161.326: Statistical Machine Learning
Neural Networks

Stephen Marsland

July 2010

This section is based on chapters 1 to 3 of the course text. These chapters will lead you through the material, and there are some additional readings in the book should you want more detail. It should take 2 weeks.

The code that you will be using is available at `http://seat.massey.ac.nz/personal/s.r.marsland/MLBook.html`. I recommend that you download the zip file of all the code, although you will only need around half of it during the course. Once you have downloaded it, start Eclipse and make a new project (File>New>Pydev project); give it any name you like. Then import the zip file using File>Import and the 'Archive file' option, selecting the zip file that you have downloaded. This should produce a set of directories under your new project, with the directories labelled by the relevant chapter of the book. Have a look in the '2 Linear' directory, and you should see the code that you will be using for the first (very simple) exercises.

1 Week 3: The Perceptron

Begin by looking at the code in `pcn.py`. This is an implementation of the Perceptron algorithm, and the important code matches the lines on the slides. Make sure that you understand what it is doing.

We are now going to reproduce the logic function solutions shown on slides 28 and 29. Have a think about how you can encode the inputs and outputs of the logic gates as 0s and 1s (it is pretty obvious) and then set them up as arrays. Then try and call the `pcn.py` code to actually learn the data. The solution to this is in `logic.py` if you get stuck. When you run that code, you will see that the Perceptron can learn the logical `and` and `or` functions, but not the `xor` function. The reason for this is obvious from the slides: the Perceptron learns a straight line classifier (in 2D; a plane in 3D and a `hyperplane` in higher dimensions). Once you are happy that you know what is going on, extend the code to reproduce slide 31. This is done by extending the XOR problem with an extra dimension that has the (0, 0) point becoming (0, 0, 1) while the others become (1, 0, 0), (0, 1, 0), and (1, 1, 0). You should then find that the Perceptron can learn this function perfectly well (although you might have to increase the number of learning steps that it does).

At the bottom of the code webpage there is a link to the UCI Machine Learning Repository (`http://www.ics.uci.edu/~mlearn/MLRepository.html`). This is a very useful resource for datasets that can be used for testing machine learning algorithms, and one that we will use a few times. For now, go there and download the Pima Indian dataset. The `pima.py` code from the website uses this dataset. You will have to modify the code inside `pima.py` so that the algorithm can find your dataset. For those of you using Windows, you will probably need to use `import os` and then `os.chdir()` as you did with some of the earlier programs you have worked on during the course.

Once you have done that, you should be able to run the code (again, Window's users might have to comment out the `show()` command with a `#` since there isn't actually a graph to show). The output that you get will look something like:

```
Output on original data
[[   8.    2.]
 [ 492.  266.]]
0.356770833333
Output after preprocessing of data
[[ 185.   65.]
```

```
 [  54.   62.]]
0.674863387978
```

although your numbers will be different. If you run it several times you will notice that the numbers change each time. This is the first important thing to remember: neural networks are stochastic, so the results will vary, and hence you need to run the algorithm several times to ensure that the good results are not just a fluke. Then second important thing to notice is that the results are better after preprocessing than before. This is both good and bad news: bad because you need to work to get good results, and good because it means that there are jobs in data analysis!

You might be wondering how to understand the results. The numbers that are shown are a confusion matrix, which is described on page 32 of the book. The leading diagonal consists of the correct answers, and the off-diagonal elements are the wrong answers. The number following it is the fraction of answers that were correct.

2 Week 4: The Multi-Layer Perceptron

The slides work through the MLP algorithm in detail. The main code is in `mlp.py` in the '3 MLP' directory, and the other examples use that. Start by running `logic.py` to see that this algorithm can indeed solve the XOR function. Then have a look at the `iris` data; the original is in the UCI repository, and there is a function to process it into a usable form in the `iris.py` program. You should just be able to run this code to see how the algorithm is used.

We are going to have a go at a more interesting problem this time, which is a task known as *time-series prediction.* We have a set of data that show how something varies over time, and we are going to use a MLP to predict how the data will vary in the future.

2.1 Time-Series Prediction

Time-series prediction is quite a difficult task, but a fairly important one. The task can be written as:

> Given data about how something has behaved over some time period, predict how it will behave in the future.

It is useful in any field where there is data that appears over time, which is to say almost any field. Most notable, if often unsuccessful, uses have been in trying to predict stock markets and disease patterns. The problem is that even if there is some regularity in the time-series, it can appear over many different scales. For example, there is often seasonal variation – if we plotted average temperature over several years, we would notice that it got hotter in the summer and colder in the winter, but we might not notice if there was a overall upward or downward trend to the summer temperatures, because the summer peaks are spread too far apart in the data.

The other problems with the data are practical. How many datapoints should we look at to make the prediction (i.e., how many inputs should there be to the neural network?), and how far apart in time should we space those inputs (i.e., should we use every second datapoint, every 10th, or every one?)? We can write this as an equation, where we are predicting y using a neural network that is written as a function $f(\cdot)$:

$$y = x(t+\tau) = f(x(t), x(t-\tau), \ldots, x(t-k\tau)), \qquad (1)$$

where the two questions about how many datapoints and how far apart they should be come down to choices about τ and k. I'll draw this in the lab, so that you can see it clearly.

The target data for training the neural network is simple, because it comes from further up the time series, and so training is easy – suppose that $\tau = 2$ and $k = 3$. Then the first input data is elements $1, 3, 5$ of the dataset, and the target is element 7. The next input vector is elements $2, 4, 6$, with target 8, and then $3, 5, 7$ with target 9. You train the network by passing through the time-series (remembering to save some data for testing), and then press on into the future making predictions.

2.2 The Data

The dataset that we are going to use is the `PNoz.dat` that is already in the chapter 3 directory. It is the daily measurement of the thickness of the ozone layer above Palmerston North between 1996 and 2004. Ozone thickness is measured in Dobson Units, which are 0.01 mm thickness at 0 degrees Celcius and 1 atmosphere of pressure. I'm sure that I don't need to tell you that the reduction in stratospheric ozone is partly responsible for global warming and the increased incidence of skin cancer, and that in New Zealand we are fairly close to the large hole over Antarctica. What you might not know is that the thickness of the ozone layer varies naturally over the year. This should become obvious when you plot the data. Your task is to predict the ozone levels into the future and see if you can detect an overall drop in the mean ozone level.

There are 4 elements to each vector in the dataset: the year, the day of the year, and the ozone level and sulphur dioxide level, and there are 2855 readings. You should plot the ozone data, so that you can see what it looks like.

2.3 What to Do

- Generate some MLPs with different numbers of input nodes and different numbers of hidden neurons.
- Train the networks on different time-series data, by choosing different values of τ and k
- Use a validation set and early stopping to decide when to stop learning. This is the only thing that involves any programming. First, split the data into training, validation, and testing sets (you can do this any way you want, but I would recommend using the entries with an even index as training data, and half of those with odd indices for training and half for testing. Take a look at section 3.4 of the book for a suggestion of how to do this. Make a new function that calls `mlptrain` and `mlpfwd` in a loop, and keeps track of the errors in the validation set, stopping the loop when the validation error starts to increase.
- Decide whether or not you want to invest in extra suncream on the basis of your results

The approach is described in the book if you get stuck, and a possible solution is in `PNoz.py`.