COMP9444/9844 Assignment 1: Neural networks and handwritten digits

Due 26th August, 18:00.

This assignment is based on a face recognition assignment designed by Tom Mitchell at Carnegie Mellon University.

1. Introduction

This assignment gives you an opportunity to apply neural network learning to the problem of classifying images. You will experiment with a neural network program to recognise a particular digit, and to recognise each digit from a collection of handwritten samples. You will experiment with the effects of changing properties of the network, such as the number of units, different activation functions, and scaling input.

You will not need to do significant amounts of coding for this assignment, and you should not let the size of this document scare you, but training your networks will take time. It is recommended that you read the assignment in its entirety first, and start early.

Download the file `MLP_numbers.tgz` from the class webpage next to where you got this assignment from and uncompress and unpack using `tar xvfz MLP_numbers.tgz` in a directory you want to use for this assignment.

1.1. Submission

The assignment report is to be submitted as a document, and submitted using the "classrun" system. To submit, log in to your account on a CSE machine, and run `9444 classrun -give mlp myfile.pdf` (replace `myfile.pdf` with the name of your document). The same command should be used for people enrolled in the extended course. Further information on using `classrun` is available at:

`http://www.cse.unsw.edu.au/help/doc/primer/node26.html` .

The deadline for submission is August 26 18:00:00. Maximum file size is 3MB however smaller files are preferred. Late penalty is one mark off the maximum mark for each day late, further details are on the class website.

1.2. The image dataset

The images used are from the MNIST set of handwritten digits. This contains 70,000 small images, and has been used in many research projects. You can see what the images look like from the large images in the `images/` directory, which give a representation of each sample[1]. The images loaded by the program are stored in binary format, and are the same as the ones shown in the large images.

Each image in the dataset has a class label representing the digit, and an identifying number, from 0 to approximately 7000 for each class. A second representation is given in the file `imglist.gz`, which includes the label for each image, and can be viewed easily using `zless`. The original dataset contains 60,000 training images and 10,000 test images, the labels in the `imglist.gz` mention whether the image is from the original "train" or "test" set, however for the purposes of this assignment they are joined together into one collection.

1.3. The neural network and image access code

We're supplying C code for a three-layer fully-connected feedforward neural network which uses the backpropagation algorithm to tune its weights. To make life as easy as possible, we're supplying the code for loading the images and the top-level program for training and testing, as a skeleton for you to modify. To help explore what the nets actually learn, you'll also find a utility program for visualizing hidden-unit weights as images.

The code is located in the directory where you unpacked the tar-file. Type `make` to build the package with the supplied code. When the compilation is done, you should have one executable program: `nettrain`. Briefly, `nettrain` takes script files as input, which represent lists of images to be used for the training and test sets, that are used for training and testing the neural network. The program can be used for training and/or recognition, and also is able to save and load the network weights of a trained network.

The code has been compiled and tested successfully on various Linux PCs such as those in the labs. If you wish to use the code on some other platform, feel free, but be aware that the code has only been tested on these platforms.

Details of the routines, explanations of the source files, and related information can be found in Section 3 of this handout.

1.4. Format of the image list scripts

The program takes three files as input that describe the list of images to be used for the training, validation and test sets. These script files describe the specific images to be included in each set, according to the class and id number of each image. This allows the description of the images to be included in each dataset.

Each line of the script file is in the format `class` $\{0-9\}$ `first` $\{0- \sim 7000\}$ `last` $\{0- \sim 7000\}$, for example the line `class 3 first 0 last 299` will add 300 images representing the digit

[1]These images are from `http://www.cs.nyu.edu/~roweis/data.html`

3 to the given dataset. Additional lines allow further images of different classes to be added, the order of the lines is not important.

2. The Assignment

Prepare a short write-up of your answers to the questions in bold face found in the following sequence of initial experiments. When you show modifications of the code, always indicate in what function you make changes and show a bit of context before and after your code and show replaced/modified code where applicable.

1. The code you have been given is currently set up to learn to recognise the digit '0'. The network is set up with 5 hidden units and one output unit. The network is trained to provide a high response for the output unit if the observed image is a '0', and give a low response otherwise. Train the network using the `zeroeight` datasets, which trains the network to identify a '0' digit in a dataset containing '0' and '8' digits. This can be done with the following command:

   ```
   nettrain -n zeroeight.net -t zeroeight_train -1 zeroeight_validate -2 zeroeight_test
   -e 100
   ```

   The arguments are described in more detail in Section 3.1.1, but a short description is in order here. `zeroeight.net` is the name of the network file which will be saved when training is finished. `zeroeight_train`, `zeroeight_validate`, and `zeroeight_test` are script files which specify the training set (2000 examples) and two test sets (1000 and 1000 examples), respectively.

