Artificial neural networks - Exercise session 1

Supervised learning and generalization

Bram De Cooman, Hannes De Meulemeester, Joachim Schreurs and David Winant

2019-2020

1 The perceptron

The perceptron is the simplest one-layer network. It consists of R inputs connected to 1 neuron arranged in a single layer via interconnection weights. These neurons have the `hardlim` transfer function, thus the output values are only 0 and 1, where the inputs can take on any value. The perceptron is used as a simple classification tool.

In order to create such a network, we can use the command:

```
net = newp(P,T,TF,LF)
```

where `P` and `T` are input and target vectors e.g. `P=[2 1 -2 -1; 2 -2 2 1]`, `T=[0 1 0 1]`. `TF` is the transfer function (typically 'hardlim'), `LF` represents the perceptron learning rule (for instance 'learnp'). In this case the number of neurons is set automatically.
The weights of the connections and the bias of the neurons are initially set to zero, which can be checked and changed with the commands

`net.IW{1,1}`	Returns the weights of the neuron(s) in the first layer
`net.b{1,1}`	Returns the bias of the neuron(s) in the first layer
`net.IW{1,1} = rand(1,2);`	Assigns random weights in [0,1]
`net.b{1,1} = rands(1);`	Assigns random bias in [-1,1]

One can initialize the network with a single command by issuing:

```
net = init(net);
```

all weights and biases of the perceptron will be initialized to zero.

We can teach a perceptron to perform certain tasks which are defined by pairs of inputs and outputs. A single perceptron can learn only linearly separable tasks: inputs belonging to different classes are separated by a hyperplane in the input space (decision boundary). In two dimensions the decision boundary is a line. The default learning function is the perceptron learning rule `learnp`.

We can use the function `train` to let the perceptron perform the classification task. In this case the learning occurs in batch mode:

```
[net,tr_descr] = train(net,P,T);
```

here the second argument is a description of the learning process.
To set the number of iterations or epochs, we can use the command

```
net.trainParam.epochs = 20;
```

After training we can simulate the network on new data with the function `sim`:

```
sim(net,Pnew)
```

with `Pnew` an input vector. For our example `Pnew` has to be a (column) vector of length 2, with elements between -2 and 2, e.g. `Pnew = [1;-0.3]`. Multiple input vectors can be fed at once to the network by putting them together in an array.

Demos

The following demos can be run from the MATLAB prompt.

`nnd4db`	decision boundary (2d input)
`nnd4pr`	perceptron learning rule (2d input)
`demop1`	classification with 2d input perceptron
`demop4`	classification with outlier (2d input)
`demop5`	classification with outlier using normalized perceptron learning rule (2d input)
`demop6`	linearly non-separable input vectors (2d input)

Exercises

Create a perceptron and train it with examples (see demos). Visualize results. Can you always train it perfectly?

Functions and commands

`newp(P,T,TF,LF)`	Creates a perceptron with the right number of neurons, based on input values P, target vector T, and with transfer function `TF` and learning function `LF`.
`init(net)`	Initializes the weights and biases of the perceptron.
`adapt(net,P,T)`	Trains the network using inputs P, targets T and some online learning algorithm.
`train(net,P,T)`	Trains the network using inputs P, targets T and some batch learning algorithm.
`sim(net,Ptest)`	Simulates the perceptron using inputs Ptest.
`learnp, learnpn`	Perceptron and normalized perceptron learning rules
`hardlim`	Transfer function

2 Backpropagation in feedforward multi-layer networks

A general feedforward network consists of at least one layer, and it can also contain an arbitrary number of hidden layers. Neurons in a given layer can be defined by any transfer function. In the hidden layers usually nonlinear functions are used, e.g. `tansig` or `logsig`, and in the output layer `purelin`. The only important condition is that there is no feedback in the network, neither delay.
In MATLAB one can create such a network object by e.g. the following command:

```
net = feedforwardnet(numN,trainAlg);
```

This will create a network of one hidden layer with corresponding `numN` neurons, which will use the trainAlg algorithm for training (e.g. `traingd`). This network can be trained using the `train` function:

```
net = train(net,P,T);
```

Finally the network can be simulated in two ways:

```
sim(net,P);
```

or,

```
Y = net(P);
```

Some available training algorithms are:

`traingd`	gradient descent
`traingda`	gradient descent with adaptive learning rate
`traincgf`	Fletcher-Reeves conjugate gradient algorithm
`traincgp`	Polak-Ribiere conjugate gradient algorithm
`trainbfg`	BFGS quasi Newton algorithm (quasi Newton)
`trainlm`	Levenberg-Marquardt algorithm (adaptive mixture of Newton and steepest descent algorithms)

To analyze the efficiency of training one can use the function `postreg` which calculates and visualizes regression between targets and outputs. For a network `net` trained with a sequence of examples `P` and targets `T` we have:

```
a=sim(net,P);
[m,b,r]=postreg(a,T);
```

where `m` and `b` are the slope and the y-intercept of the best linear regression respectively. `r` is a correlation between targets `T` and outputs `a`.

Demos

`nnd11nf`	network function
`nnd11bc`	backpropagation calculation
`nnd11fa`	function approximation
`nnd12sd1`	steepest descent backpropagation
`nnd12sd2`	steepest descent backpropagation with various learning rates
`nnd12mo`	steepest descent with momentum
`nnd12vl`	steepest descent with variable learning rate
`nnd12cg`	conjugate gradient backpropagation
`nnd9mc`	comparison between steepest descent and conjugate gradient

Exercises

- Function approximation: comparison of various algorithms:
  Take the function $y = \sin(x^2)$ for $x = 0 : 0.05 : 3\pi$ and try to approximate it using a neural network with one hidden layer. Use different algorithms. How does gradient descent perform compared to other training algorithms?
  Use following examples as a basis:

  `algorlm1` (can be found on Toledo): Script that compares the performance of Levenberg-Marquardt 'trainlm' and quasi-Newton backprogration 'trainbfg' algorithms.

- Learning from noisy data: generalization
  The same as in the previous exercise, but now add noise to the data (using `randn`). Compare the performance of the network with noiseless data. You may have to increase the number of data.

- Personal Regression example:

  - **Problem Description and data preparation:** In this problem, the objective is to approximate a nonlinear function using a feedforward artificial neural network. The nonlinear function is unknown, but you are given a set of 13 600 datapoints uniformly sampled from it.

