significantly as well.

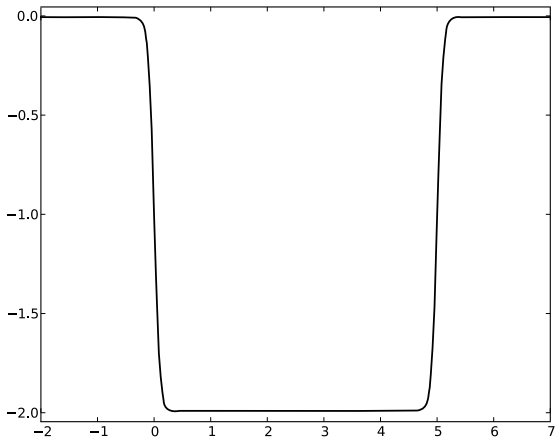


Figure 3.4: Function with distinct plateau

In the case of the logistic regression our goal is to minimize the cost function (3.16). The update has then the following form:

$$\Theta_j := \Theta_j - \lambda \frac{\partial}{\partial \Theta_j} J(\Theta) \tag{3.18}$$

this update is applied simultaneously to all Θ.

There exists two flavours of the gradient descent algorithm. The *batch gradient descent* in listing 3.1 where before each step, every training sample is examined.

Listing 3.1 Batch gradient descent

```
1  while not converged() {
2    for j := 0 to n {
3      Θ_j := Θ_j − α Σ_{k=1}^{m} (h_Θ(x^(i)) − y^(i)) x_j^(i)
4    }
5  }
```

The other is the *stochastic gradient descent* in listing 3.2. Which makes step after every sample is examined.

Listing 3.2 Stochastic gradient descent

```
1  while not converged() {
2    for i := 0 to m {
3      for j := 0 to n {
4        Θ_j := Θ_j − α (h_Θ(x^(i)) − y^(i)) x_j^(i)
5      }
6    }
7  }
```

3.4 Implementation Details

In this section I will cover few implementation details of this method. This section is meant only as an implementation guideline, it is not meant as a program documentation.

3.4.1 Haar Wavelet Decomposition

As mentioned in section 3.1.1 there are two versions of the Haar wavelet decomposition, I will cover only the standard decomposition type.

Listing 3.3 Standard Haar decomposition

```
1  haar1D(A: array[0...h-1]) {
2    while h > 1 {
3      h := h/2;
4      for i := 0 to h-1 {
5        A'[i] := (A[2i] + A[2i + 1] )/√2;
6        A'[h+i] := (A[2i] - A[2i + 1] )/√2;
7      }
8      A := A';
9    }
10 }
```

The code above performs decomposition on array A of h elements, with h being a power of two. When working with images, this code would decompose a single row or a column of a single colour channel of the image. An entire $n \times n$ image T can be thus decomposed as follows:

After the decomposition, the element $T[0, 0]$ contains a value proportional to the average intensity of the channel while the rest of the array are the wavelet coefficients. It is important to emphasize that

Listing 3.4 Image Haar decomposition

```
1 haar2D(T: array[0...n-1, 0...n-1]){
2   for row := 0 to n-1 {
3     haar1D(T[row, 0...n-1]);
4   }
5   for col := 0 to n-1 {
6     haar1D(T[0...n-1, col]);
7   }
8 }
```

the algorithm processes the image in place, if the original image is needed, it is necessary to create a copy first.

In the final database only value $T[0,0]$ and m coefficients with the highest magnitude are stored (see section 3.2.1 on page 11). The m coefficients are stored in six *search arrays* with one array for each combination of sign (positive or negative) and colour channel.

For example, let $\mathcal{D}_c^+$ denote the search array for positive elements of the channel c. Each element of $\mathcal{D}_c^+[i,j]$ then contains a list of images that have a positive coefficient in the channel c at position $[i,j]$. The search arrays are computed for each target image. Because of the structure of the database it is simple to add new samples to the collection.

The querying process is best described by the algorithm presented in the original paper [13], shown in listing 3.5.

The resulting list of scores is sorted and the value with the lowest score is the best match. The *bin* function is defined in eq. (3.7).

Listing 3.5 Ranking query in database

```
1  rankQuery(Q: array[0...n-1, 0...n-1], m){
2    list := ∅
3    haar2D(Q);
4    scores[i] := 0, ∀i;
5    foreach T in Database {
6      scores[T.index] += w[0] * |Q[0,0] - T[0,0]|
7    }
8    Q̃ := truncate(Q, m)
9    for Q̃[i,j], ∀i,j where Q̃[i,j] ≠ 0 {
10     if Q̃[i,j] > 0
11       list := list ∪ D⁺[i,j]
12     else
13       list := list ∪ D⁻[i,j]
14
15     for ℓ in list
16       scores[ℓ] -= w[bin(i,j)]
17   }
18   return scores
19 }
```

4 Low Dimensional Codes

In this chapter I will focus on the low dimensional codes approach. There are few techniques how to extract the low dimensional codes from images but I will mostly focus on neural networks. At first, I will introduce principal component analysis, then the neural networks in general, then I will cover few architectures of interest, and at last I will discuss the tested methods in details.

4.1 Principal Component Analysis

The central idea of principal component analysis (PCA) is to reduce the dimensionality of a data set consisting of numerous interrelated variables while retaining as much of the variance present in the data as possible. This reduction of dimensionality allows faster execution of the algorithms and data compression. The compression is possible when there is a redundancy present in the data. Correlation is a redundancy as well; if the data is correlated it is not necessary to store the whole information but only the main components and calculate the rest of the original data. The image data are in general highly locally correlated. The intensity of the picture element is strongly correlated to the intensities of its neighbouring elements.

4.1.1 "Toy Example"

For the sake of this explanation, let's pretend we are physicists studying motion of ideal spring. The system in fig. 4.1 consists of a ball of mass m attached to *massless*, *frictionless* spring. During the experiment, the spring is stretched and because we are dealing with an ideal system, the ball will oscillate along z-axis about its equilibrium.

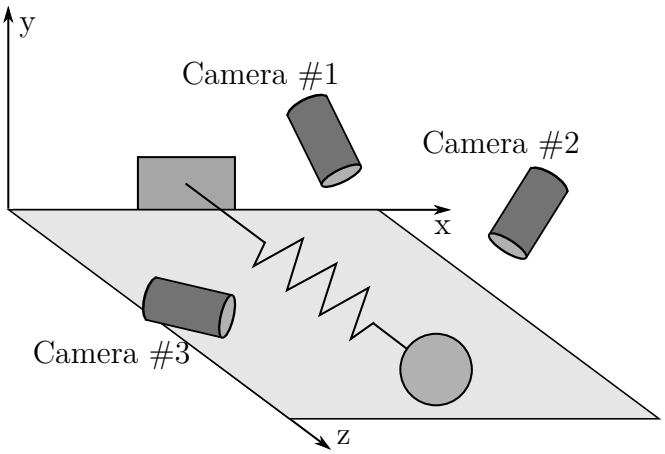


Figure 4.1: Spring experiment

The motion along the z-axis is solved by an explicit function of time.

In our experiment we set up three cameras at arbitrary positions and angles recording the ball. Our goal then is to extract the equation describing the movement of the ball. The task is also burdened by a noise – the cameras have discrete resolution, our measurements are not precise, or even the spring is *less-than-ideal*.

Dataset

Assume that the cameras are taking 30 frames per second, the experiment run for 1 minute. This dataset has 1,800 6-dimensional vectors where each camera provides 2-dimensional projection of the scene. In general, each sample is m-dimensional vector where m is the dimension of the measurement. Those samples lay in m-dimensional vector space defined by orthonormal basis. A simple choice of basis $\mathbf{B}$ is

identity matrix **I**.

$$\mathbf{B} = \begin{bmatrix} \mathbf{b_1} \\ \mathbf{b_2} \\ \vdots \\ \mathbf{b_n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \mathbf{I}$$

If we mark the three cameras A, B and C, then each sample can be expressed as 6-dimensional vector:

$$\mathbf{x} = \begin{bmatrix} x_A & y_A & x_B & y_B & x_C & y_C \end{bmatrix}^T$$

4.1.2 Idea Behind PCA

The goal of PCA is to find another basis, which is a linear combination of the original basis that re-expresses the data best. Under the assumption that PCA is linear we can define $m \times n$ matrices **X** and **Y** related by linear transformation **P**:

$$\mathbf{PX} = \mathbf{Y}, \tag{4.1}$$

where **X** is matrix of samples, **Y** is matrix of samples projected into a new basis, and **P** is the transformation matrix. To find the best basis we have to find the correct transformation **P**.

