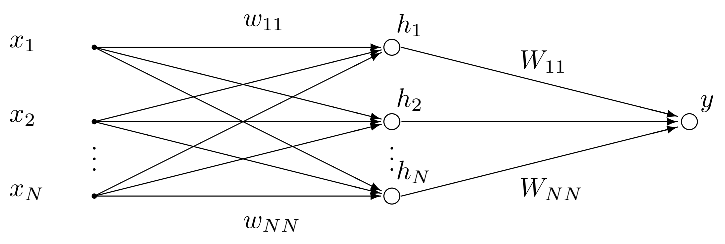
Neural Networks and Learning Systems, TBMI26

TBMI26, Computer Assigments

2018



Neural Networks and Learning Systems
Computer Assignment Collection

Contents

General Information

This compendium contains all computer assignments for TBMI26. You have to pass all assignments to get the lab credits.
The assignments will be graded based on your reports. Templates for the reports are found in the .zip file together will all data needed for the assignments (LISAM). No more than two persons may cooperate on a single report. Reports are sent in to LISAM, don't forget to include both group members, for grading:
lisam.liu.se
The deadlines for the reports can be found in LISAM, if a report is sent in late we will not look at it until the next re-examination period. *So, send in your reports in time*!

Assignments

1. Supervised learning: kNN and Backpropagation

We start of the course by exploring different supervised learning methods. And to get some practical experience with two of the most common methods you will implement the kNN algorithm and backpropagation neural networks.
Your task is to implement the algorithms and evaluate them on different datasets. And, whilst doing so you will learn how the parameters of the methods influences the results.

The Data

All data is divided into three parts the matrix X with all features, the vector L with all class labels and the matrix D with the desired output of the neural networks.
For this assignment you will work with 4 different datasets. The first three consists of 2D data with 2 to 3 classes. These are good for trying out your classifiers while writing the code. Also you can use these to test linear non-linear properties.
The last dataset consists of OCR digits so there is 10 classes with 64 features. See Include images of your best result for each dataset. See http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The Code

We have provided you with some code to help you get started, the code together with the data can be downloaded from the course web page.
Once you have downloaded and unpacked the code you start the assignment by starting Matlab and going to the folder and run:

```
>> run setupSupervisedLab.m
```

By running this script you will setup the paths Matlab need in order to find the other files and functions.
In the folder you will also find folders for the data and some help functions as well as a folder for each classifier you will implement.

kNN

We will start of with the rather straightforward kNN classifier. To help you load your data and evaluate the performance of your implementation you will use *evaluate_kNN.m*. Once you have familiarized yourselves with the evaluation code you can start to implement the algorithm by filling in the *kNN.m* function found in the *kNN* folder.
Furthermore, to evaluate the performance and to identify any weaknesses of our classifiers you will calculate the confusion matrix and the accuracy after each training.
kNN Implementation Tasks:

- Read and understand *evaluate_kNN.m*
- Implement the kNN algorithm in *kNN/kNN.m*
- Implement *calcConfusionMatrix.m*
- Implement *calcAccuracy.m*

kNN Classification Task:
Once you have a stable kNN classifier implementation its time to find the optimal k for each dataset. Do this using cross validation. You have to create your own script for this, but you can borrow most of the code form the previous script.

Backpropagation Networks

Now, lets move on to neural networks. You will implement two types. First a linear classifier based on a single layer which will take an input X and directly calculate an output. Second, a multi-layer network with a single hidden layer.

There are many ways of coding the output of neural networks. In this assignment you will use a voting scheme. The single layer, which is actually the last layer of the multi layer network, will have one output for each class, i.e. one neuron for each class. The output-neuron that gives the strongest, i.e. maximal, output wins and sets the class.

Backprop Implementation Tasks:

The code is implemented in a similar way as kNN evaluation. Start of with the single layer network as this is the easiest to implement. Do not forget to handle the bias weights in a proper way, especially for the two layers of the multi-layer network.

Single Layer Implementation Tasks:

- Read and understand *evaluate_SingleLayer.m*
- We have supplied you with the D matrices, these are used to train a certain behaviour that can be translated to a class. Look at this matrices in combination with the labels in L. Figure out how D enforces a network were the class is given by the neuron with the strongest output.
- Fill in the code for adding the bias.
- Fill in the code that initialize the parameters in W. If you get stuck, think about how many neurons you need for the dataset and how many features there is.
- Implement *runSingleLayer.m*
- Implement *trainSingleLayer.m*

Multi Layer Implementation Tasks:

- Read and understand *evaluate_MultiLayer.m*
- Fill in the code that initialize the parameters in W and V
- Implement *runMultiLayer.m*
- Implement *trainMultiLayer.m*

Backprop Classification Tasks:

Train networks that can consistently produce the test-data accuracies specified below, when using all the data available to you. You are expected to use different parameter settings for each dataset, i.e. different combinations of number of hidden, learning rate and number of iterations. Note that some of the more advanced dataset may require a long training time.