    You have to build your individual dataset from 5 existing nonlinear functions $f_1, f_2, \ldots, f_5$. In the given matlab datafile, you will find the following variables: $X1, X2, T1, T2, T3, T4$ and $T5$. The vectors $X1$ and $X2$ contain the input variables (in the domain $[0, 1]\times[0, 1]$). The vectors $T1$ to $T5$ are the 5 independent nonlinear functions evaluated at the corresponding point from $(X1, X2)$. In other words, $f_1(X1(i), X2(i)) = T1(i)$, $f_2(X1(i), X2(i)) = T2(i)$, $\ldots$, $f_5(X1(i), X2(i)) = T5(i)$, for $i = 1, \ldots, N$, where N is the length of the vectors mentioned above. The datapoints are noise free (the evaluation is exact).

    You have to build a new target $Tnew$, which represents an individual nonlinear function to be approximated by

your neural network. For this, consider the largest 5 digits of your student number in descending order, represented by d_1, d_2, d_3, d_4, d_5 (with d_1 the largest digit). You have to build your individual target as follows:

$Tnew = (d_1T1 + d_2T2 + d_3T3 + d_4T4 + d_5T5)/(d_1 + d_2 + d_3 + d_4 + d_5)$.

For example, if your student number is m0224908, then your list of largest digits in descending order is 9, 8, 4, 2, 2 and therefore your target is:

$Tnew = (9T1 + 8T2 + 4T3 + 2T4 + 2T5)/(9 + 8 + 4 + 2 + 2)$.

– **Exercises:** The problem can be split into three tasks:

1. Define your datasets: your dataset consists now of $X1, X2$ and $Tnew$. Draw 3 (independent) samples of 1 000 points each. Use them as the training set, validation set, and test set, respectively. Motivate the choice of the datasets. Plot the surface of your training set using the Matlab functions `scatteredInterpolant`, `plot3` and `mesh`.
2. Build and train your feedforward Neural Network: use the training and validation sets. Build the ANN with 2 inputs and 1 output. Select a suitable model for the problem (number of hidden layers, number of neurons on each hidden layer). Select the learning algorithm and the transfer function that may work best for this problem. Motivate your decisions. When you try different networks, clearly say at the end which one you would select as the best for this problem and why.
3. Performance Assessment: evaluate the performance of your selected network on the test set. Plot the surface of the test set and the approximation given by the network. Plot the error level curves. Compute the Mean Squared Error on the test set. Comment on the results and compare with the training performance. What else could you do to improve the performance of your network?

3 Understanding Bayesian Inference: the Case of Network Weights

For simplicity of exposition, we begin by considering the training of a network for which the architecture is fixed in advance. More precisely we focus on the case of a one-neuron network. In the absence of any data, the distribution over weight values is described by a prior distribution denoted $p(w)$, where w is the vector of adaptive weights (normally also biases, but in our example we will consider the case without bias). We also denote by D the available dataset. Once we observe the data, we can write the expression for the posterior probability distribution of the weights using Bayes theorem:

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)} \tag{1}$$

where $p(D)$ is a normalization factor ensuring that $p(w|D)$ gives unity when integrated over the whole weight space. $p(D|w)$ corresponds to the likelihood function used in maximum likelihood techniques.
Since in the beginning we do not know anything about the data, the prior distribution of weights is set to (for example) a Gaussian distribution. The expression of the prior is then:

$$p(w) = (\frac{\alpha}{2\pi})^{\frac{W}{2}} \exp(-\frac{\alpha}{2}||w||^2) \tag{2}$$

with W the number of weights, and α is the inverse of the variance. For simplicity we will choose $\alpha = 1$.
This choice of the prior distribution encourages weights to be small rather than large, which is a requirement for achieving smooth network mappings. So when $||w||$ is large, the parameter of the exponential is large, and thus $p(w)$ is small (small probability that this is the correct choice of weights. Things are reversed for small $||w|$).

A concrete example: binary classification We consider the case where input vectors are 2-dimensional $x = (x_1, x_2)$, and we have four data points in our dataset as in figure 1. We take a network of a single neuron (compare to the perceptron of the first exercise session), so there is only a single layer of weights, and choose the logistic function as transfer function:

$$y(x; w) = \frac{1}{1 + \exp(-w^T x)} \tag{3}$$

The weight vector $w = (w_1, w_2)$ is two-dimensional and there is no bias parameter. We choose a Gaussian prior distribution for the weights (with $\alpha = 1$). This prior distribution is plotted in figure 2.
The data points can belong to one of the two classes (cross or circle), the output y giving the membership to one of these classes. The likelihood function $p(D|w)$ in Bayes theorem will be given by a product of factors, one for each data point, where each factor is either y or $(1 - y)$ according to whether the data point belongs to the first or the second class.

First we consider just the points labeled (1) and (2). Then we consider all four points and recompute the posterior distribution of the weights. Note: For an example of this case, run demo `bayesNN`. After training with only the first two data points, we see that the network function is a sigmoidal ridge (w_1 and w_2 control the orientation and the slope of this sigmoid). Weight vectors from approximately half of the weight space will have probabilities close to zero, as they represent decision boundaries with wrong orientation.When using all 4 data points, there is no boundary to classify all 4 points correctly. The most probable solution is the one corresponding to the peak point of the sigmoid, the others having quite low probabilities (so the posterior distribution of the weights is relatively narrow).

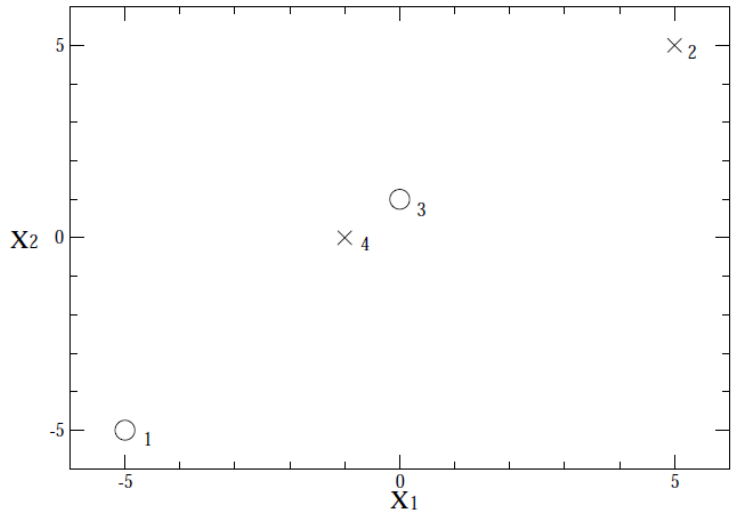


Figure 1: The four numbered data points in the dataset.