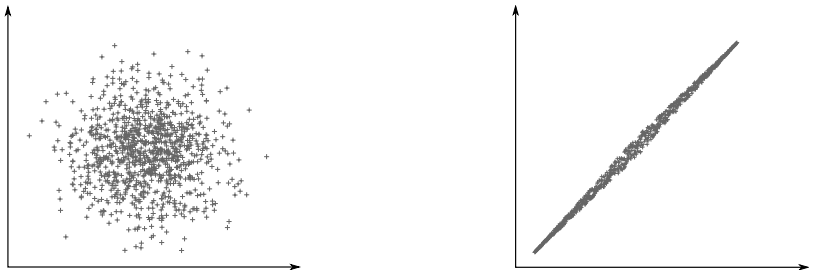
The best basis should assure the following properties:

Low noise – The noise in data must be low so it does not obscure the interesting features. There is no absolute scale for the noise but rather a relation of the noise to the signal is used. Signal to noise ratio (SNR) is defined as a ratio of variances:

$$SNR = \frac{\sigma^2_{signal}}{\sigma^2_{noise}} \tag{4.2}$$

A high SNR ($\gg 1$) indicates a precise data while a low SNR indicates a noisy data.

Low redundancy – In our example with the spring, the redundancy is induced to the experiment by using 3 cameras. In fig. 4.2a



(a) Data with low correlation (b) Data with high correlation

Figure 4.2: Data correlation example

the data has low correlation and redundancy. The 2 variables plotted has little to no mutual information. In fig. 4.2b the data form a distinct line. The two variables are clearly related i.e. one could be a measurement in inches and the second a measurement in centimetres. In the second example it would be enough to record the linear combination of those variables which would reduce the dimensionality of the data by one.

To find the redundancy between individual samples we have to calculate *covariance*. Consider two sets of measurements with zero mean.

$$A = a_1, a_2, \ldots, a_n, \;\; B = b_1, b_2, \ldots, b_n$$

The variance of set S is defined as follows.

$$\sigma_S^2 = \langle a_i a_i \rangle_i$$

where $\langle \cdot \rangle_i$ denotes average over values indexed by i. The covariance between A and B is a straightforward generalization

$$\text{covariance of } A \text{ and } B \equiv \sigma_{AB}^2 = \langle a_i b_i \rangle_i$$

Two important facts about covariance:

- $\sigma^2_{AB} = \langle a_i b_i \rangle_i = 0$ if and only if A and B are entirely uncorrelated,
- $\sigma^2_{AB} = \sigma^2_A$ if $A = B$

The sets can be converted to vectors without loss of generality.

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}$$

Then the covariance can be expressed as a dot product of two vectors

$$\sigma^2_{\mathbf{ab}} = \frac{1}{n-1}\mathbf{a}\mathbf{b}^T, \tag{4.3}$$

where the first term is a normalization factor. Finally from vectors we can make one step further to matrix composed from vectors.

$$\mathbf{X} = \begin{bmatrix} \mathbf{x_1}(=\mathbf{a}) \\ \mathbf{x_2}(=\mathbf{b}) \\ \vdots \\ \mathbf{x_m} \end{bmatrix}.$$

The covariance matrix is then defined as

$$\mathbf{S_X} \equiv \frac{1}{n-1}\mathbf{X}\mathbf{X}^T. \tag{4.4}$$

- The ij^{th} element of matrix $\mathbf{S_X}$ is equivalent to substituting $\mathbf{x_i}$ and $\mathbf{x_j}$ into eq. (4.3).
- $\mathbf{S_X}$ is a square symmetric $m \times m$ matrix.
- The diagonal terms of $\mathbf{S_X}$ are the *variance* of the corresponding vectors.

- The off-diagonal terms of $\mathbf{S_X}$ are the *covariance* of the corresponding vectors.

$\mathbf{S_X}$ quantifies the correlations among all possible pairs of vectors in the dataset.

As our goal is to remove redundancy from the dataset and as the redundancy is expressed in terms of covariance, we want the covariances between separate measurements to be zero. The covariance matrix $\mathbf{S_X}$ contains the covariances of the measurements in off-diagonal elements. We want these elements to be zero. Therefore, removing redundancy diagonalizes the matrix $\mathbf{S_X}$.

PCA assumes that all basis vectors $\mathbf{p_1}, \ldots, \mathbf{p_m}$ are orthonormal and the matrix $\mathbf{P}$ is also orthonormal. Secondly, PCA assumes that directions with the largest variance are the most important, i.e. the most *principal*.

There are few other assumptions made about PCA:

I *Linearity*
Linearity assures that PCA can be reduced to a change of the basis.

II *Mean and variance are sufficient statistics*
The only zero-mean probability distribution that is fully described by the variance is the Gaussian distribution. If the examined distribution is not Gaussian it could skew the results of PCA. In practice, a lot of real-world data has Gaussian distribution due to Central Limit Theorem.

III *Large variances hold important information*
This assumption is based upon the idea that a dataset has a high SNR. The principal components with higher variances are

associated with the useful data and components with low variances with the noise.

IV *The principal components are orthogonal*
This assumption simplifies the problem so that it is solvable with linear algebra decomposition techniques.

4.1.3 Solving PCA

One way to solve PCA is the eigenvector decomposition bu PCA is more commonly associated with Singular Value Decomposition (SVD).

Singular Value Decomposition

Let $\mathbf{X}$ be an arbitrary $m \times n$ matrix and $\mathbf{X}^T\mathbf{X}$ be a rank r, square symmetric n by n matrix.

Other definitions of interest are:

- $\{\hat{\mathbf{v}}_\mathbf{1}, \hat{\mathbf{v}}_\mathbf{2}, \ldots, \hat{\mathbf{v}}_\mathbf{r}\}$ is the set of *orthonormal* $m \times 1$ eigenvectors with associated eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ for the symmetric matrix $\mathbf{X}^T\mathbf{X}$
$$\left(\mathbf{X}^T\mathbf{X}\right)\hat{\mathbf{v}}_i = \lambda_i\hat{\mathbf{v}}_i,$$
- $\sigma_i \equiv \sqrt{\lambda_i}$ are *singular values*,
- $\{\hat{\mathbf{u}}_\mathbf{1}, \hat{\mathbf{u}}_\mathbf{2}, \ldots, \hat{\mathbf{u}}_\mathbf{r}\}$ is a set of *orthonormal* $n \times 1$ vectors defined by
$$\hat{\mathbf{u}}_\mathbf{i} \equiv \frac{1}{\sigma_i}\mathbf{X}\hat{\mathbf{v}}_\mathbf{i},$$
- $\{\hat{\mathbf{u}}_\mathbf{i}, \hat{\mathbf{u}}_\mathbf{j}\} = \delta_{ij}$ where δ_{ij} is **Kronecker delta**, function of two

variables defined as follows

$$\delta_{ij} = \begin{cases} 0 \text{ if } i \neq j, \\ 1 \text{ if } i = j, \end{cases}$$

- $\|\mathbf{X}\hat{\mathbf{v}}_i\| = \sigma_i$.

The third assumption can be rewritten as

$$\mathbf{X}\hat{\mathbf{v}}_\mathbf{i} = \sigma_i \hat{\mathbf{u}}_\mathbf{i}. \tag{4.5}$$

This can be extended to cover all vectors in the set and ultimately deriving the SVD formula.

The first step is constructing a diagonal matrix $\mathbf{\Sigma}$:

$$\Sigma \equiv \begin{bmatrix} \sigma_1 & & & & \\ & \ddots & & 0 & \\ & & \sigma_r & & \\ & & & 0 & \\ & 0 & & & \ddots \\ & & & & 0 \end{bmatrix}, \tag{4.6}$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_r$ are singular values. Next, we construct two additional orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$.

$$\mathbf{V} = \left[\hat{\mathbf{v}}_\mathbf{1}, \hat{\mathbf{v}}_\mathbf{2}, \ldots, \hat{\mathbf{v}}_\mathbf{r}\right],$$
$$\mathbf{U} = \left[\hat{\mathbf{u}}_\mathbf{1}, \hat{\mathbf{u}}_\mathbf{2}, \ldots, \hat{\mathbf{u}}_\mathbf{r}\right].$$

The missing vectors are added so that the matrix is square and orthogonal. By combinig the previously defined matrices we get the matrix form of SVD:

$$\mathbf{XV} = \mathbf{U\Sigma}. \tag{4.7}$$

As $\mathbf{V}$ is orthogonal, we can multiply both sides of equation by $\mathbf{V}^{-1} = \mathbf{V}^T$,

which will give the final form of the decomposition:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T. \tag{4.8}$$

Now the first k columns of $\mathbf{U}$ define a new basis for the reduced space. To get the projections of the input dataset we have to project the input samples into the reduced space:

$$\mathbf{z}_i = {\mathbf{U}_\mathbf{r}}^T \mathbf{x}_i,$$

where the $\mathbf{U_r}$ is the reduced matrix $\mathbf{U}$ containing the first k columns and $\mathbf{z}$ is projection of the sample.

