Introductory Lab

Machine Learning (1DT071)

18/1-2018

Goal

The purpose of this lab is to introduce some of the concepts and tools that will be used throughout the course, and to give a general idea of what machine learning is. Don't worry for now if some of this fails to make sense—everything you see here will be covered later on in the lectures.

Report

There is no report to write for this exercise. The tasks below include several questions of the sort you will be expected to answer in future labs. For today, you only need to think about possible answers to the questions. Again, don't worry if you aren't able to answer everything.

Task 1: Supervised Learning

We'll start by looking at a type of learning called *supervised learning*. For this type of learning, we have to have a set of input data for which we already know the desired output. In particular, we'll be training a neural network.

A neural network is a network of simple processing elements, called neurons or nodes. Each node computes a single value at a time. Some nodes are dedicated *output nodes*, i.e. their computed values are also the output values of the whole network. Other nodes are *input nodes*, which take their value from part of the input data. In between the input and output nodes are a number of *hidden nodes*, nodes that only deliver value to other nodes.

All nodes listen to the inputs and/or to each other through weighted connections. The network implements a function defined by these weights. Hence, to train a neural network is to find a set of weight values that implement the desired function.

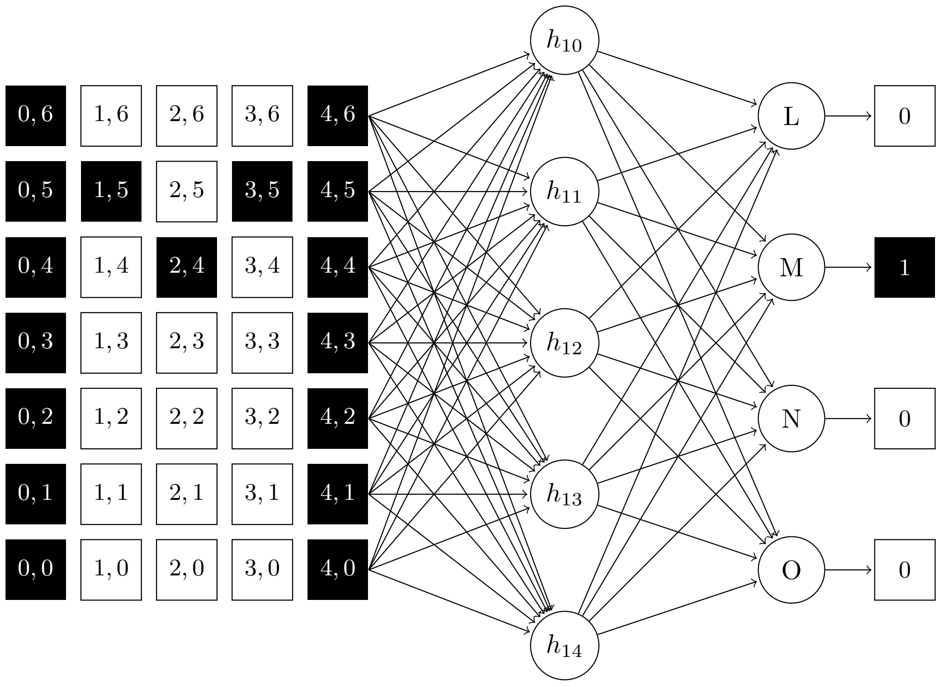


Figure 1: Part of a neural network for character recognition. The (square) input nodes are arranged in a grid to demonstrate their relationship to the bits of the character image. Only some of the weighted connections between nodes are shown; in actuality, each input node would have a connection to each hidden node. In addition, several nodes of the hidden and output layers are omitted for the purposes of space.

In this exercise, we will teach a neural network to classify ("recognize") bit maps of the capital letters A-Z of the English alphabet. A partial diagram of this neural network can be seen in Figure 1. The training data are pairs of input and output vectors. The input vectors are binary bit maps of the letters, and the corresponding outputs are binary vectors of 26 bits, with a 1 in the position representing the desired letter and 0s in all the other positions.

1.1 Training the Neural Network

1. Obtain the file `nncr.zip` from the file area "Introductory lab" in Studentportalen (where you found this document), and unpack it somewhere you can access it.