4 Bayesian Inference of Network hyperparameters

From an optimization point of view the training is performed by iteratively adjusting w so as to minimize an objective function $M(w)$. In a maximum likelihood framework, M is taken to be the error function $E_D(w)$ that depends on the data D. A common prescription to avoid overfitting is to include in M a regularization term $E_W(w)$ (a.k.a. weight decay) to favor small values of the parameter vectors, as discussed above. In particular, if we let

$$M(w) = \beta E_D(w) + \alpha E_W(w) \tag{4}$$

we can immediately give to this a Bayesian interpretation. In fact it is not difficult to show that βE_D can be understood as minus the log likelihood for a noise model whereas αE_W can be understood as minus the log prior on the parameter vector. Correspondingly, the process of finding the optimal weight vector minimizing M can be interpreted as a Maximum a Posteriori (MAP) estimation. Notice that we have implicitly considered α and β fixed here. However these are hyperparameters that control the complexity of the model. Once more within a Bayesian framework one can apply the rules of probability and find these parameters accordingly. In MATLAB for a general feedforward neural network this can be accomplished by means of `trainbr`. This training function updates the weight according to Levenberg-Marquardt optimization. It minimizes (4) with respect to w and determines at the same time the correct combination of the two terms so as to produce a network that generalizes well.

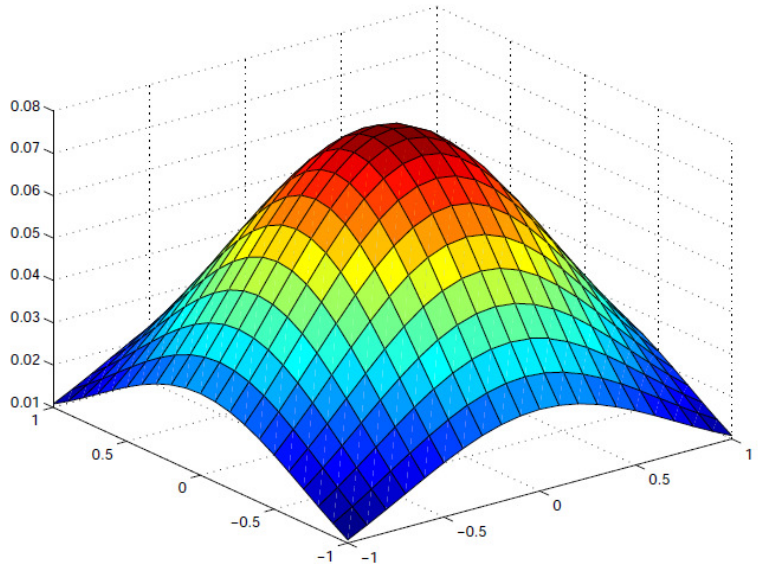


Figure 2: The prior distribution: a Gaussian.

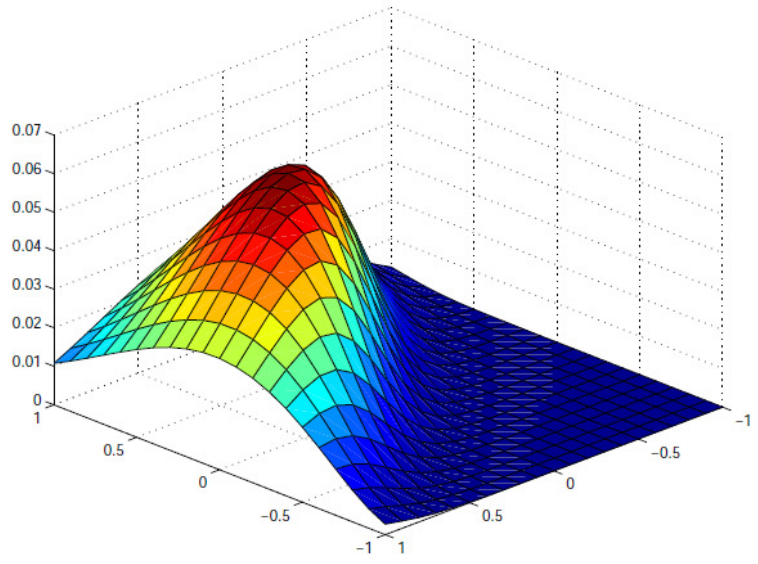


Figure 3: The distribution after presenting all four data points. Half of the weight space has become very improbable.

5 Exercises

- Use `trainbr` to analyse the first two datasets of Exercise Session 1, section 2 (the function $y = \sin(x^2)$ and the noisy version) and compare it with the training functions seen during the first exercise session. Compare the test errors. Consider overparametrized networks (many neurons): do you see any improvement with `trainbr`?

6 Report

Based on the previous exercises of section 2 and 5, write a report of maximum 3 pages (including text + figures) to discuss speed, overfitting, generalization of different learning schemes.

References

[1] H. Demuth and M. Beale, Neural Network Toolbox (user's guide),
`http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/nnet.shtml`

[2] C. M. Bishop, `Neural Networks for Pattern Recognition`, Oxford University Press.

[3] Lecture slides and references therein.

Artificial neural networks - Exercise session 2

Recurrent neural networks

Bram De Cooman, Hannes De Meulemeester, Joachim Schreurs and David Winant

2019-2020

1 Hopfield Network

A Hopfield recurrent network has one layer of N neurons with `satlins` transfer functions, and is fully interconnected: each neuron is connected to every other neuron. After initialization of all neurons (the initial input), the network is let to evolve in a synchronous way: an output at time t becomes an input at time $t + 1$. Thus to generate a series of outputs we have to provide only one initial input. In the course of this dynamical evolution the network should reach a stable state (an attractor), this is a configuration of neuron values which is not changed by an update of the network. Networks of this kind are used as models of associative memory. After initialization the network should evolve to the closest attractor.

Creation of a Hopfield network with N neurons:

```
net = newhop(T);
```

where `T` is a $N \times Q$ matrix containing Q vectors with components equal to ± 1. This command will create a recurrent Hopfield network with stable points being the vectors from `T`. For a 2-neuron network with 3 attractors `[1 1] [-1 -1] [1 -1]`, `T` has the form[1]:

```
T = [1 1; -1 -1; 1 -1]';
```

We can simulate a Hopfield network in two modes:

Single step iteration

```
Y = net([],[],Ai);
```

Multiple step iteration

```
Y = net({num_step},{},Ai);
```

with example inputs

```
Ai = [0.3 0.6; -0.1 0.8; -1 0.5]';
```

If `Y == Ai`, the columns of `Ai` are attractors of the network `net`.