The selection of k designates the dimensionality of the target space. The following formula is used to select a suitable k

$$\frac{\sum_{i=1}^{k} \sigma_{ii}}{\sum_{i=1}^{n} \sigma_{ii}} \geq p$$

where p is the parameter which controls the percentage of the retained variance in the output space. If we need to retain 99% of the input variance, we set $p = 0.99$ and choose the smallest k for which the inequality holds true.

4.2 Artificial neural networks

Artificial neural networks (ANN) are computational models inspired by biological structures. The neural network is composed of artificial neurons modeled after the biological neurons. They are usually presented as systems of interconnected neurons. The neurons are organized into layers and significance of connections are determined by associated synaptic weights.

4.2.1 Biological neuron

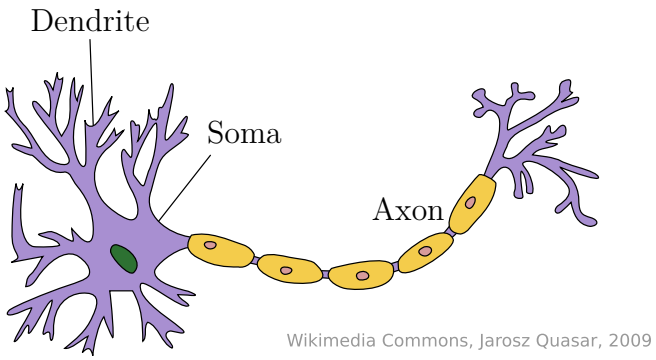


Figure 4.3: Diagram of *typical neuron* by Jarosz Quasar[1]

Biological neuron is a basic unit of neural system. The neurons are highly specialized cells capable of transmitting and processing signals. The neurons exist in many forms but for simplicity I will show only the basic *typical neuron*. A typical neuron is divided into three main parts: the soma or cell body, dendrites, and axon. The soma is a compact core from which the axon and dendrites extend. The dendrites usually span profusely extending their farthest branches only few microns far from the soma; axons, on the other hand, span great lengths. Synaptic signals from other neurons are received by the

[1] available under CC BY-SA 3.0 at `http://en.wikipedia.org/wiki/File:Neuron_Hand-tuned.svg`

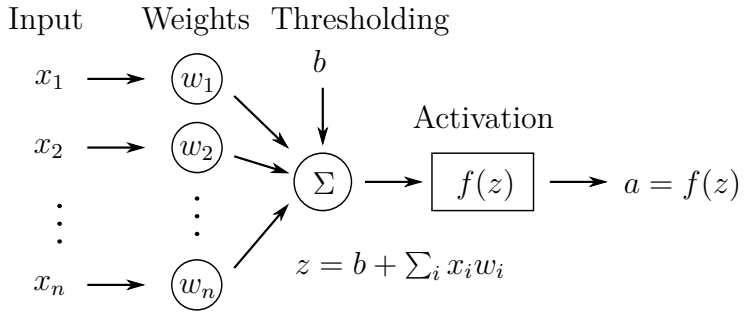


Figure 4.4: Diagram of artificial neuron

soma and dendrites while the signal to other neurons is transmitted by the axon.

The synaptic signals are generated in the axon hillock which is a point where the axon leaves its soma. When a signal travels along an axon and arrives at a synapse of a receiving neuron it causes a release of a transmitter chemical. The transmitter molecules diffuse across the synaptic cleft and bind to receptor channels in the receiving neuron. The effectiveness of transmission can be modified by changing the number of the released transmitter molecules and the number of the receiving channels. The synaptic weights adapt over the time so that the whole network learns to perform useful computations.

4.2.2 Neuron Models

There are few types of artificial neurons. Artificial neurons are modeled after the biological ones but there are some differences. The most notable difference is that artificial neurons communicate using real values rather than discrete peaks of activity. A scheme of the artificial neuron is shown in fig. 4.4.

Binary threshold neuron – is the first model of the artificial neuron authored in 1943 by McCulloch and Pitts. The neuron first com-

putes the weighted sum of the inputs, then sends a fixed peak of activity if the weighted sum exceeds a threshold. McCulloch and Pitts thought that each spike is like a truth value of a proposition. The input to the neuron is conceived as the logical formula and the activation is the result of this formula.

There are two equivalent ways to write the equations for a binary threshold neuron:

$$\begin{aligned} z &= \sum_i x_i w_i, & z &= b + \sum_i x_i w_i, \\ & & b &= -\Theta \\ y &= \begin{cases} 1 & \text{if } z \geq \Theta, \\ 0 & \text{otherwise.} \end{cases} & y &= \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{4.9}$$

In eq. (4.9) z is the total synaptic input of the neuron, y is the total activation of the neuron and b or Θ is the activation threshold.

Linear neurons are defined by the following equation:

$$y = \sum_i x_i w_i. \tag{4.10}$$

The linear neuron separates the input space into two half-spaces.

Rectified linear neuron also known as linear threshold neurons compute weighted sum of their inputs. The output is a non-linear function of total input.

$$\begin{aligned} z &= b + \sum_i x_i w_i, \\ y &= \begin{cases} z & \text{if } z > 0, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{4.11}$$

Sigmoid neurons give real-valued output that is a smooth function of their total input. They use logistic function which has reasonable derivative. The derivative is important for efficient learning algorithm.

$$\begin{aligned} z &= b + \sum_i x_i w_i \\ y &= \frac{1}{1 + e^{-z}} \end{aligned} \tag{4.12}$$

Stochastic binary neurons are practically the same as the sigmoid neurons (4.12). The only difference is that they treat the output as probability of activation.

4.3 Deep Learning

Disclaimer

The text in sections 4.3.1–4.3.4 is transcribed and/or adapted from the copyrighted material publicly available at `http://deeplearning.net/tutorial`. The intellectual property in those sections belongs to their respective owners.

4.3.1 Autoencoders

Autoencoder is one of basic building blocks used in early stages of deep architectures research. An autoencoder is 3-layer architecture with one hidden layer. An autoencoder transforms input $\mathbf{x} \in [0,1]^d$ i.e. binary vector, to hidden representation $\mathbf{y} \in [0,1]^d$ through deterministic mapping:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Where $f(\cdot)$ is a non-linearity, common choice for the non-linearity is *sigmoid* (3.14) or *tanh*. $\mathbf{W}$ and $\mathbf{b}$ are weight matrix and bias vector respectively. The hidden representation $\mathbf{y}$ is then mapped back into reconstruction $\mathbf{z}$ through similar transformation:

$$\mathbf{z} = f(\mathbf{W}'\mathbf{y} + \mathbf{b}')$$

In this case the $'$ does not indicate transpose operation. $\mathbf{z}$ represents reconstruction of $\mathbf{x}$ from code $\mathbf{y}$. The weight matrix $\mathbf{W}'$ of reverse mapping can be constrained by $\mathbf{W}' = \mathbf{W}^T$.

The parameters of this model are optimized in such way that the average squared error

$$L(\mathbf{x}, \mathbf{z}) = |\mathbf{x} - \mathbf{z}|^2$$

is minimized. In case the input is either bit vector or vector of bit probabilities the loss function is defined as *cross-entropy*:

$$L_H(\mathbf{x}, \mathbf{z}) = -\sum_{k=1}^{d} [\mathbf{x}_k \log \mathbf{z}_k + (1 - \mathbf{x}_k) \log(1 - \mathbf{z}_k)]$$

The idea is that the $\mathbf{y}$ captures the most interesting aspects of input variation similar to principal component analysis (see section 4.1 on page 23). $\mathbf{y}$ is viewed as lossy compression of $\mathbf{x}$, but it cannot be a good compression of all possible inputs. Learning drives the autoencoder to provide good compression for training samples and hopefully for samples with same distribution as the training samples as well.

If the network has only one linear hidden layer and the mean squared error is used to train the network, then the k hidden units learn to project the input in the first k principal components od the data. If the hidden layer is non-linear then, the auto-encoder behaves differently from *PCA*, with the ability to capture multi-

modal aspects of the input distribution.

4.3.2 Denoising autoencoders

One potential issue with autoencoders is that it there is no other constraint than square error, then it could learn identity function only. If the autoencoder learns identity, then it is only copying input to output and no compression is happening. This problem can be overcome by training the autoencoder to reconstruct from corrupted version of input.

The denoising auto-encoder is a stochastic version of the autoencoder where the input is stochastically corrupted, but the uncorrupted input is still used as target for the reconstruction. Intuitively, a denoising auto-encoder does two things: try to encode the input (preserve the information about the input), and try to undo the effect of a corruption process stochastically applied to the input of the autoencoder. The stochastic corruption consists in randomly setting some inputs to zero.