   This command creates a neural net and trains it on a sample of 2000 images, and uses further sets of 1000 images for validation and testing. One way to think of this test strategy is that $\frac{1}{4}$ of the images have been held over for testing. The remaining $\frac{3}{4}$ have been used for a train and cross-validate strategy, in which $\frac{2}{3}$ of these are being used for as a training set and $\frac{1}{3}$ are being used for the validation set to decide when to halt training.

   The output of the program is described in Section 3.1.2. Note that if you run the program multiple times and the network weights have already been saved into the `zeroeight.net` file from a previous run, it will load those weights again. You need to delete or rename the file to start new training, or load the file to continue from where it was last saved. Keep the `zeroeight.net` file produced by one training run as it will be used later.

   How are the values in the 2nd column changing over time, and what does this represent?

   How long does the network take to reach the maximum accuracy level on the training set? At what point does it reach the minimum error level on the validation set, and what is the accuracy on the test set at this point?

2. Create a new dataset that will train the network to learn to identify the digit '0' from amongst a set of digits from '0-9'. You can copy and modify the `zeroeight_*` files, to create new training, validation and test sets, that will include all digits.

   Because we want to train to recognise the digit '0' as opposed to any other digits, we need to create the dataset such that there are an equal number of samples for each class we want to identify. That means that 50% of our samples are for the positive case ('0'), and 50% for

others ('1-9'), otherwise the network will become biased towards a particular class. This can be done using a dataset with 900 samples for '0' and 100 samples for each other digit, and a similar proportion for the validation and test sets, such as using half of each amount. Make sure no images are re-used between the different sets.

Show the datasets you have constructed. Train your network using this new dataset for 100 epochs, and place the output into a new network file (eg `-n zeroall.net`).

Find the point where the error on the validation set is minimum. What is the classification accuracy on the test set at this point? What happens to the error and accuracy on each set as training continues after this point, and why is this so?

What do the values in the 4th, 6th and 8th columns represent, and how are they changing over time? what does this mean?

When you are looking for the point where the error level is at a minimum, try and look for a minimum over an average of several epochs (time steps) rather than a single lowest point.

3. Now, try taking a look at how backpropagation tuned the weights of the hidden units with respect to each pixel. First type `make hidtopgm` to compile the utility on your system. Then, to visualize the weights of hidden unit n, type the following (using the name of your network, image name and hidden unit number):

   ```
   hidtopgm network-name.net image-filename.pgm 28 28 n
   ```

   This will produce an image, which you can view using a program such as `eog` or `xv`, which should be available on lab machines. This will display the range of weights, with the lowest weights mapped to pixel values of zero, and the highest mapped to 255.

   Using the network produced for the zero-vs-eight classifier and the zero-vs-all classifier, have a look at the weights of some of the hidden units for each. Can you describe properties of the features constructed?

4. Next you will modify the code to adjust the number of weights used by the network, and the training class. Change it so that it recognises the number '8', and uses one hidden unit.

   What code did you modify?

5. Using this code, we will examine the result of training the system using different size training sets and different numbers of hidden units. A number of training and test sets have been provided that we will use. `eightall_sm_train, eightall_med_train` and `eightall_lg_train` are datasets to train to recognise '8' amongst all digits, using a small, medium and large training set. `eightall_validate` and `eightall_test` are test sets that we will use for each of the tests.

   Conduct a number of runs using these datasets, to compare results using different size datasets, using the network with 1 hidden unit to identify the digit '8'.

   For each run, find the maximum accuracy level reached on the training set, and the find the test accuracy for this run. The test accuracy used should be the value at the point where the validation set error is minimum, and may be at a different time than the maximum accuracy level reached on the training set.

   How do these values change as different size training datasets are used? What is your explanation?

Be careful not to re-use the same network file between runs.

6. Change the network to use 5 and 50 hidden units, and perform the same tests as the previous step. This should give you 3x3 results, using a small, medium and large dataset for 1, 5 and 50 hidden units respectively. All of these runs should use the same validation and test sets.

   Discuss how the different size network and different size training sets affect training times and the overall accuracy on the test set.

7. Lets take a closer look at which images the net may have failed to classify (use the appropriate .net file for your trained network, you can use the network from any of the previous runs):

   ```
   nettrain -n eightall_sm_50.net -T -1 eightall_validate -2 eightall_test
   ```

   This will show the output of the network when presented with each test image, which can be used to identify misclassified images. You can see what each image looks like according to its label (eg `train3_5143`). Copy the label and run `zless imglist.gz`, if you press '/' and type/paste the image label you can find what the image looks like. Use '?' to search backwards.

   Can you describe notable aspects or commonalities between the misclassified images?

8. Now, change the network to learn to classify each digit, rather than identifying a single digit. The network will be presented with an image, and will need to identify which digit it is. To do this, you will need to implement a different output encoding (since you must now be able to distinguish among 10 digits). Use 10 output units for this and interpret the outputs such that the highest output value of a unit will be taken as indicating the respective class.