Demos

Run the following demos (see Toledo):

`demohop1`	A two neuron Hopfield network
`demohop2`	A Hopfield network with unstable equilibrium
`demohop3`	A three neuron Hopfield network
`demohop4`	Spurious stable points

[1] The operator `'` or T transposes the matrix or vector. It is often more clear to write down a matrix row by row and then transpose it so the rows become columns. In above example the equivalent would be `[1 -1 1; 1 -1 -1]`

Exercise

- Create a Hopfield network with attractors `T` $= [1\ \ 1; -1\ -1; 1\ -1]^{\mathrm{T}}$ and the corresponding number of neurons. Start with various initial vectors and note down obtained attractors after a sufficient number of iterations. Is the number of real attractors bigger than the number of attractors used to create the network? How many iterations does it typically take to reach the attractor?
- Execute script `rep2`. Modify this script to start from some particular points (e.g. of high symmetry) or to generate other numbers of points. Are the attractors always those stored in the network at creation?
- Do the same for a three neuron Hopfield network. This time use script `rep3`.
- The function `hopdigit` creates a Hopfield network which has as attractors the handwritten digits $0, \cdots, 9$. Then to test the ability of the network to correctly retrieve these patterns some noisy digits are given to the network. Is the Hopfield model always able to reconstruct the noisy digits? If not why?

  You can call the function by typing:

  ```
  hopdigit(noise,numiter)
  ```

  where:

  `noise` represents the level of noise that will corrupt the digits and is a number between 0 and 10

  `numiter` is the number of iterations the Hopfield network (having as input the noisy digits) will run.

  Try to answer the above question by playing with these two parameters.

2 Long Short-Term Memory Networks

2.1 Introduction: Time-series Prediction

A time series is a sequence of observations, ordered in time. Forecasting involves training a model on historical data and using them to predict future observations. A simple example is a linear auto-regressive model. The linear auto-regressive (AR) model of a time-series Z_t with $t = 1, 2, \ldots, \infty$ is given by:

$$\hat{z}_t = a_1 z_{t-1} + a_2 z_{t-2} + \cdots + a_p z_{t-p} \ , \tag{1}$$

with $a_i \in \mathbb{R}$ for $i = 1, \ldots, p$ and p the model lag. The prediction for a certain time t is equal to a weighted sum of the previous values up to a certain lag p. At the same time, the nonlinear variant (NAR) is described as:

$$\hat{z}_t = f(z_{t-1}, z_{t-2}, \ldots, z_{t-n}) \ . \tag{2}$$

A depiction of these processes can be found in Figure 1. Remark that in this way, the time-series identification can be written as a classical black-box regression modeling problem:

$$\hat{y}_t = f(x_t) \ , \tag{3}$$

with $y_t = z_t$ and $x_t = [z_{t-1}, z_{t-2}, \ldots, z_{t-p}]$.

2.2 Neural network

The Santa Fe data set is obtained from a chaotic laser which can be described as a nonlinear dynamical system. Given are 1000 training data points. The aim is to predict the next 100 points (it is forbidden to include these points in the training set!). The training data are stored in lasertrain.dat and are shown in Figure 2a. The test data are contained in laserpred.dat and shown in Figure 2b.

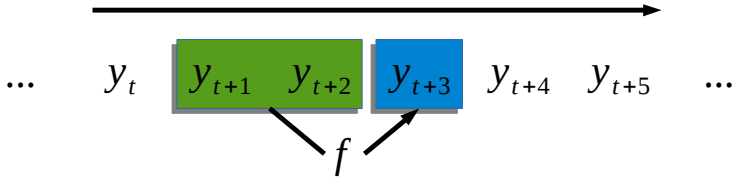
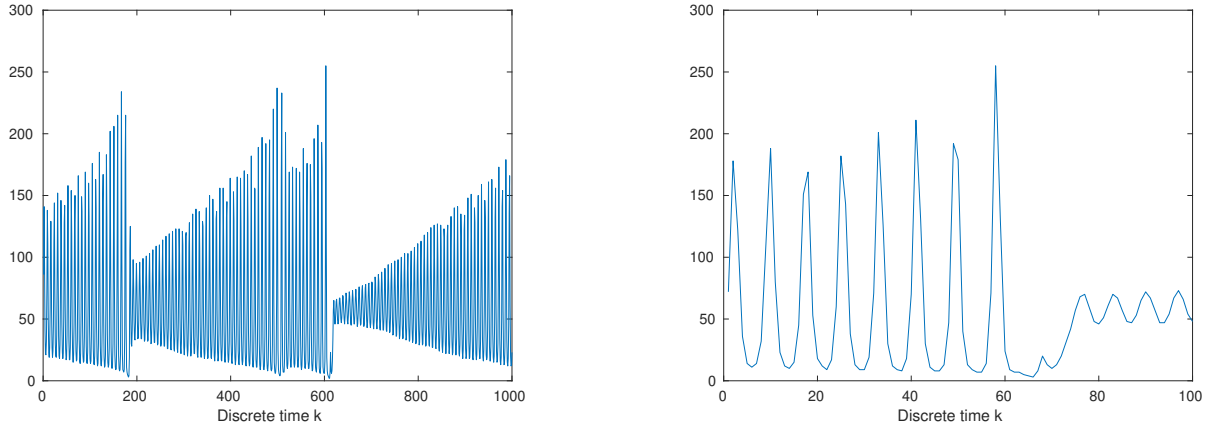


Figure 1: Schematic representation of the nonlinear auto-regressive model.



(a) Training set

(b) Test set

Figure 2

Exercise

Train a MLP with one hidden layer after standardizing the data set. The training is done in feedforward mode:

$$\hat{y}_{k+1} = w^{\mathrm{T}} \tanh(V[y_k; y_{k-1}; ...; y_{k-p}] + \beta). \tag{4}$$

In order to make predictions, the trained network is used in an iterative way as a recurrent network:

$$\hat{y}_{k+1} = w^{\mathrm{T}} \tanh(V[\hat{y}_k; \hat{y}_{k-1}; ...; \hat{y}_{k-p}] + \beta). \tag{5}$$

To format the data you can use the provided function `getTimeSeriesTrainData`. Make sure you understand what the function does by trying it out on a small self-made toy example. To predict the test set you will have to write a `for` loop that includes the predicted value from the previous timestep in the input vector to predict the next timestep. Investigate the model performance with different lag and number of neurons. Which combination of parameters gives the best prediction result on the test set?

2.3 Long short-term memory network

Long Short Term Memory networks, usually just called LSTMs, are a special kind of RNN, capable of learning long-term dependencies [2]. LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, much like data in a computers memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Those gates act on the signals they receive, and similar to the neural networks nodes, they block or pass on information based on its strength and importance, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.

Demo

Study the following example, where an LSTM is build to predict the monthly cases of chickenpox by running `openExample('nnet/TimeSeriesForecastingUsingDeepLearningExample')`.

Exercise

Based on the previous demo, try to model the Santa Fe data set.