4.3.3 Stacked autoencoders

The denoising autoencoders can be stacked to form a deep network by feeding the latent representation (output code) of the denoising autoencoder found on the layer below as input to the current layer. The *unsupervised pre-training* of such an architecture is done one layer at a time. Each layer is trained as a denoising auto-encoder by minimizing the reconstruction of its input (which is the output code of the previous layer). Once the first k layers are trained, we can train the $k+1$-th layer because we can now compute the code or latent representation from the layer below. Once all layers are pre-trained, the network goes

through a second stage of training called fine-tuning. Here we consider *supervised fine-tuning* where we want to minimize prediction error on a supervised task. For this we first add a logistic regression layer on top of the network (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training.

4.3.4 Energy-Based Models

Energy-based models (EBM) associate a scalar energy to each configuration of the variables of interest. Learning corresponds to modifying that energy function so that its shape has desirable properties. For example, we would like plausible or desirable configurations to have low energy. Energy-based probabilistic models define a probability distribution through an energy function, as follows:

$$p(x) = \frac{e^{-E(x)}}{Z}. \tag{4.13}$$

The normalizing factor Z is called the partition function by analogy with physical systems.

$$Z = \sum_x e^{-E(x)}$$

An energy-based model can be learnt by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data. As for the logistic regression we will first define the log-likelihood and then the loss function as being the negative log-likelihood.

$$\mathcal{L}(\theta, \mathcal{D}) = \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log\ p(x^{(i)}) \ell(\theta, \mathcal{D}) = -\mathcal{L}(\theta, \mathcal{D})$$

using the stochastic gradient $-\frac{\partial \log p(x^{(i)})}{\partial \theta}$, where θ are the parameters of the model.

EBMs with Hidden Units

In many cases of interest, we do not observe the example x fully, or we want to introduce some non-observed variables to increase the expressive power of the model. So we consider an observed part x and a hidden part h. We can then write:

$$P(x) = \sum_h P(x,h) = \sum_h \frac{e^{-E(x,h)}}{Z}. \tag{4.14}$$

In such cases, to map this formulation to one similar to (4.13), we introduce the notation of free energy, defined as follows:

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)} \tag{4.15}$$

which allows us to write

$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ with } Z = \sum_x e^{-\mathcal{F}(x)}.$$

The data negative log-likelihood gradient then has a particularly interesting form.

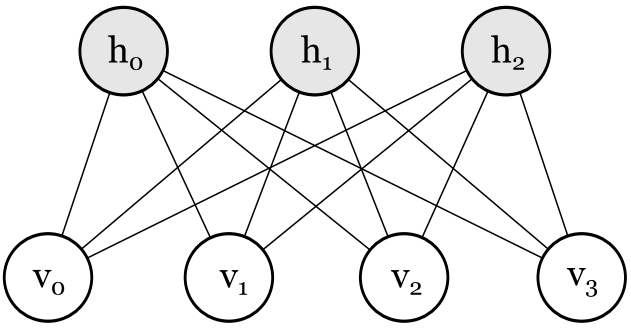
$$-\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \tag{4.16}$$

Notice that the above gradient contains two terms, which are referred to as the positive and negative phase. The terms positive and negative do not refer to the sign of each term in the equation, but rather reflect their effect on the probability density defined by the model. The first term increases the probability of training

data (by reducing the corresponding free energy), while the second term decreases the probability of samples generated by the model.

Restricted Boltzmann Machines (RBM)

Boltzmann Machines (BMs) are a particular form of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. A graphical depiction of an RBM is shown below.



The energy function $E(v, h)$ of an RBM is defined as:

$$E(v, h) = -b'v - c'h - h'Wv \tag{4.17}$$

where W represents the weights connecting hidden and visible units and b, c are the offsets of the visible and hidden layers respectively.

This translates directly to the following free energy formula:

$$\mathcal{F}(v) = -b'v - \sum_i \log \sum_{h_i} e^{h_i(c_i + W_i v)}.$$

Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another. Using this property, we can write:

$$p(h|v) = \prod_i p(h_i|v)$$
$$p(v|h) = \prod_j p(v_j|h).$$

RBMs with binary units

In the commonly studied case of using binary units (where v_j and $h_i \in \{0, 1\}$), we obtain from eq. (4.14) and eq. (4.17), a probabilistic version of the usual neuron activation function:

$$P(h_i = 1|v) = sigm(c_i + W_i v) \tag{4.18}$$

$$P(v_j = 1|h) = sigm(b_j + W'_j h) \tag{4.19}$$

The free energy of an RBM with binary units further simplifies to:

$$\mathcal{F}(v) = -b'v - \sum_i \log(1 + e^{(c_i + W_i v)}). \tag{4.20}$$

4.3.5 Sampling in an RBM

Samples of $p(x)$ can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator.

Gibbs sampling of the N random variables $S = (S_1, ..., S_N)$ is

done through a sequence of N sampling sub-steps of the form $S_i \sim p(S_i|S_{-i})$ where S_{-i} contains the $N-1$ other random variables in S excluding S_i.

For RBMs, S consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visibles. A step in the Markov chain is thus taken as follows:

$$h^{(n+1)} \sim sigm(W'v^{(n)} + c) \tag{4.21}$$

$$v^{(n+1)} \sim sigm(Wh^{(n+1)} + b), \tag{4.22}$$

where $h^{(n)}$ refers to the set of all hidden units at the n-th step of the Markov chain. What it means is that, for example, $h_i^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm(W_i'v^{(n)} + c_i)$, and similarly, $v_j^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $sigm(W_{.j}h^{(n+1)} + b_j)$. As $t \to \infty$, samples $(v^{(t)}, h^{(t)})$ are guaranteed to be accurate samples of $p(v, h)$.

In theory, each parameter update in the learning process would require running one such chain to convergence. It is needless to say that doing so would be prohibitively expensive. As such, several algorithms have been devised for RBMs, in order to efficiently sample from $p(v, h)$ during the learning process.

4.3.6 Deep Belief Networks

Hinton showed [10] that RBMs can be stacked and trained in a greedy manner to form so-called Deep Belief Networks (DBN). DBNs are graphical models which learn to extract a deep hierarchical representation of the training data. They model the joint

distribution between observed vector x and the ℓ hidden layers h^k as follows:

$$P(x, h^1, \ldots, h^\ell) = \left(\prod_{k=0}^{\ell-2} P(h^k | h^{k+1}) \right) P(h^{\ell-1}, h^\ell) \tag{4.23}$$

where $x = h^0, P(h^{k-1}|h^k)$ is a conditional distribution for the visible units conditioned on the hidden units of the RBM at level k, and $P(h^{\ell-1}, h^\ell)$ is the visible-hidden joint distribution in the top-level RBM.

The principle of greedy layer-wise unsupervised training can be applied to DBNs with RBMs as the building blocks for each layer [10, 2]. The process is as follows:

1. Train the first layer as an RBM that models the raw input $x = h^{(0)}$ as its visible layer.

2. Use that first layer to obtain a representation of the input that will be used as data for the second layer. Two common solutions exist. This representation can be chosen as being the mean activations $p(h^{(1)} = 1|h^{(0)})$ or samples of $p(h^{(1)}|h^{(0)})$.

3. Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of that RBM).

4. Iterate (2 and 3) for the desired number of layers, each time propagating upward either samples or mean values.

5. Fine-tune all the parameters of this deep architecture with respect to a proxy for the DBN log-likelihood, or with respect to a supervised training criterion

4.4 Training Neural Networks

One way to look at the feed-forward neural network is a directed graph. Each neuron of the network is one node of the graph. Every node is a computational unit whose edges transmit numerical information. Each unit is capable of evaluating some function of its input: that is the activation function of neuron. The network represents a chain of function compositions which transform input vectors to output vectors. The network can be seen as a particular implementation of some function φ which approximates some other function f. The explicit form of function f is not known, we only have some implicit knowledge through the examples present in training set. The goal of learning the neural network is to approach with the φ as closely as possible to f with the expectation that φ is a good approximation of f. The difference between φ and f can be described by cost function C.

To get the better idea of the cost function let us consider simple case of mean squared error. We are given training set

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_n, y_n)\}$$

and feed-forward neural network. The $\mathbf{x}$ is training sample, the y is the target value. We introduce new set of vectors $\{\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_n\}$ which is the activation of output layer of the neural network. We want to capture how well the neural network approximates the target values from the training set. To quantify this information we define the following cost function:

$$C \equiv \frac{1}{2} \sum_{i=1}^{n} \|\mathbf{a}_i - \mathbf{x}_i\|^2. \tag{4.24}$$

As seen in previous section, there is also another definition of the cost function but the goal is still the same: to find such values of weights

and biases in neural network so that the cost is minimal.