2. Launch MATLAB by navigating to:

   Applications → Education → MATLAB

   As MATLAB is run on central servers, the first time you start the application, you will get a terminal window asking you to accept to connect to that server. Type "`yes`". Your password might be requested, which is standard procedure for you to connect to the server. Enter your account password and press `Enter`.

3. The default editor mode for the Linux version of MATLAB is set to 'Emacs' by default. Basically, this manifests by having different specific shortcuts for, e.g., cutting and pasting text or saving files. If you are used to Emacs, this is no problem, but if you would prefer keyboard shortcuts common in, e.g., Windows, this may be changed by selecting the **HOME** tab and click **Preferences**. In the left pane, go to:

   MATLAB → **Keyboard** → **Shortcuts**

   . . . and select `Windows Default Set` instead of `Emacs` in the topmost **Active settings** pane. Click **OK**.

4. Change the working directory to the folder you unzipped in step 1. You can browse for this directory using a button on top of the **Current Folder** pane to the left in the MATLAB window.

5. Load the data that defines the problem. You can accomplish this by running the command:

   ```
   nncr_clean
   ```

   The script `nncr_clean` will print all the commands that it runs and prompts for key-presses a few times. You do not have to understand all the commands that it runs at this point. The script will create two matrices:

   `alpha_clean` 26 5×7 bitmaps, one for each letter of the alphabet.

   `targets_clean` a 26×26 identity matrix, representing the goals of the training. Each column of this matrix represents one of the 26 bit output vectors.

   You can view one of the bitmaps by typing:

   ```
   nncr_image(alpha_clean, index)
   ```

   where `index` is the index corresponding to the letter (a value from 1 to 26).

   The script also creates a neural network, `net1`, and assigns random weights to each of the connections in the network.

6. As described in the introduction, this neural network consists of several connected nodes, and those connections are given a weight. So far, the network we've created has randomly assigned weights for each connection. Let's get an idea of how well this randomly-weighted network classifies the input characters:[1]

[1]Adding a semicolon to the end of a MATLAB command will suppress the output of that command, you can freely choose which outputs that you want to see.

```
outputs1 = sim(net1, alpha_clean);
plotconfusion(targets_clean, outputs1);
```

The function `sim` takes a matrix of inputs and a neural network, and returns the outputs that the network would generate for each input. Here, the inputs are our letter images, and the outputs are the classes that the network would assign to each image. The `plotconfusion` command generates a *confusion matrix*, a grid comparing the expected classifications with the classifications generated by the network. You may want to maximize the window in order to make it readable. The rows of this grid represent the classes that can be output by the network, while the columns represent the actual classes of the input data; here, we have 26 classes in each dimension, representing our 26 types of letter images. So, for example, the cell at row 7 and column 10 shows the number of times that the network got an input image corresponding to the letter '`J`', and classified that image as belonging to the letter '`G`'.

7. The randomly weighted network probably didn't perform particularly well. For the network to correctly recognize the letters, we need to *train* it. During training, the network is repeatedly given input data for which the correct output is known. When the network's output differs from the expected output, the weights are adjusted to make it closer. To begin the training, type:

```
net1_trained = train(net1,alpha_clean,targets_clean)
```

This opens up a window displaying the state of the training. The new, trained network will be saved in `net1_trained`, while `net1` will continue to hold the untrained weights.

MATLAB will run through all of the data several times; one run through all 26 data points is called an *epoch*, and you can watch as Matlab counts off the epochs it has run at the top of the **Progress** section of the window.

During training you can watch the error decrease in the **Performance** meter, which displays the initial error at the left hand side, and should steadily move towards 0 on the right hand side. Once training is complete, you can graph the error by pressing the **Performance** button in the **Plots** section of the window.

Observe the change in the error as the training progresses. Your network may have gone through several epochs with only a slight decrease in error, but then after sometime between 10 and 20 epochs it should have decreased to being very close to zero.