- Train the LSTM model and explain the design process. Discuss how the model looks, the parameters that you tune, ... What is the effect of changing the lag value?
- Afterwards try to predict the test set. Use the `predictAndUpdateState` function to predict time steps one at a time and update the network state at each prediction. For each prediction, use the previous prediction as input to the function.
- Compare results of the recurrent neural network with the LSTM. Which model do you prefer and why?

3 Report

Write a report of maximum 3 pages (including text and figures) to discuss the exercises in sections 1 and 2.

References

[1] H. Demuth and M. Beale, Neural Network Toolbox (user's guide), `http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/nnet.shtml`

[2] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

Artificial neural networks - Exercise session 3

Deep feature learning

Bram De Cooman, Hannes De Meulemeester, Joachim Schreurs and David Winant

2019-2020

1 Principal Component Analysis

1.1 Introduction

Principal Component Analysis (PCA) involves projecting onto the eigenvectors of the covariance matrix. The basic idea behind PCA is to map a vector $x = (x_1, x_2, \ldots, x_p)$ of a p dimensional space to a lower-dimensional vector $z = (z_1, z_2, \ldots, z_q)$ in a q dimensional space (where $q < p$). We are going to only consider linear mappings, these are ones where $z_d = e_d^T x$ for some unknown column vectors e_d and T denotes vector (and later on, matrix) transposition. In other words, z can be obtained from x by simple matrix multiplication:

$$z = E^T x \tag{1}$$

where E is a p by q matrix whose column are e_d: $E = [e_1, e_2, \ldots, e_q]$. Our goal is then to reconstruct as well as possible the original vectors by using another matrix F such that

$$\hat{x} = Fz = FE^T x \tag{2}$$

where F is a p by q matrix, and $\hat{x}$ resembles x as good as possible. The idea is that any low dimensional intermediate representation z which allows the original data x to be well reconstructed should have captured a lot of the important structure of the data and so will be interesting or useful to work with.

PCA projects the data onto the subspace spanned by the q eigenvectors corresponding to the q largest eigenvalues of the correlation matrix. More concrete, in order to perform a PCA reduction of a dataset containing N datapoints of dimension p, one can use the following algorithm:

- Zero-mean the data by subtracting the mean of the dataset from each datapoint.
- Calculate the $p \times p$ dimensional covariance matrix of the zero-mean dataset.
- Calculate the eigenvectors and eigenvalues of this covariance matrix.
- Determine the dimension q of the reduced dataset by looking at the largest eigenvalues. The quality of the reduction depends on how close the sum of the largest q eigenvalues is to the sum of all p eigenvalues.
- Create the $q \times p$ projection matrix E^T from the eigenvectors corresponding to the q largest eigenvalues, and reduce the dataset by multiplying it with this matrix.
- To obtain the corresponding p-dimensional datapoints multiply the new data with the transpose of the projection matrix. Notice that this corresponds to choosing F in (2) such that $F = E$. If q is well chosen these regenerated datapoints should be fairly similar to the original datapoints, thus capturing most of the information in the dataset. Remember to add the mean again when comparing with the original data instead of the zero-mean data.

1.2 Exercises

1.2.1 Redundancy and Random Data

The idea of this exercise is to implement the PCA algorithm in Matlab and apply this to different datasets. The above algorithm can be easily programmed in Matlab with the following functions[1]:

- `cov(X)` calculates the average of the dataset X, subtracts it from each data point, and returns the $p \times p$ covariance matrix of the result. It takes as input an $N \times p$ matrix X with N data points and p the dimensionality of each point. Make sure to check the dimensions of your data matrix and transpose when necessary.
- `diag(X)` returns the diagonal of the square matrix X as a column vector.
- `[V,D]=eigs(X,q)` returns the q largest eigenvalues in the matrix (D) and the $p \times q$ matrix (V) with the corresponding eigenvectors of the square matrix X.
- `transpose(X)` returns the transpose of the matrix X. The command X' has the same effect.
- `help` command will show the available documentation on command.

Generate a 50×500 matrix of Gaussian random numbers (`randn(50,500)`) and try to reduce dimensions with PCA (interpret this as 500 datapoints of dimension 50). Examine different reduced datasets for different dimensions. Try to reconstruct the original matrix. Estimate the error, e.g. by calculating the root mean square difference between the reconstructed and the original data (`sqrt(mean(mean((X-Xhat).^2)))`).

Do the same using the data file `choles_all` (standard in Matlab, can be loaded with `load choles_all`). For the exercise, use only the p component, which is a 21×264 matrix (so the dimension of the data equals 21). How does the reduction of random data compare to the reduction of highly correlated data?

1.2.2 Principal Component Analysis on Handwritten Digits

Perform PCA on handwritten images of the digit 3 taken from the US Postal Service database. To access these images, load the Matlab data called `threes.mat` by typing `load threes -ascii`. This loads a 2 megabyte 500×256 matrix called `threes`. Each line of this matrix is a single 16 by 16 image of a handwritten 3 that has been expanded out into a 256 long vector. You can look at the i-th image by typing the command `imagesc(reshape(threes(i,:),16,16),[0,1])`. To have a black-white picture use the command `colormap('gray')` first.

- Compute the mean 3 and display it. Take a look at the command `mean` for this.
- Compute the covariance matrix of the whole dataset of 3s (note that the Matlab function `cov` subtracts the mean automatically, subtracting it beforehand is not incorrect however). Compute the eigenvalues and eigenvectors of this covariance matrix. Plot the eigenvalues (`plot(diag(D)` where D is the diagonal matrix of eigenvalues).
- Compress the dataset by projecting it onto one, two, three, and four principal components. Now reconstruct the images from these compressions and plot some pictures of the four reconstructions.
- Write a function which compresses the entire dataset by projecting it onto q principal components, then reconstructs it and measures the reconstruction error. Note that by choosing how many eigenvectors we use to reconstruct the image we are fixing the number of components, and the quality of the reconstruction. Now call this function for values of q from 1 to 50 (here you probably want to use a loop) and plot the reconstruction error as a function of q.
- What should the reconstruction error be if $q = 256$? What is it if you actually try it? Why?

[1] Matlab also has built-in functions for the PCA algorithm such as `pca(X)` and `processpca(X,maxfrac)` (try for example `maxfrac=0.001`). Be sure to first standardize the data with the function `mapstd(X)`. You can use `processpca('reverse',z,PS)` to get the reconstructed dataset.

- Use the Matlab function `cumsum` to create a vector whose i-th element is the sum of all but the i largest eigenvalues for $i = 1 : 256$. Compare the first 50 elements of this vector to the vector of reconstruction errors calculated previously. What do you notice?