4.4.1 Backpropagation Algorithm

The goal of backpropagation is to compute partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the cost function C with the respect to network weights w and biases b. For backpropagation to work, the cost function must have the following property:

$$C = \sum_x C_x.$$

x is a set of all training samples i.e. the total cost function is the sum of all individual cost functions for each training sample. If this assumption is true, we can calculate the partial derivatives for each sample individually. The total value of C is then a sum of individual derivatives.

The backpropagation is based on basic algebraic operation such as vector addition and matrix multiplication. One exception is the Hadamard product which is less common. The Hadamard product denoted as $\mathbf{u} \odot \mathbf{v}$ is defined as follows: $\mathbf{u} \odot \mathbf{v} = u_i * v_i, \forall i$. An example of this operation is:

$$\begin{bmatrix} a \\ b \end{bmatrix} \odot \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ bd \end{bmatrix}.$$

Let me now introduce notation used in the following paragraphs. First, w^l_{jk} is the weight from k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the $(l)^{th}$ layer. Next, b^l_j is the bias of j^{th} neuron in l^{th} layer. Finally, δ^l_j is the error of the j^{th} neuron in l^{th} layer.

Following are the backpropagation algorithm and the explanation of the equations:

1. **Input** – The activation $\mathbf{a}^1$ of the input layer is set.

2. **Feedforward** – Computes $\mathbf{z}^l = \mathbf{w}^l\mathbf{a}^{l-1} + \mathbf{b}^l;\ \forall\, l = 2, 3, \ldots, L$ and $\mathbf{a}^l = f(\mathbf{z}^l)$, where L is the number of network layers and $f(\cdot)$ is the neuron activation function.

3. **Network error** – Calculates the output layer error $\delta^L = \nabla_{\mathbf{a}} C \odot f'(\mathbf{z}^L)$. $f'(\cdot)$ is derivation the of $f(\cdot)$, the neuron activation function and $\odot$ is the Haddamard product.

4. **Backpropagation** – Propagates the error back through the network. Computes $\boldsymbol{\delta}^l = ((\mathbf{w}^{l+1})^T\boldsymbol{\delta}^{l+1}) \odot f'(\mathbf{z}^l);\ \forall\, l = L - 1, L - 2, \ldots, 2$. The input layer $l = 1$ is not affected by the error.

5. **The network parameters** – The change of cost function with respect to parameters of the network is $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

In the input step of algorithm, the activation of neurons is set the values obtained from training set. Then in the feedforward step the input sample is propagated through the whole network. For each layer is calculated the weighted input $\mathbf{z}$ and then the activation of the layer. The equation presented in step 3 is actually matrix form of following equation for the error of single neuron in output layer:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} f'(z^L_j). \tag{4.25}$$

The first term $\partial C/\partial a^L_j$ measures how fast the cost is changing as the function of the j^{th} activation. The second term measures how fast is the activation function f changing. The derivation of these two terms is more clear if we look how small change of weighted neuron input affects the final cost.

The small change Δz^l_j of the neuron's weighted input propagates

through network in the forward pass. When Δz_j^l is added to the weighted neuron input the actual neuron's activation will be $a_j^l = f(z_j^l + \Delta z_j^l)$ changing activation of all subsequent neurons. The overall amount of cost change is then $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$.

The backpropagation equation shows how to calculate the error in layer l in terms of the error in the next layer $l+1$. The transpose of weight matrix $(w^{l+1})^T$ is used because the error is propagated from through network backwards – from layer $l+1$ to layer l. Given this equation we can compute the error for any layer l in the network.

With the knowledge of the change of cost function of the network with respect to parameters, we can apply the gradient descent algorithm with following update rules:

$$\mathbf{w}^l \leftarrow \mathbf{w}^l - \alpha \sum_x \boldsymbol{\delta}^{x,l} (\mathbf{a}^{x,l-1})^T$$
$$\mathbf{b}^l \leftarrow \mathbf{b}^l - \alpha \sum_x \boldsymbol{\delta}^{x,l}$$

5 Experitments and Results

5.1 Datasets

This section conveys general information about the dataset I have considered and tested while working on this thesis. The following datasets are all of natural images. I have purposely disqualified the MNIST dataset even though many breakthrough discoveries in the field of deep learning were presented on it. The reason is that MNIST dataset comprises of images of handwritten digits and thus is more suitable for supervised classification algorithms.

5.1.1 Imgur Dataset

The first idea was to write a web crawler program which would download random images from image hosting server `Imgur.com`. To avoid potential trouble with owners of the server, I have created a simple algorithm, inspired by the operating system scheduler, which periodically updated a list of public proxy servers which were used to download the images.

Although the approach worked, it was impractical. The public web proxy servers are quite slow, usually limiting the bandwidth. Given this limitation, the mean time of image transfer was around 10 seconds. Another problem was the latency of the proxies, in some cases it took up to 50 seconds to get the response from a server. In the end, I was able to download around 80 images per hour.

5.1.2 NUS WIDE

NUS WIDE[4] is a dataset created at National University of Singapore. It contains over 250.000 images acquired from photography hosting server `Flickr.com`. There is also a number of derived datasets containing various features extracted from the images available in the NUS WIDE dataset.

The dataset is distributed as a set of urls pointing to the images. The problem with this dataset is again the potential violation of Flickr's terms of service. Another issue is that the dataset was published in 2009. During the 5 years, a significant portion of the dateset was removed from Flickr. Unfortunately, in the case of a missing image, Flickr does not respond with standard `404` HTTP response code but with custom image with message that the requested image is missing. It should be possible to automatically detect this image, as it is same every time, but I have opted for other dataset.

5.1.3 CIFAR 10

The CIFAR-10 and CIFAR-100[16] are labeled subsets of a 80 Million Tiny Images dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

The dataset consists of 60,000 32x32 colour images. Each image has one of 10 labels assigned, hence the name CIFAR 10. This dataset is widely recognised and used for testing various image classification and computer vision tasks.

I have done some tests with this dataset, but in the end I have decided to use other dataset because the 32x32 images are not suitable for printing. The image size is not actually a problem because all the algorithms I have tested are using images of this size. The only issue

is that is hard to demonstrate the results with images of this size.

5.1.4 MIRFLICKR

The MIRFLICKR[12] dataset comes in two versions. The first version MIRFLICKR25000 contains 25,000 images distributed under Creative Commons licence. The MIRFLICKR-1M contains 1 million of images. Both dataset were retrieved from photography hosting server `Flickr.com`.

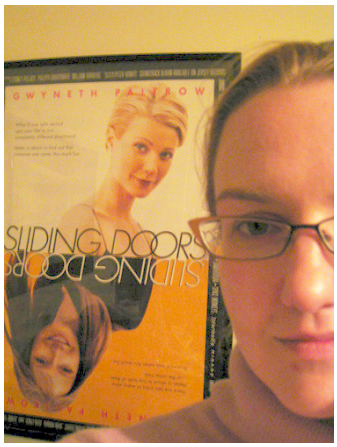


Image tags
sassy sugarpants
project
sliding doors
love
day 361
absolute clarity
365 more

Figure 5.1: Random image with its associated tags – bad tags

Every image has a set of user defined tags associated with them. These tags are often not descriptive enough to aid in the search for similar images. One example from MIRFLICKR is shown in fig. 5.1. In this case the tags are probably useful for the author of the image but not for image retrieval system.

Of course not all images have tags which are not useful. For example, in the fig. 5.2 the tags are very descriptive.

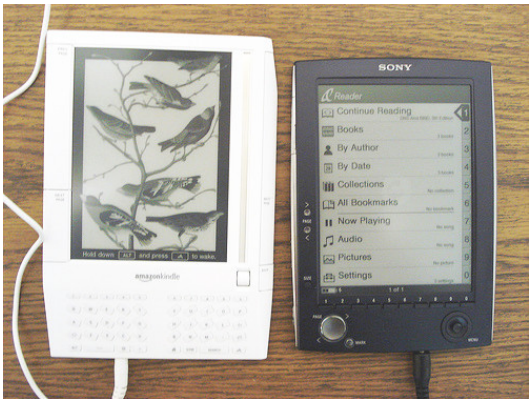


Image tags
amazon kindle
ebook
ereader
digital

Figure 5.2: Random image with its associated tags – good tags

5.2 Tools

5.2.1 Python

Python is object-oriented dynamically typed interpreted scripting language. Python is developed as an open source project. Thanks to its user-friendliness and high performance it has been utilized by many large organizations, among others for example Google, CERN, Yahoo and NASA. A variety of interpreters exists for all major platform Windows, Mac OS and Linux. Many Linux distributions are shipped with Python.

For scientific calculations, there are libraries like Numpy, Scipy and Matplotlib, whose goal is to provide mathematical computations to Python. I think that with proper libraries, Python can easily replace Matlab. There is also a number of specialized libraries designated for machine learning and artificial intelligence. The main advantage of Python is the speed of development, broad spectrum of available libraries and powerful syntax.