Accuracy limits for dataset Nr.:
1: 98 %
2: 99 %
3: 99 %
4: 96 % (OCR)

Last but not least we want you to use the third dataset and create an example that illustrate a non-generalisable solution by reducing the number of training samples and adjusting the parameters accordingly.

Have fun and do not forget to save images and numbers that you may need for the report.

2. Boosting

In this assignment you will implement the AdaBoost algorithm for face detection in images. The AdaBoost algorithm sequentially trains a number of so-called weak classifiers that are combined into one strong classifier. In a seminal article by Viola and Jones, Robust Real-time Object Detection, it is shown how boosting based on simple Haar-features can create very powerful detection algorithms for computer vision, for example to detect faces in images.

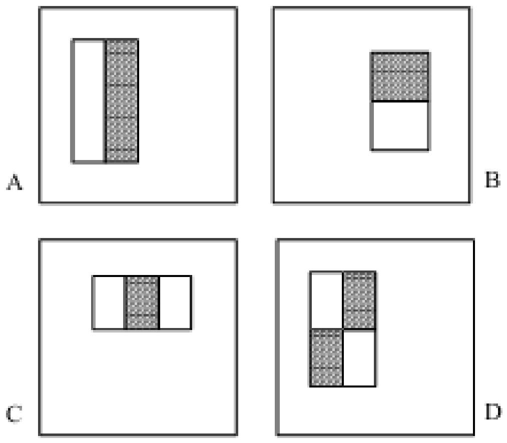


Figure 1:

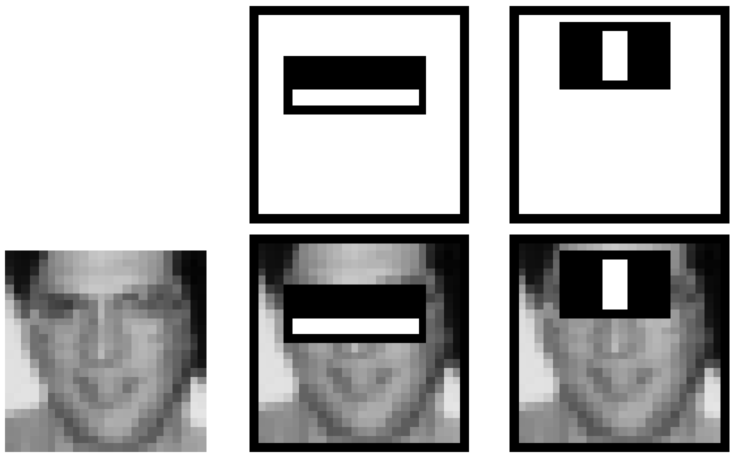


Figure 2: Simple Haar features measuring contrasts at different positions in an image box have proven to be efficient for learning to detect faces and other objects in images. Images taken from Robust Real-time Object Detection by P. Viola and M. Jones

2.1. Your Task

Your task is to implement the standard AdaBoost algorithm to recognize face images. A large number of face images and non-face images are available in the supplementary material. All images are 24x24 pixels large and the non-face images are random image patches from photos found on the internet. Your classifier should be trained to distinguish images of a face from other images using Haar features. To kick-start your work, two ready functions for the feature extraction are available:
GenerateHaarFeatureMasks(nbrHaarFeatures) ExtractHaarFeatures(images,haarFeatureMasks)

These two functions generate the Haar filter masks, as shown in the figure above, and apply them to the images to calculate the features. A set of randomized Haar features will be used in this work. It is recommended to start with a relatively low number of features to keep the computational time low while developing your algorithm. To help you get started we have provided a skeleton for the assignment. Start with the script called AdaBoost and read the comments. You will also have to implement the functions WeakClassifier and WeakClassifierError. From this point your job is to implement the AdaBoost algorithm for the classification of faces vs. non-faces. The weak classifiers in the AdaBoost algorithm should be a thresholding of a well-selected Haar feature xi at a well selected threshold t, i.e., $xi > t$ or $xi < t$. This can be written $pxi > pt$ for polarity $p = 1$ or $p = -1$ respectively.
The implementation should include the following parts:

- Training the AdaBoost algorithm with a smart choice of number of training data and haar-features.
- Vary the number of weak classifier to determine the best one based on the number of training data and number of haar-features used.
- Apply the final strong classifier onto the test data, i.e. the rest of the data not used for training. At least half of the available data must be used for testing the AdaBoost performance. (You should NOT train the system again, only apply the strong classifier when you test the algorithms performance on the test data)
- Once done with the implementation, optimize the parameters such that the accuracy is greater than 0.8 on the test data.