The last question should have shown you a very interesting and important fact: the squared reconstruction error induced by not using a certain principal component is proportional to its eigenvalue. That is why if the eigenvalues fall off quickly then projecting onto the first few components gives very small errors because the sum of the eigenvalues that we are not using is not very large.

2 Stacked Autoencoders

2.1 Introduction

An *autoencoder* neural network is an unsupervised learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. I.e., it uses $y(i) = x(i)$ [1].
Figure 1 shows an example of an autoencoder. The autoencoder tries to learn a function $h_{W,b}(x) \approx x$. In other words, it is trying to learn an approximation to the identity function, so as to output $\hat{x}$ that is similar to x. The identity function seems a particularly trivial function to be trying to learn; but by placing constraints on the network, such as by limiting the number of hidden units (called a *sparse* autoencoder), we can discover interesting structure about the data. In fact, this simple autoencoder often ends up learning a low-dimensional representation similar to PCAs.

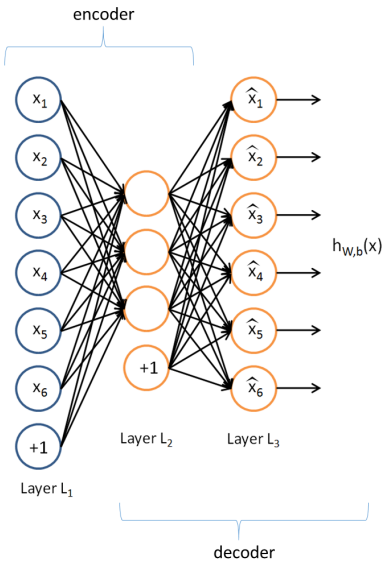


Figure 1: An example of an autoencoder. The part from the input layer to the hidden layer is called the *encoder*. The part from the hidden layer to the output layer is called the *decoder*.

An example of the use of an Autoencoder can be found in [1].

A *stacked autoencoder* is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer are wired to the inputs of the successive layer [2]. The information of interest is contained within the deepest layer of hidden units. This vector gives us a representation of the input in terms of higher-order features. For the use of classification, the common practice is to discard the ”decoding” layers of the stacked autoencoder and link the last hidden layer to a classifier, as depicted in Figure 2.

A good way to obtain good parameters for a stacked autoencoder is to use greedy layer-wise training. So we first train the first autoencoder on the raw input to obtain the weights and biases from the input to the first hidden layer. Then we use this hidden layer as input to train the second autoencoder to obtain the weights and biases from the first to the second hidden layer. Repeat

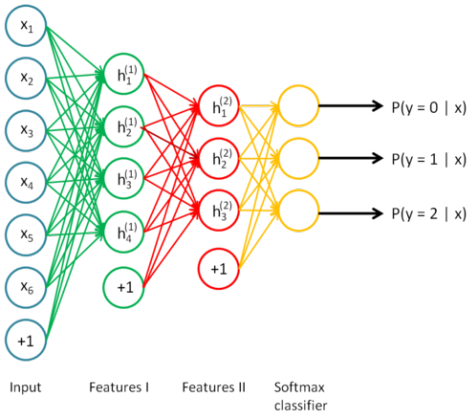


Figure 2: An example of a stacked autoencoder used for classification.

for subsequent layers, using the output of each layer as input for the subsequent layer. This method trains the parameters of each layer individually while freezing parameters for the remainder of the model. To produce better results, after this phase of training is complete, fine-tuning using backpropagation can be used to improve the results by tuning the parameters of all layers at the same time.

A concrete example of a stacked autoencoder can be found in [2].

2.2 Exercise

Run the script `StackedAutoEncodersDigitClassification.m` [3] from Toledo, try to understand what is happening. Use the script `DigitClassification.m` from Toledo to investigate the different parameters (`MaxEpochs`, number of hidden units in each layer, number of layers) and to compare the performance of the Stacked Autoencoder to a normal multilayer neural network. Do not forget to comment the command `rng('default')` when you want to average the results over multiple runs.

Are you able to obtain a better result with different parameters (wrt to the default ones)? How many layers do you need to achieve a better performance than with a normal neural network? What can you tell about the effect of finetuning? Explain *why* and *how* this effects the performance.

3 Convolutional Neural Networks

3.1 Introduction

Convolutional Neural Networks (CNN) is a deep learning technique that uses the concept of *local connectivity*. In a normal multilayer neural network all nodes from subsequent layers are connected, we call these models *fully connected*. The idea is that in a lot of datasets, points that are close to each other, are likely to be a lot more connected than points that are further away. For example, in image datasets where the datapoints represent pixels. Pixels that are close are likely to represent the same part of the image, while pixels that are further away can represent different parts.
CNN's are explained in the Stanford tutorial [4], and also in the following YouTube video trough an example [5].

3.2 Exercise

Run the script `CNNex.m` [6] from Toledo, try to understand what is happening[2].

Take a look at the layers of the downloaded CNN and answer the following questions:

- Take a look at the first convolutional layer (layer 2) and at the dimension of the weights (`size(convnet.Layers(2).Weights)`). If you think back at what you saw in class and/or in [5], what do these weights represent?
- Inspect layers 1 to 5. If you know that a ReLU and a Cross Channel Normalization layer do not affect the dimension of the input, what is the dimension of the input at the start of layer 6 and why?
- What is the dimension of the inputs before the final classification part of the network (i.e. before the fully connected layers)? How does this compare with the initial dimension? Briefly discuss the advantage of CNNs over fully connected networks for image classification.

The script `CNNDigits.m` [7] runs a small CNN on the handwritten digits dataset. Use this script to investigate some CNN architectures. Try out some different amount of layers, combinations of different kinds of layers, dimensions of the weights, etc. Discuss your results. Be aware that some architectures will take a long time to train!

4 Report

Write a report of maximum 3 pages (including text and figures) to discuss the exercises :

- Handwritten Digits PCA and reconstruction
- Digit Classification with Stacked Autoencoders and CNN.
- The answers to the three questions in Section 3.2.

References

[1] UFLDL Tutorial: Autoencoders
http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders

[2] UFLDL Tutorial: Stacked Autoencoders
http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders

[3] MathWorks documentation: Train Stacked Autoencoders for Image Classification
https://nl.mathworks.com/help/nnet/examples/training-a-deep-neural-network-for-digit-classification.html?requestedDomain=www.mathworks.com

[4] UFLDL Tutorial: Convolutional Neural Network
http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork

[5] YouTube: How Convolutional Neural Networks work
https://www.youtube.com/watch?v=FmpDIaiMIeA&list=LLoyD7vXkP56-wnw6rR18S6w&index=1

[6] MathWorks documentation: Image Category Classification Using Deep Learning
https://nl.mathworks.com/help/vision/examples/image-category-classification-using-deep-learning.html

[7] MathWorks documentation: Create Simple Deep Learning Network for Classification
https://nl.mathworks.com/help/nnet/examples/create-simple-deep-learning-network-for-classification.html

[2] Since we don not have access to a CUDA-capable GPU on the computers we can not train a large CNN in class. If you are interested in CNNs and you have access to a GPU for computing you may try the full demo [6] at home (note: this is not necessary for the report).