5.2.2 Theano

Theano [3] is a Python library designed for defining, optimizing and evaluating mathematical expressions, especially those with multidimensional arrays. Theano is much more expressive than hand crafted C code and can be faster if the generated code is run on GPU. The GPU support is to a large degree transparent to the user. The GPU code is generated by the framework itself, the user only has to use the data containers correctly.

Theano combines features of a computer algebra system with features of an optimizing compiler. This allows Theano to produce highly optimized custom code for many mathematical operations. Thanks to the optimization features, Theano can improve speed and numerical stability. Theano also features support for the symbolic features such as automatic differentiation.

5.2.3 Pylearn

Even though Theano provides wide spectrum of features with a high level of abstraction, it is still a bit cumbersome to work with. This was probably one of reasons to create the Pylearn [7] framework. If I were to compare these two I would say that Theano is to Pylearn what is assembly to C++. The main goal of Pylearn is to provide standard, easy-to-use toolkit for efficient machine learning.

Pylearn is being developed at the University of Montreal and can be seen as an abstraction layer on top of Theano (together with few other scientific libraries). The idea behind Pylearn is that user creates description of the experiment without the need to do any programming with Theano. Unfortunately, Pylearn is still being developed, with new features introduced as needed. This, and lack of proper documentation, makes it hard to use without extensive effort and reverse

engineering the framework.

5.2.4 Theanets

Theanets is another library based on top of Theano. It provides simple and clean interface for testing and experimenting with various types of neural networks. At the moment, the library contains a frame of generic feed-forward network and implementations of the classifier, autoencoder and regressor networks. More advanced networks are not available. The concept of the library is similar to the concept of Pylearn but because the scale of the library is much smaller it is much easier to use.

5.2.5 RBM Implementation

The authors of the Theano framework provided some example code for training RBMs. The code provided is only for machines with binary input units and binary hidden units, which are not suitable for working with natural images. To successfully train deep belief network with natural images it is necessary to use RBM with Gaussian visible units and binary hidden units. This machine acts like an adapter for the whole belief network.

Unfortunately my attempts of implementing one using Theano were unsuccessful so I have used pure Python implementation provided by David Warde-Farley at `https://gist.github.com/dwf/359323`. This implementation cannot use GPU but the Numpy [17] library is capable of utilizing multiple cores if available.

5.3 Experiments

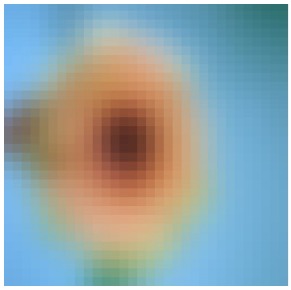
During my research I did not find any method which could objectively quantify the similarity of two images. Furthermore, the small part of the dataset I have chosen does not contain many images which are similar. With respect to this work, the similarity is the visual one, not the semantic one. That is to say that two images of sunset are considered similar because of akin colours and position of those colours. On the other hand, an image of a black cat on the street is not considered similar to an image of an orange cat in grass, even though that in both cases the key motive is the cat.

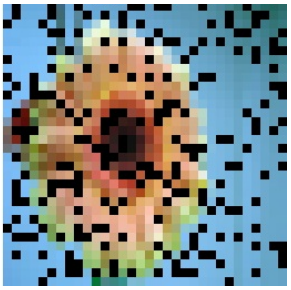
I have created a custom dataset derived from the training data. First, I have resized all images from the MIRFLICKR25000 dataset to 32 by 32 pixels. This way I have the input images consistent with those from the CIFAR dataset. At this resolution, the total number of pixels is 3072 – 1024 per colour channel – which yields reasonable times when training the neural networks. Next, I have created a copy of these images and converted them from RGB to Lab colour space. My reasoning behind this was that Lab is more suitable for expressing perceived colour similarity[14] which could improve the algorithm recall.

For the training phase I have selected 5000 images, for testing purposes I took the first 1000 images. The testing images were further altered providing 4 testing sets in total – 2 alterations, 2 colour spaces. To one set of testing images I have applied Gaussian blur with parameter $\sigma = 5$. In the second set 20% of pixels chosen randomly were set to black.

For the testing I have first trained the system using the training set of 5000 images and then I have measured the rank of the test images in the trained system. For systems where it is applicable, I have used euclidean system of low dimensional codes produced to

(a) Original image





(b) Distorted versions of image scaled up 8 ×

Figure 5.3: Test dataset example

rank the queries. If the system does not produce codes I have used corresponding rank function. The rank of query expresses the distance of the actual target image, the image that was selected as best match. This means that an image with a rank of 0 is exact match. In following subsection I will present results of the experiments.

5.3.1 How to read results

Each table is divided into 4 sectors: 2 sectors horizontally, one for Lab colour space and second for RGB colour space and 2 sectors vertically, one for blurry query images and one for noisy query images. The rank means how many images were selected as better match than the actual target. 0 means exact match, 1 means one image was closer to query than the correct target and so on. The ranks are divided into 5 groups: best three results, then ranks from 2 to 20 and the rest. All data in tables are in percent. The highest score in each row is emphasized.

5.3.2 Euclidean distance

This is the simplest method presented. I have measured the Euclidean distance between pairs of images directly. The results seem satisfactory, but these recall rates are to be expected. In this artificially constructed use case the noisy images are virtually the same as the originals. In the case of blurry images the recall rate drops by ~30% but almost all the target queries are still ranked in the top 20. The average distance of codes – in this cases pixels – are 4346 and 5043 for blurry and noisy images respectively. In the RGB colour space the average distances of codes are 4350 and 5044.

Table 5.1: Results of Euclidean distance experiment

	blur				noise			
rank	0	1	2–20	>20	0	1	2–20	>20
Lab	**69.2**	9.6	18.1	3.1	**98.3**	0.5	1.2	0
RGB	**69.2**	9.6	18.1	3.2	**98.1**	0.5	1.3	0.1

5.3.3 PCA

In this experiment I have first fitted the model with 1000 training images. These images were then projected into a given number of principal components. From the model parameters I have found that 99 principal components contains $> 99\%$ of the input variance. I have chosen two more numbers of principal components to test the algorithm under different conditions. For the noisy images, the recall is almost the same as the values for euclidean distance. The recall for blurry images improved by ~40% in comparison to the euclidean distance (table 5.1). The number in first column in table 5.2 is the number of principal components used.

Table 5.2: Results of PCA experiment

	blur				noise			
rank	0	1	2–20	>20	0	1	2–20	>20
Lab								
33	**98.3**	0.5	0.9	0.3	**88.6**	4.1	67	6
99	**89.2**	3.6	6.3	0.9	**95.6**	1.7	24	3
297	**73.8**	7.9	15.1	3.2	**97.6**	1	13	1
RGB								
33	**98.3**	0.5	0.9	0.3	**87.1**	4	80	9
99	**89.2**	3.5	6.4	0.9	**94.7**	2.2	29	2
297	**73.8**	7.9	15	3.3	**96.9**	1.2	18	1

5.3.4 Autoencoder and Denoising Autoencoders

In neural network experiments I have tested different settings of neuron activation and number of hidden units. In case of the autoencoder network, the network learned the identity function in all test cases. When the network learns the identity function, the reconstruction error is low but all codes extracted from the hidden units are same. As

a code, I have used the activation of the hidden units stimulated with an input sample.

For the denoising autoencoder, in addition to various activation functions and sizes of hidden layer, I have added the noise ratio parameter. This parameter expresses how much of the input neurons will be randomly set to zero. In this case, the networks have to reconstruct the original data from the corrupted input. This is the exact use case of my tests, unfortunately for the parameters I have chosen the network learned mostly identity function. In table 5.3 the first column from the left is activation function, second is number of hidden units and the third is the ratio of noise added to input units. I have run the calculations for all combinations of parameters: three activation functions (linear, sigmoid, tanh), three counts of hidden units (33, 99, 297) and three setting of the noise parameter (0.1, 0.3, 0.5). In the table 5.3 I have omitted the rows where the result was identity function.

The first column marks the activation function and number of hidden units, in second column are the values of noise parameter.