8. Let's take another look at the confusion matrix, and see how well the trained network classifies the images:

```
outputs1_trained = sim(net1_trained, alpha_clean);
plotconfusion(targets_clean, outputs1_trained);
```

The new confusion matrix should look quite a bit better, with lots of ones in the green cells which represent correct classifications, and none in the red cells which represent incorrect classifications.

1.2 Generalization

Hopefully you have now seen that the network can learn to recognize all the letters in the alphabet. The next thing to test is whether a network that has been trained on one set of letters can also recognize letters that look slightly different. This ability is called *generalization*, and is a very important property of artificial neural networks.

1. To generate a new set of input data, use the script:

   ```
   nncr_noisy
   ```

   The script will create the following variables:

   `alpha_noisy` A matrix with 286 different letter bitmaps. The first 26 are copies of the clean alphabet, while the rest have randomly generated values added to each bit, making them noisy images, and

   `targets_noisy` The 286 expected outputs for the noisy inputs.

   `alpha_test` Another set of noisy inputs (with different noise), for testing network performance.

   `targets_test` Expected outputs for the test data.

   `net2` An initialized neural network with the same architecture as `net1`.

   Again, you can view one of the "noisy" bitmaps using:

   ```
   nncr_image(alpha_noisy, index)
   ```

   The images at indices 26 and below are copies of the clean images used before; every index over 26 holds a noisy image.

2. Let's see how well our trained neural net recognizes these noisier characters. To do this, we'll plot a confusion matrix based on the test alphabet and targets (`alpha_test` and `targets_test`), but using the **old** network (`net1_trained`) that was trained on clean data:

```
outputs2 = sim(net1_trained, alpha_test);
plotconfusion(targets_test, outputs2);
```

The resulting confusion matrix probably still shows a number of correct classifications, but now there should be many incorrect classifications as well. Pay special attention to the summary columns at the right and bottom edges of the grid. These cells show the percentage of correct and incorrect classifications; those on the right show how many times each class of output was assigned the correct classification, while those at the bottom show how often each class of input was correctly identified. The cell at the bottom right summarizes all the classifications.

Question 1: *What percentage of the noisy letter images did the neural network trained on clean images classify correctly?*

3. Now we'll train another neural network, `net2_trained`, using the noisy data set we just created, and see if it is better able to generalize when given test data it has never seen before.

```
net2_trained = train(net2,alpha_noisy,targets_noisy);
```

   The training this time will take longer.

4. When the training is completed, take a look at a confusion matrix for the test data using the new network:

```
outputs3 = sim(net2_trained, alpha_test);
plotconfusion(targets_test, outputs3);
```

   This is the same test data we tried on the network trained on clean data, above. Our new network, `net2_trained` has been trained on noisy data (`alpha_noisy`), but it is important to remember that it has never seen this test data (`alpha_test`) before.

   Question 2: *How well did the network trained on noisy data classify the test data set?*

5. We'll now test both networks on increasingly noisy sets of bitmaps. Run the command:

```
nncr_test
```

   and Matlab will generate new noisy data sets and test each of the networks on this new data. Note that the networks aren't being retrained here. The weights will remain the same, we're just comparing the actual and expected outputs. When all the tests have been run, you'll be given a plot that shows what percentage of the test data was classified incorrectly by each of the two networks you trained.

Question 3: *Which network performed better with noisy data? Why do you think that might be?*

Question 4: *Now that you've seen a supervised learning example, can you think of another application this kind of learning would be good for?*