Good luck!

3. Deep Learning

The instructions for the Deep Learning Assignment will be provided separately on LISAM.

4. Reinforcement Learning

This assignment deals with reinforcement learning, i.e. a system that learns from a scalar feedback value. The goal of the system is to maximize the reward over time. The result can be interpreted as a map describing expected feedback (a sum or a weighted sum of future feedback) given that the system is in a certain state and performs a certain action. This may be more similar than anything else in the course to what we usually mean with learning, i.e. learning a behaviour in order to solve a task as good as possible.
The aim of this assignment is to give you a better understanding of how reinforcement learning works in practice and how it can be used to solve easy tasks. The method that you will implement is called Q-learning, a robust method that is also relatively easy to implement.

Good Luck!

4.1. Gridworld – The fastest way

In a world consisting of a grid, the task is to teach a small robot different behaviours using reinforcement learning. The different squares represent states that naturally correspond to the position of the robot. Given a certain state, the robot can choose to move left, right, up or down. The robot's objective is to make its way as efficiently as possible from a arbitrary starting point to a given goal, i.e. find the goal and try to maximize the sum of the feedback under-way. It can choose to explore new parts of the world, or pursue old accustomed patterns.
You can control the robot and draw its position with the help of a set of Matlab functions. Your task is to write a program for Q-learning that given output data from the robot, plans the next action and updates the Q-function.

Example experiment

A rat is dropped into a water tank, while somewhere in the tank there is a small platform hidden precisely under the water surface. Rats are small animals that lose body heat fast since their body surface is relatively large compared to their body volume. Therefore the rats are motivated to find the platform and get out of the cold water. This experiment is subsequently repeated, several times, in order to see if the rat learns to immediately find the platform. The rat is assumed to approximately know its position in the tank by looking at the surrounding environment. The number of successful trials a rat needs before it swims the shortest way to the platform is very low, under 10. Our robot is not as bright, as you will notice.

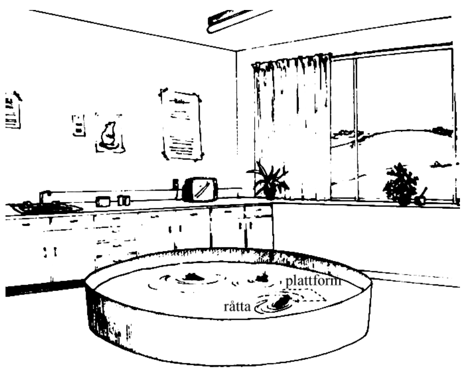


Figure 3: Diagram of the experiment (Adapted from: Fundamentals of Human Neuropsychology by B. Kolb and I. Q. Whishaw)

4.2. Your Task

Write a Matlab program using the provided commands (and the hints below) to implement a Q-learning algorithm that learns the optimal policy for moving the robot to the goal in the different worlds. The following worlds should be explored:

a) ”Irritating blob”, a static world that is good for exploration and to test your algorithm. Consider how different choices of exploration strategy affect the convergence rate of your solution. Also, try different initializations of the Q-function. Is it better, for instance, to use noise instead of a constant when initializing it?

b) ”Suddenly irritating blob”, a variation of the previous world where some randomness has been introduced. How does the robot behave now?

c) ”The road to HG”, experiment giving different values to the parameters.

d) ”The road home from HG”. This is the most important of all the worlds, it illustrates a central property of Q-learning, which? Try playing around with γ and η. You will need a large number (maybe thousands) of iteration to converge.

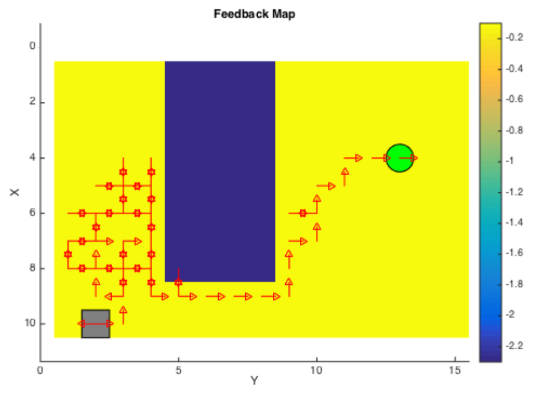


Figure 4: The figure shows a world from the assignment. The robot is symbolized by the grey square, the goal by the green circle. A bright background marks a small negative feedback while a dark background marks a large negative feedback. The arrows show a calculated optimal strategy for how to get to the goal.