Table 5.3: Results of denoising autoencoder experiment

Lab		blur				noise			
	rank	0	1	2–20	>20	0	1	2–20	>20
sigmoid									
297	0.3	0.6	0.2	3.2	**96.0**	0.7	0.3	2.7	**95.9**
	0.5	0.2	0.2	3.6	**96.0**	0.3	0.2	3.2	**96.3**
tanh									
99	0.3	0.2	0.2	3.4	**96.2**	0.2	0.0	3.2	**96.6**
	0.5	0.2	0.2	3.5	**96.1**	0.2	0.2	3.5	**96.1**
297	0.1	1.3	1.2	17.1	**80.4**	1.1	0.9	17.9	**80.1**
RGB		blur				noise			
	rank	0	1	2–20	>20	0	1	2–20	>20
tanh									
99	0.3	0.3	0.2	2.7	**96.4**	0.1	0.2	3.5	**96.2**
297	0.5	1.6	1.8	16.6	**80.0**	1.6	1.4	16.4	**80.6**

5.3.5 Deep Belief Network

For the deep belief network I have used the deep autoencoder with the following layer sizes: [3072, 8192, 4096, 2048, 1024, 512, 256, 128, 64]. Where the the first layer on the left (3072) is the visible layer and last is the deepes layer. Each layer was first pretrained using the dedicated RBM and the whole autoencoder was fine tuned. For my network I have chosen the same parameter as Hinton in [10].

The results of the DBN testing are comparable with result from the PCA with respect to recall. The DBN actually provides much better results than the PCA, as seen from the average code distance.

For PCA with 297 principal components the average code distance is ~4300. In the case of the DBN, the average distance of the codes is ~3; three orders of magnitude smaller. This means that the code for corrupted queries are much closer to desired target in case of the DBN which means better accuracy.

Table 5.4: Results of DBN experiment

	blur				noise			
rank	0	1	2–20	>20	0	1	2–20	>20
Lab								
64	18.9	6.6	**41.4**	33.1	21.8	9.9	**48.7**	19.6
128	33.5	10.5	**36.3**	19.7	**41.7**	13.9	34.5	9.9
256	**70.2**	12.4	16.3	1.1	**77.6**	11.5	9.8	1.1
512	**89.4**	5.9	4.7	0.0	**92.2**	4.3	3.2	0.3
1024	**95.6**	2.5	1.9	0.0	**95.9**	2.3	1.5	0.3
RGB								
64	0.1	0.1	1.8	**98.0**	0.1	0.1	1.8	**98.0**
128	0.1	0.1	1.8	**98.0**	0.1	0.1	1.8	**98.0**
256	11.4	5.2	28.0	**55.4**	5.6	3.1	19.3	**72.0**
512	**44.3**	9.3	28.3	18.1	**34.3**	8.2	31.2	26.3
1024	**61.0**	9.3	19.3	10.4	**70.4**	8.5	15.3	5.8

The main disadvantage of using the DBN is the performance. The training of stack of RBMs took ~22 hours to train the network on 8 core CPU (Intel i7-2600K @ 3.4GHz) on 5000 images. This duration would be definitely shorter in a case of training on GPU, but I was unable to create working GPU implementation of the RBM network.

5.3.6 Multi-resolution Image Querying

In this experiment, I have created database with 1000 testing images. Then I have used these images to tune the weights of the ranking function (see eq. (3.10) on page 14) using the logistic regression. The number of buckets was fixed at 4. I have tested the recall for various values of the parameter m and also for a database for which all the weights were set to 1.

The tuning of the parameters especially improved the recall for blurry images for lower values of m. On the other hand, the recall

Table 5.5: Results of Multi-resolution Image Querying experiment. Metric weights are set to 1.

	blur				noise			
rank	0	1	2–20	>20	0	1	2–20	>20
Lab								
20	2.6	1.1	26.6	**69.7**	5.2	3.0	35.9	**55.9**
40	37.2	12.6	**41.6**	8.6	36.0	14.1	**38.2**	11.7
60	**78.9**	7.2	11.8	2.1	**56.2**	15.0	23.5	5.3
80	**89.8**	3.7	5.5	1.0	**63.1**	12.5	20.0	4.4
100	**93.6**	2.4	3.4	0.6	**66.2**	11.0	19.4	3.4
120	**95.6**	1.5	2.4	0.5	**69.7**	9.9	17.5	2.9
RGB								
20	1.9	1.3	25.9	**70.9**	5.5	2.6	35.8	**56.1**
40	36.4	12.5	**42.3**	8.8	36.7	13.1	**38.8**	11.4
60	**76.6**	8.5	12.9	2.0	**55.3**	15.5	23.7	5.5
80	**87.9**	4.2	6.3	1.6	**62.5**	12.7	20.4	4.4
100	**92.4**	3.0	3.6	1.0	**66.1**	11.3	19.1	3.5
120	**95.1**	1.5	2.7	0.7	**69.4**	9.8	17.6	3.2

Table 5.6: Results of Multi-resolution Image Querying experiment. Metric weights are tuned using logistic regression.

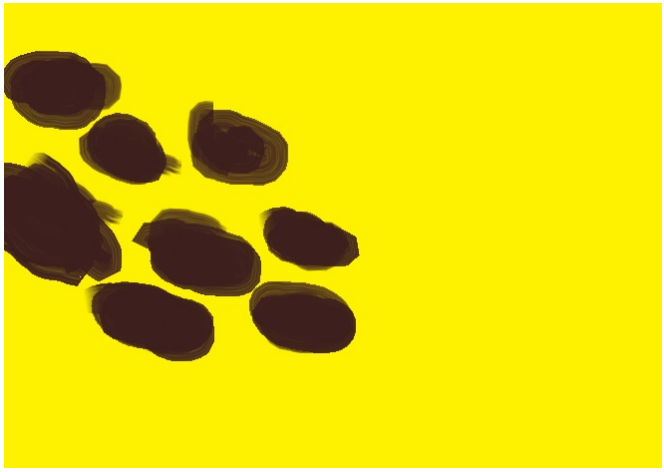
	blur				noise			
rank	0	1	2–20	>20	0	1	2–20	>20
Lab								
20	**73.6**	10.6	13.9	1.9	2.4	1.2	8.8	**87.6**
40	**90.8**	4.5	3.5	1.2	5.1	2.5	12.9	**79.5**
60	**94.4**	3.2	2.0	0.4	9.4	2.5	13.8	**74.3**
80	**95.1**	2.5	2.1	0.3	14.3	2.4	14.0	**69.3**
100	**95.6**	2.3	1.9	0.2	17.4	3.0	13.5	**66.1**
120	**96.6**	1.9	1.4	0.1	20.9	3.5	14.0	**61.6**
RGB								
20	**74.8**	9.8	13.5	1.9	2.2	1.2	9.2	**87.4**
40	**90.9**	4.8	3.1	1.2	5.8	2.1	11.1	**81.0**
60	**94.9**	2.8	1.9	0.4	9.3	3.0	13.1	**74.6**
80	**95.2**	2.6	1.9	0.3	14.0	3.2	12.9	**69.9**
100	**95.9**	2.3	1.6	0.2	17.6	2.4	15.0	**65.0**
120	**96.5**	2.1	1.3	0.1	20.2	2.9	16.2	**60.7**

rate for noisy images dropped significantly. This is caused by the training on the clear images.

As the last experiment, I have tested the ability of this method to retrieve target images based on a hand-painted query. I have selected 18 random images from the training set and painted them in MS Paint. I was painting using the mouse and from memory so the paintings are very inaccurate. From the 18 painted queries the system was able to correctly identify 9. These 9 queries had rank 0 which means an exact match.

(a) Painted query image (b) Target dataset image

Figure 5.4: Painted query test



(a) Painted query image

(b) Target dataset image

Figure 5.5: Painted query test

6 Conclusion

The original assignment for this work was to find a metric which allows to objectively measure the similarity of two images. I have to admit that the only algorithm capable of this, that I have tested, is the Euclidean distance. All the other methods either provide relative information or does not work objectively. On the other hand, I think that it is impossible to find such metric without defining what is the similarity of two images. There are many ways of how to approach this problem. The main two approaches, that I can think of, are the semantic approach and the content based approach. In the case of the semantic approach, the images are considered similar if they can be described by the same and/or similar keywords. Assigning these keywords is still an open problem. The other approach is based on processing the actual content of the image, either the pixels directly or some extracted features.

In this work, I have focused on the content based approach, or more specifically, on extracting low dimensional codes from the images. These codes can be used to find images with similar codes which should be similar. I have found out that neural networks with shallow architecture are not suitable for this task. At least when used directly. The deep belief networks have shown a big potential which is supported by the results of recent research in the field of deep learning.

Two main problems with neural network caused the poor results. Neural networks need a big amount of data to learn, extracting useful codes from natural images in unsupervised manner. Together with big amount of data, the neural networks demand extended periods of time for learning, which slows down the research and testing significantly. The other significant problem with neural networks is the scalability. Codes extracted by neural network are useful for closed static set of

images only. If the set should be updated regularly, while learning on the new data, then the existing codes would have to be discarded and the new codes calculated again. For this reason, I do not think it would be sensible to use a system backed by neural network as, for example, a web search engine.