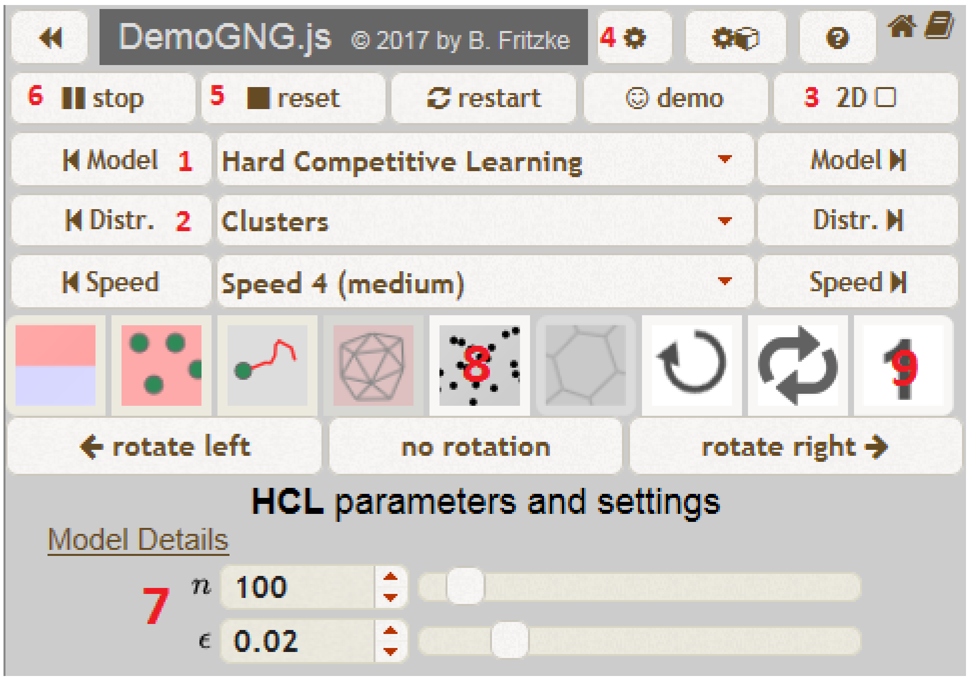
Task 2: Unsupervised Learning

The supervised learning that we saw earlier relied on a set of training data with known correct output. There are problems where the correct output is not known in advance. For example, in a clustering problem there are data points representing different individuals, and the goal is to determine whether these individuals can be divided into an undetermined number of distinct categories. The categories are not known in advance; as a result, there is no target data. *Unsupervised learning* attempts to discover underlying patterns in the input data without relying on known target data.

For this section we'll be using a web applet that demonstrates several unsupervised learning methods. Specifically, these will all be *competitive learning* methods. In competitive learning, there are several nodes with different values. In each iteration, the node that is closest to the data in some way is declared the *winner*; the values of the winner are then modified to move the node closer to the data. In some methods, the values of nodes other than the winner may be updated as well (either the second closest node, or other nodes related to the winning node). In terms of clustering problems, the values of the nodes can be seen as representing a position in the search space; the algorithm then tries to move the nodes to locate all the clusters.

You can find the applet at: `http://www.demogng.de/`.

1. Open the *full-page simulator* at the bottom of the page and then select the *Hard Competitive Learning* algorithm from the **Model** menu (1). Next, find the **Distr.** menu (2) (directly under the Model menu), and select **Clusters**. Switch to 2D (3). You will see a data distribution divided into several clusters (the gray squares represent clusters of data points). The green circles are the current 'location' of the nodes in the search space. Select the **general settings** (4) and uncheck the **initFromPD** box. Click **Reset** (5) and then **Start** (6) to run the algorithm. As the algorithm runs, it will attempt to move the nodes to areas of high density.



2. Click **Stop** (5) so that you can make some changes to the setup. Change the number of nodes **n** to 10 (7) . Click **Reset** to randomly redistribute the ten nodes. Click **Start** to run, and observe how these changes affect the placement of the nodes.

3. To get a better idea of what's going on choose a lower speed. Then enable the **display input signals** (8) and the **single step simulaton** (9) (two of the pictures buttons). This will include two new pieces of information: the "signal"(black dot) is the information the algorithm is responding to (in this case, the position of an individual data point), while the "winner" (red node) of each iteration is the node that was closest to the signal.

   Question 5: *What happens to the node that wins each iteration? What happens to the nodes that do not win?*

4. The **Network Model** menu lets you select other competitive learning algorithms, some of which (such as Self-Organizing Maps and Growing Neural Gas) will be covered in class. Switch to the **Ring** probability distribution, reset and try a few of these other algorithms. Once you have a feel for what each is doing, you can try setting the speed higher in order to see how well the algorithm matches the distribution over time.