As a suggestion you should start with a ϵ-strategy in order to deliberate between performing an optimal action given the current Q-function, or taking a step in another direction in order to explore more of the state space. This means you should with a probability of $1 - \epsilon$ choose an optimal action according to the current Q-function, and with the probability ϵ choose an arbitrary (other) action. Use the function $gwinit(k)$, where k is the number of the world, each time you have reached the goal in order to re-initiate the robot to a new random position. Other available functions are documented in the end of this instruction.

4.3. Appendix A – Hints

- For some worlds, the convergence of the Q-function will take a long time; you may have to wait many minutes before it is finished. If you don't wait the result may be hard to interpret, so don't be afraid to run your algorithm for a long time if you get strange results.

- In order to get max or min along some dimension in a multidimensional array, use the Matlab commands $min(A, [], dim)$ and $max(A, [], dim)$. These commands will work along, and therefore collapse, the dimension dim.
- Remember that the Q-function has to be 0 for all actions at the goal. Remember also that several positions in the table representing the Q-function will not be valid actions, standing at the top of the world and taking a step "up" for instance. It can therefore be appropriate to set the Q-value for these positions/actions to minus infinity (-Inf), so that they don't affect the result of some max-operation.
- In order to plot the Q-function you can use the command $imagesc(Q(:, :, k))$, where k is a specific action. In general, $imagesc$ can be used to plot a 2-D array. You can also plot things with the command $surf$ that works in the same way but will return a 3D-plot instead. Try both.
- A way to represent the Q-function in the initial world is $Q(xposition, yposition, action)$, where Q is a multidimensional array in matlab. You can initialize such an array using commands like $ones$, $zeros$ och $rand$.
- An exploration strategy that simply chooses a random direction for robot motion is not as dumb as it may seem. It could be an easy way to get started before implementing the ϵ-strategy.
- The command $gwplotarrow$ can plot an arrow in your world. Please note that the position has to be a vector of the type $[xpos\ ypos]^T$.

4.4. Appendix B – Matlab code

```
---------------------------------------------------------------
  state = gwaction(action)

  Let the robot perform an action and then return the resulting robot state.

  action           - integer, moves the robot, 1=down, 2=up, 3=right and 4=left

  state.feedback   - robot reinforcement for this action
  state.pos        - robot position after action
  state.isterminal - 1=robot reached goal, 0=goal not reached
  state.isvalid    - 1=action valid, 0=action invalid

  Typical reasons for invalid actions is when the robot bumps into
  a wall. When this happens, simply ignore all state variables and
  perform another (hopefully) valid action.



---------------------------------------------------------------
  gwdraw

  Draw Gridworld and robot.


---------------------------------------------------------------
  gwinit(k)

  Initialize Gridworld 1, 2, 3 or 4 and robot. This should be done after
  the robot finds the goal, to ensure that the robot starts from a
  random location again.

---------------------------------------------------------------
  s = gwstate
  Return s, the state resulting from the last robot action. An example:

s =

         xsize: 10                  - xsize of the grid
         ysize: 15                  - ysize of the grid
           pos: [2x1 double]        - current robot position
    isterminal: 0                   - did the last action reach the goal?
       isvalid: 1                   - was the last action a valid action?
      feedback: -0.1000             - feedback from the last action



---------------------------------------------------------------
  out = sample(values,probs)

  e.g. out = sample([1 2 3 4], [0.2 0.1 0.1 0.6]);
  will give out=1 with probability 0.2, out=2 with prob. 0.1 etc...


---------------------------------------------------------------
  gwplotarrow(position, action)

  Plots an arrow at position given the action. Same encoding of
  actions as in gwaction. Useful for plotting the behavior of an
  optimal policy given a Q-function.
```

4.5. Appendix C – Test the robot

A good way to get started with the assignment is to test by manually driving the robot around.

```
>>
>> gwinit(2)
>> gwdraw
>> gwaction(2)

ans =

         xsize: 10
         ysize: 15
           pos: [2x1 double]
    isterminal: 0
       isvalid: 1
      feedback: -0.1000

>> gwdraw
>> gwaction(2)

ans =

         xsize: 10
         ysize: 15
           pos: [2x1 double]
    isterminal: 0
       isvalid: 1
      feedback: -0.1000
```