The method based on Haar wavelet transform works surprisingly well. The main issue that I see with this approach is, again, a lack of training data. Compared to the neural network approach, this method has many benefits. It can work in online manner, adding new images to database on the fly. It is easy to tune to specific domain of queries. And it provides better results with less data, so the initial cost of deployment is low in the term of resources.

I would say that neural networks definitely have a bright future. And there are fields where other approaches does not work. But for such an open ended problem, as the image similarity assessment truly is, the neural networks, as I have used them, are not the best tool available.

Bibliography

[1] Yoshua Bengio. "Learning deep architectures for AI". In: *Foundations and trends in Machine Learning* 2.1 (2009), pp. 1–127.

[2] Yoshua Bengio et al. "Greedy layer-wise training of deep networks". In: *Advances in neural information processing systems* 19 (2007), p. 153.

[3] James Bergstra et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX, June 2010.

[4] Tat-Seng Chua et al. "NUS-WIDE: A Real-World Web Image Database from National University of Singapore". In: *Proc. of ACM Conf. on Image and Video Retrieval (CIVR'09)*. Santorini, Greece., July 8-10, 2009.

[5] Matthijs Douze et al. "Evaluation of gist descriptors for web-scale image search". In: *Proceedings of the ACM International Conference on Image and Video Retrieval*. ACM. 2009, p. 19.

[6] `gettyimages.com`. *Getty Images unveils innovative embed feature for sharing of tens of millions of images*. `http://press.gettyimages.com/getty-images-unveils-innovative-embed-feature-for-sharing-of-tens-of-millions-of-images/`. Mar. 2014.

[7] Ian J. Goodfellow et al. “Pylearn2: a machine learning research library”. In: *arXiv preprint arXiv:1308.4214* (2013). URL: `http://arxiv.org/abs/1308.4214`.

[8] Geoffrey Hinton. “A practical guide to training restricted Boltzmann machines”. In: *Momentum* 9.1 (2010), p. 926.

[9] Geoffrey E. Hinton. “Learning multiple layers of representation”. In: *Trends in cognitive sciences* 11.10 (2007), pp. 428–434.

[10] Geoffrey E Hinton and Ruslan R Salakhutdinov. “Reducing the dimensionality of data with neural networks”. In: *Science* 313.5786 (2006), pp. 504–507.

[11] Jing Huang and Ramin Zabih. “Combining color and spatial information for content-based image retrieval”. In: *Proceedings of ECDL*. Vol. 1998. 1998.

[12] Mark J. Huiskes and Michael S. Lew. “The MIR Flickr Retrieval Evaluation”. In: *MIR ’08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval*. Vancouver, Canada: ACM, 2008.

[13] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. “Fast Multiresolution Image Querying”. In: *Proceedings of SIGGRAPH 95*. Aug. 1995, pp. 277–286.

[14] Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, 1989, p. 73.

[15] Neal Krawetz. *Looks Like It*. `http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html`. cited in April 2014. May 2011.

[16] Alex Krizhevsky and Geoffrey Hinton. “Learning multiple layers of features from tiny images”. In: *Computer Science Department, University of Toronto, Tech. Rep* (2009).

[17] Travis E. Oliphant. “Python for Scientific Computing”. In: *Computing in Science & Engineering* 9.33 (May 2007), pp. 10–20. URL: `http://www.numpy.org`.

[18] Aude Oliva and Antonio Torralba. “Modeling the shape of the scene: A holistic representation of the spatial envelope”. In: *International journal of computer vision* 42.3 (2001), pp. 145–175.

[19] Dong Kwon Park, Yoon Seok Jeon, and Chee Sun Won. “Efficient Use of Local Edge Histogram Descriptor”. In: *Proceedings of the 2000 ACM Workshops on Multimedia*. MULTIMEDIA ’00. Los Angeles, California, USA: ACM, 2000, pp. 51–54.

[20] Raúl Rojas. *Neutral Networks: A Systematic Introduction*. Springer, 1996.

[21] Jonathon Shlens. “A tutorial on principal component analysis”. In: *Systems Neurobiology Laboratory, University of California at San Diego* 82 (2005).

[22] Li Weng and B. Preneel. “Attacking Some Perceptual Image Hash Algorithms”. In: *2007 IEEE International Conference on Multimedia and Expo*. July 2007, pp. 879–882.

[23] Filip Zlámal. “Logistická regrese v R”. Bakalářská práce. Masarykova univerzita, Přírodovědecká fakulta, 2013.

List of Abbreviations

- **DCT** – discrete cosine transform, a mathematical transform related to the Fourier transform
- **JPEG** – a method of compression of digital photographs
- **GIST** – a low dimensional representation of the scene, which does not require any form of segmentation
- **SIFT** – Scale-invariant feature transform, an algorithm in computer vision to detect and describe local features in images
- **DWT** – discrete wavelet transform, a mathematical procedure in numerical analysis and functional analysis
- **PCA** – principal component analysis, a statistical procedure
- **SNR** – signal-to-noise ratio, a measure that compares the level of a desired signal to the level of background noise
- **SVD** – singular value decomposition, a matrix factorization method
- **ANN** – artificial neural network, a computational model inspired by a biological nervous systems
- **EBM** – energy based model, see section 4.3.4 on page 38
- **BM** – Boltzmann machine, a type of stochastic recurrent neural network
- **RBM** – restricted Boltzmann machine, a generative stochastic neural network
- **DBN** – deep belief network, type of neural network/graphical model in machine learning

- **GPU** – graphics processing unit, a special stream processor used in computer graphics hardware
- **RGB** – RGB colour model is an additive colour model
- **Lab** – Lab colour space, a colour space based on nonlinearly compressed CIE XYZ color space

A MIRFLICKR Top 20 Tags

Table A.1: 20 most frequent tags in MIRFLICKR dataset

tag	count	tag	count
explore	1483	night	621
sky	845	nature	596
nikon	805	sunset	585
2007	794	green	569
blue	761	clouds	558
bw	737	macro	547
canon	686	light	516
water	641	flower	510
red	623	big fave	469
portrait	623	white	431

B User Guide

This guide is intended to provide some basic information on how to run the included software. Detailed program documentation is included with the software.

B.1 Requirements

Even though Python is multiplatform the recommended platform is Linux.

- Python – all code is written in Python, and was tested with Python 2.7. The support for Python 3 is not guaranteed.
- Theano – this framework can be obtained at `http://deeplearning.net/software/theano/install.html` the linked page includes detailed installation instructions. If you want to run the code on GPU you will also need computer with NVIDIA graphics card with NVIDIA CUDA drivers and SDK installed.
- Theanets – framework built on top of Theano can be obtained from `https://github.com/lmjohns3/theano-nets`
- RBM code – code for training and testing RBMs and DBNs can be obtained from `https://gist.github.com/dwf/359323`
- dataset – it is possible to use any set of images. The datasets I have used are listed in section 5.1 on page 48.
- Imagemagick suite – this tool is needed for preparation of the data, namely resizing and converting colour space.

B.2 Installation

1. Install and test all required libraries. Then copy all files to target destination in system and run script `setup_vars.sh` this script update environment variables for easier use of the programs.

2. Download dataset and unpack it in desired location. Then run the conversion script. The script is called with one parameter – location of the data. The script require all images to be in JPEG format. The conversion script creates necessary directories and prepares the images. When the script run successfully it will create resized copy of the dataset with resized images. Copy of dataset in Lab colour space with resized images and two test datasets for each colour space. Next the script creates two gzipped Pickle files for each of four datasets. One file contains all images in Numpy ndarray format, the second contains images from dataset in transformed into vector and stored in matrix.

B.3 Running experiments

Experiments are divided into directories named according to their method. Each directory contains two Python scripts. One for building the experiment and second for testing. Only exception is the Euclidean distance experiment, where is not any model to build. In the build phase the script creates the model of the experiment and stores it in gzipped file. The test code then runs a test on this model and outputs results. The test code expects the model with certain filename, so it is not recommended to change the generated files.

The build phase can be quite long and performance demanding.

Especially the code for building DBN model. During my tests the code run for 22 hours. The resulting models are fairly small, but the size of model depends on the parameters of model. In case of neural networks in the files are stored weights and biases for the networks and some internal metadata. As a rule of thumb the model of neural network has at least (`layer_in`×`layer_out`+`bias_in`+`bias_out`)×8 bytes. The actual size will probably be smaller thanks to gzip compression. At least 1GB is necessary for the DBN with parameters presented in this work. Other models are much smaller.

B.4 Results

The output of the test script is text file woth collected results. The file has 2 columns separated by space. The first column is rank of the sample, the second column is the distance of code of sample and code of target. Each line represents single sample from test dataset. The line number corresponds to id of sample, e.g. first sample is on first row and so on. When evaluating the results bear in mind that Python and Numpy internally uses C-like array indexing where firt element has index 0.