Artificial neural networks - Exercise session 4

Generative models

Bram De Cooman, Hannes De Meulemeester, Joachim Schreurs and David Winant

2019-2020

1 Restriced Boltzmann Machines

1.1 Introduction

Restricted Boltzmann machines (RBMs) are probabilistic graphical models that can be interpreted as stochastic neural networks [1, 2]. A RBM is a parameterized generative model representing a probability distribution. Given some observations, the training data, learning a RBM means adjusting the RBM parameters such that the probability distribution represented by the RBM fits the training data as well as possible. The standard type of RBM has binary-valued hidden and visible units, and consists of a matrix of weights $W = (w_{i,j}) \in \mathbb{R}^{m \times n}$ associated with the connection between hidden unit h_j and visible unit v_i, as well as bias weights (offsets) a_i for the visible units and b_j for the hidden units. Given these, the energy of a configuration is defined as:

$$E(v, h) = -\sum_{i,j} v_i w_{i,j} h_j - \sum_i a_i v_i - \sum_j b_j h_j.$$

This energy function is analogous to that of a Hopfield network. As in general Boltzmann machines, probability distributions over hidden and/or visible vectors are defined in terms of the energy function. Where the probability function is equal to:

$$P(v, h) = \frac{1}{Z} e^{-E(v,h)}.$$

where Z is a partition function defined as the sum of $e^{-E(v,h)}$ over all possible configurations (in other words, just a normalizing constant to ensure the probability distribution sums to 1). Similarly, the marginal probability of a visible (input) vector of booleans is the sum over all possible hidden layer configurations:

$$P(v) = \frac{1}{Z} \sum_h e^{-E(v,h)}.$$

The visible and hidden neurons can be thought of as being arranged in two layers. The visible units constitute the first layer and correspond to the components of an observation (e.g., one visible unit for each pixel of a digital input image). The hidden units model dependencies between the components of observations (e.g., dependencies between pixels in images). They can be viewed as non-linear feature detectors. A schematic representation is visible on Figure 1.

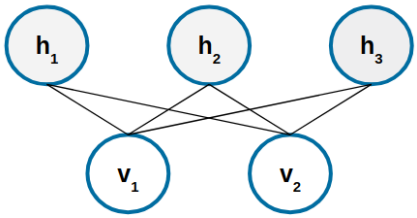


Figure 1: Schematic representation of the Restricted Boltzmann machine.

After successful learning, an RBM provides a closed-form representation of the distribution underlying the observations. It can be used to compare the probabilities of (unseen) observations and to sample from the learned distribution (e.g., to generate new images), in particular from marginal distributions of interest. For example, we can fix some visible units corresponding to a partial observation and sample the remaining visible units for completing the observation (e.g., to solve an image inpainting task). This is done by using the conditional probability. That is, for m visible units and n hidden units, the conditional probability of a configuration of the visible units v, given a configuration of the hidden units h, is:

$$P(v|h) = \prod_{i=1}^{m} P(v_i|h).$$

Conversely, the conditional probability of h given v is given by:

$$P(h|v) = \prod_{j=1}^{n} P(h_j|v).$$

The individual activation probabilities are given by:

$$P(h_j = 1|v) = \sigma\left(b_j + \sum_{i=1}^{m} w_{i,j} v_i\right)$$
$$P(v_i = 1|h) = \sigma\left(a_i + \sum_{j=1}^{n} w_{i,j} h_j\right),$$

where σ denotes the logistic sigmoid. The term **restricted** comes from the fact that there no hidden-to-hidden and visible-to-visible connections. If we also allow these, we get a Boltzmann machine.

1.2 Exercises

The exercises consist of python notebooks that you will run in Google Colab (`https://colab.research.google.com`/). Upload the file `RBM.ipynb` using the `"upload tab"`. Navigate trough the different cells with `Run` icon or pressing `Ctrl + enter`. A more elaborate introduction can be found at `https://colab.research.google.com/notebooks/intro.ipynb`. Afterwards, answer the following questions:

1. What is the effect of different parameters (# epochs and # components) on training the RBM, evaluate the performance visually by reconstructing unseen test images. Change the number of Gibbs sampling steps, can you explain the result?

2. Use the RBM to reconstruct missing parts of images[1]. What is the role of the number of hidden units, learning rate and number of iterations on the performance. What is the effect of removing more rows on the ability of the network to reconstruct? What if you remove rows on different locations (top, middle,...)?

[1] An interesting video about image reconstruction using RBM's can be found at `https://www.youtube.com/watch?v=tk9FTdKOL5Q`

2 Deep Boltzmann Machines

2.1 Introduction

Deep Boltzmann machines can be understood as a series of restricted Boltzmann machines stacked on top of each other [3]. The hidden units are grouped into a hierarchy of layers, such that there is full connectivity between subsequent layers, but no connectivity within layers or between non-neighbouring layers. A schematic representation is visible on Figure 2.

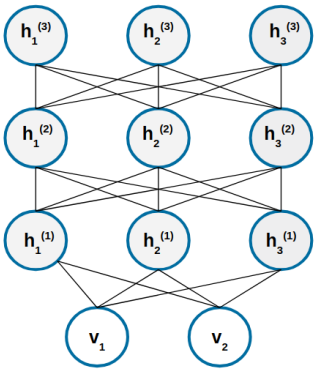


Figure 2: Schematic representation of a 3 layer deep Boltzmann machine.

2.2 Exercises

Upload the file `DBM.ipynb` and go trough the code. Afterwards, answer the following questions:

1. Load the pretrained DBM that is trained on the MNIST database. Show the filters (interconnection weights) extracted from the previously trained RBM (see exercise 1) and the DBM, what is the difference? Can you explain the difference between filters of the first and second layer of the DBM?

2. Sample new images from the DBM. Is the quality better than the RBM from the previous exercise and explain why?

3 Generative Adversarial Networks

3.1 Introduction

Generative adversarial networks (GANs) are a class of algorithms used in unsupervised machine learning, implemented by a system of two neural networks competing with each other in a zero-sum game framework [4]. One neural network, called the generator, generates new data instances, while the other, the discriminator, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data belongs to the actual training dataset or not.