   Question 6: *Which algorithm seems the best suited to the ring distribution?*

5. You can also experiment with some parameters at the bottom of the screen, especially:

- The number of nodes.
- **Epsilon** (ϵ) is the "learning rate" of the nodes, which controls how much the winning node's values change in each iteration. What happens when the learning rate is changed?

6. In the probability distribution menu, some distributions are dynamic, i.e. the distribution changes over time (e.g. Square-N). The "DragMe," allows you to change the location of the dense region by dragging it with the right mouse button. Try some of the learning algorithms with one or more of the dynamic distributions.

Question 7: *Which algorithms seem to cope with a changing distribution well? Which algorithms perform poorly?*

Question 8: *Can you think of an application for a clustering algorithm like the one you've seen here?*

Task 3: Reinforcement Learning

The last learning paradigm we'll consider today is *reinforcement learning* (RL). Similarly to unsupervised learning, reinforcement learning does not rely on having a known, correct answer. In RL, the problem is defined as an environment consisting of a set of states. There is also a rule that defines what transitions are possible from one state to the next, and a function that returns an immediate reward based on the selected state. The algorithm starts with no knowledge of the best solution or the environment, and makes a stochastic exploration by making transitions from state to state. When a transition results in a reward, the likelihood of making that same transition in the future is increased. This is called *exploitation*: the algorithm makes use of previous knowledge of rewards when deciding what transition to make next.

Today we'll see RL applied to maze solving, using the GridWorld application. Begin by downloading the `GridWorld.jar` file from the file area in Studentportalen, and launching it. You may have to give permission to run the file as a program (right click ⇒ properties ⇒ permissions ⇒ "Allow executing file as program").

In GridWorld, there is a grid map (5×7 to start with). Each square is a state. One square has a reward, and others are blocked by walls. In each state the agent can make one of four transitions: *up, down, left,* and *right.* If there is no obstacle in the neighboring state indicated by the transition, the agent moves there; otherwise it remains in the same state. The agent receives a reward of 1 for moving to the goal state, and a reward of 0 for moving to any other state.

1. To start the reinforcement learning, click the **Run** button. The agent will start at a random position, and will begin to explore the maze. At first, this exploration will be quite random (you may want to increase the speed to **Medium** using the pull down menu at the top of the window). When the agent reaches the goal, you'll notice that a tiny arrow appears in the grid that led to the goal. Moving to the goal resulted in a reward; the arrow shows an increased tendency for the agent to move in the same direction that resulted in a reward when it is next in that same state.

2. For now, the agent should still be moving randomly over the rest of the maze. Increase the speed to **Fast** and soon you'll see that other squares are getting arrows as well. This is because the RL algorithm being used (Q-Learning) adjusts the probability of making a particular transition based not just on the immediate reward, but also on a discounted sum of expected future rewards. In other words, the algorithm is more likely to select a transition that, in the past, has eventually led to a reward.

3. Let the applet run for 500 episodes. When it stops, change the speed back to **Slow** and let it run again. You'll see that the agent almost always takes the shortest path to the goal now. Sometimes, though, it will still move in a different direction, away from the goal. This is another important feature of RL algorithms, called *exploration*: sometimes the algorithm decides to ignore its knowledge of previous rewards.

Question 9: *Why is exploration so important in RL? Think about what would happen to an agent that always took the best previous result.*

4. You can create or remove walls by right-clicking on a grid. Try extending the wall on one side, so that there is no longer a path on that side of the map. Now click **Run** again without clicking **Init** first.

Question 10: *How well does the algorithm cope with the changed environment?*

5. Note the several settings that can be changed at the top of the screen. Try changing the settings one at a time and rerunning the application. Can you figure out what difference each of the settings makes in the way the robot explores the maze?

Question 11: *Both unsupervised learning and reinforcement learning seek the best solution to a problem on their own, instead of being trained on a set of correct inputs. What's the difference between the type of problem one could use unsupervised learning for, versus a problem that is suitable for reinforcement learning?*

Conclusion

The purpose of this lab has been to give a very quick introduction to several types of machine learning algorithms and applications.There's a good chance that the exercises you've done today raised more questions for you than they answered—these questions will (hopefully!) be addressed in future lectures and labs.