To summarize, here are the steps a GAN takes for an image generation example:

1. The generator takes in random numbers and returns an image.

2. This generated image is fed into the discriminator together with a batch of images taken from the actual dataset.

3. The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

4. Update the weights of the competing neural networks.

3.2 Exercises

Upload the file `DCGAN.ipynb` and go trough the code. Afterwards, answer the following questions:

1. Select one class from the CIFAR dataset and train a Deep convolutional generative adversarial network (DCGAN). Take into account the architecture guidelines from Radford et al. [5]. Make sure that you train the model long enough, such that it is able to generate "real" images. Monitor the loss and accuracy of the generator vs discriminator, and comment on the stability of the training. Explain this in context of the GAN framework.

4 Optimal transport

Optimal transport (OT) [6] theory can be informally described using the words of Gaspard Monge (1746-1818): A worker with a shovel in hand has to move a large pile of sand lying on a construction site. The goal of the worker is to construct with all that sand a target pile with a prescribed shape (for example, that of a giant sand castle). Naturally, the worker wishes to minimize her total effort, quantified for instance as the total distance or time spent carrying shovels of sand. People interested in OT cast that problem as that of comparing two probability distributions-two different piles of sand of the same volume. They consider all of the many possible ways to morph, transport or reshape the first pile into the second, and associate a "global" cost to every such transport, using the "local" consideration of how much it costs to move a grain of sand from one place to another. In OT, one analyzes the properties of that least costly transport, as well as its efficient computation. An example of the computation of OT and displacement interpolation between two 1-D measures is visible on Figure 3.

A common problem that is solved by OT is the assignment problem. Suppose that we have a collection of n factories, and a collection of n stores which use the goods that the factory produce. Suppose that we have a cost function c, so that $c(x, y)$ is the cost of transporting one shipment of the factory from x to y. For simplicity, we ignore the time taken to do the transporting and a factory can only deliver complete goods (no splitting of goods). Let us introduce some notation so we can formally state this as an optimization problem. Let r be the vector containing the amount of goods every store needs. Similarly, k denotes the vector of how much goods every factory produces. Often r and k represent marginal probability distributions, hence their values sum to one. We wish to find the optimal transport plan, whose total cost is equal to:

$$d_M(r, k) = \min_{P \in U(r,k)} \sum_{ij} P_{ij} M_{ij}, \tag{1}$$

where M is the cost matrix, $U(r, k)$ all possible ways to match factories with stores and P_{ij} quantifies the amount of goods that is transported from factory i to store j. This is called the optimal transport between r and k. It can be solved relatively easily using linear programming. The optimum, $d_M(r, k)$, is called the Wasserstein metric. It is a distance between two probability distributions, sometimes also called the earth mover distance as it can be interpreted as how much 'dirt' you have to move to change one 'landscape' (distribution) in another (see Monge's original problem).

Consider a slightly modified form of optimal transport:

$$d_M^\lambda(r, k) = \min_{P \in U(r,k)} \sum_{ij} P_{ij} M_{ij} - \frac{1}{\lambda} h(P), \text{with } h(P) = -\sum_{ij} P_{ij} \log(P_{ij}). \tag{2}$$

Which is called the Sinkhorn distance, where the second term denotes the information entropy of P. One can increase the entropy by making the distribution more homogeneous, i.e. giving everybody a more equal share of goods. The parameter λ determines the trade-off between the two terms: trying to give every store only goods from the closest factory (lowest value in the cost matrix) or encouraging equal distributions. This is similar to regularization in, for example, ridge regression. Similar as that for machine learning problems a tiny bit of shrinkage of the parameter can lead to an improved performance, the Sinkhorn distance is also observed to work better than the Wasserstein distance on some problems. This is because we use a very natural prior on the distribution matrix P: in absence of a cost, everything should be homogeneous.

[1] Above explanation and figures are based on [6].

In many situations the primary interest is not to obtain the optimal transportation map. Instead, we are often interested in using the optimal transportation cost as a statistical divergence between two probability distributions. A statistical divergence is a function that takes two probability distributions as input and outputs a non-negative number that is zero if and only if the two distributions are identical. Statistical divergences such as the KL divergence are frequently used in statistics and machine learning as a way of measuring dissimilarity between two probability distributions. For example, suppose you want to compare different recipes, where every recipe is a set of different ingredients. There is a meaningful distance or similarity between two ingredients, but how do you compare the recipes themselves? Using optimal transport boils down to finding the effort needed to turn one recipe into another.

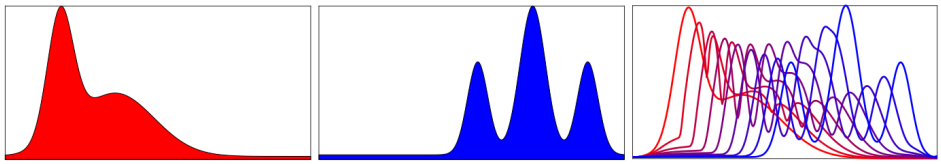


Figure 3: Example of the computation of OT between two 1-D measures. The third figure shows the displacement interpolation between the two using OT.

4.1 Exercises

Upload the file `OT.ipynb` and go trough the code[2]. Afterwards, answer the following question:

1. Upload your own images (of equal size) using the `Files` tab. Afterwards transfer the colors between the two images using the provided notebook. Show the results and explain how the color histograms are transported, how is this different from non optimal color swapping (e.g. just swapping the pixels)?

Upload the file `WGAN.ipynb` and go trough the code. Afterwards, try to answer the following question:

1. Train a fully connected minimax GAN and Wasserstein GAN on the MNIST dataset. Compare the performance of the two GAN's over the different iterations. Do you see an improvement in stability and quality of the generated samples? Elaborate on the knowledge you have gained about optimal transport and the Wasserstein distance.

5 Report

Write a report of maximum 3 pages (including text and figures) to discuss the exercises in sections 1,2,3 and 4.

References

[1] Fischer, A. and Igel, C. (2012, September). An introduction to restricted Boltzmann machines. In Iberoamerican Congress on Pattern Recognition (pp. 14-36). Springer, Berlin, Heidelberg.

[2] Hinton, G. E. (2012). A practical guide to training restricted Boltzmann machines. In Neural networks: Tricks of the trade (pp. 599-619). Springer, Berlin, Heidelberg.

[3] Salakhutdinov, R. and Larochelle, H. (2010, March). Efficient learning of deep Boltzmann machines. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 693-700).

[4] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S. and Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680).

[2]The code is taken from the OT toolbox `https://pot.readthedocs.io/en/stable/`

[5] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[6] Peyré, Gabriel, and Marco Cuturi. "Computational optimal transport." Foundations and Trends in Machine Learning 11.5-6 (2019): 355-607.