   Describe what code you modified.

9. A number of training and test sets have been provided to train and test the network to classify all digits. This uses the `all_*_train`, `all_validate` and `all_test` sets. Using each training set (small, med and large), conduct tests using a network of 1, 5 and 50 hidden units, as previously. This should give 3x3 sets of results (9 total). You may need to change the number of epochs being run.

   Conduct these runs and describe how the training time and test accuracy changes with different size networks, and with different size training sets. Discuss why you think these effects occur.

   For the smallest network (1 hidden unit), compare what happens to the training accuracy and test accuracy as the size of the training set is increased. Why does this happen?

10. The program `outtopgm` provides output of the weights between the hidden units and the output nodes. You can use this to identify which features are providing the most influence on the responses for each output node. For example the following will show the weights for output unit 3 in a network with 10 output nodes (note that 10 + 1 is used for the third parameter, as described in Section 3.6).

    ```
    outtopgm network-name.net image-filename.pgm 11 1 3
    ```

    The text output from this program will likely be more useful than the image.

    Find some of the features (hidden units) that are playing a significant role in the output of a particular class, either positively or negatively. You can use the

network of any of the previous runs. Display the features using `hidtopgm`. Are there recognisable aspects of the features?

11. These runs have been conducted using the logistic activation function, $\varphi(x) = \frac{1}{1+e^{-x}}$. Change the code so that the program uses the activation function $\varphi(x) = tanh(x)$. This should not require many changes, however requires an understanding of the way the activation function is used, and other parts of the code that need to be changed. Note that the derivative of the logistic function is

$$\varphi'(x) = \varphi(x)(1 - \varphi(x))$$

and the derivative of the *tanh* function is

$$\varphi'(x) = 1 - \varphi(x)^2$$

Implement this change and describe the various parts of the program that you changed. Conduct at least one new run using the modified code. Try to verify that your program is running correctly, and describe any changes to the training time, behaviour or accuracy of the program.

12. (optional) If you like you can try running the system using the full training set of 50,000 images (`full_train`, and 10,000 for validation in `all_validate`), to see the performance against the standard test dataset (`all_test`). This may take some time, depending on the computer you are running it on. There are examples of results from other systems available at `http://yann.lecun.com/exdb/mnist` (this will not be assessed).

3. Documentation

The code for this assignment is broken into several modules:

- `readimage.c .h`, `readMNIST.c .h`: these provide a number of methods for loading and handling images. This supports read/write of PGM image files and pixel access/assignment. The `readMNIST` methods provide a means of reading images from the binary dataset file. **You will not need to modify any code in this module to complete the assignment.**
- `backprop.c`, `backprop.h`: the neural network package. Supports three-layer fully-connected feedforward networks, using the backpropagation algorithm for weight tuning. Provides high level routines for creating, training, and using networks. **You will only need to make changes to this code to alter the activation function and any necessary related changes, for the last step of the assignment.**
- `nettrain.c`: the top-level program which uses the previous modules to implement an image recogniser. You will need to modify this code to change network sizes and learning parameters, both of which are trivial changes. The performance evaluation routines `performance_on_imagelist()` and `evaluate_performance()` are also in this module, you will need to modify these for the all-digit recogniser.

  This also includes interface routines for loading images into the input units of a network, and setting up target vectors for training. You will need to modify the routine `load_target`, when implementing the all-digit recogniser, to set up appropriate target vectors for the output encodings you choose.

- `hidtopgm.c`: the hidden unit weight visualization utility. It's not necessary modify anything here, although it may be interesting to explore some of the numerous possible alternate visualization schemes.

Feel free to modify anything you want in any of the files if it makes your life easier or if it allows you to do a nifty experiment.

3.1. `nettrain`

3.1.1. Running `nettrain`

`nettrain` has several options which can be specified on the command line. This section briefly describes how each option works. A very short summary of this information can be obtained by running `netfacetrain` with no arguments.

`-n <network file>` - this option either loads an existing network file, or creates a new one with the given name. At the end of training, the neural network will be saved to this file.

`-e <number of epochs>` - this option specifies the number of training epochs which will be run. If this option is not specified, the default is 100.

`-T` - for test-only mode (no training). Performance will be reported on each of the three datasets specified, and those images misclassified will be listed, along with the corresponding output unit levels.

`-s <seed>` - an integer which will be used as the seed for the random number generator. A constant default seed is given. This allows you to reproduce experiments if necessary, by generating the same sequence of random numbers. It also allows you to try a different set of random numbers by changing the seed.

`-S <number of epochs between saves>` - this option specifies the number of epochs between saves. The default is 100, which means that if you train for 100 epochs (also the default), the network is only saved when training is completed.

`-t <training image list>` - this option specifies a script file describing the images from the MNIST dataset that are to be used in the training set. The format is described in Section 1.4. If this option is not specified, it is assumed that no training will take place ($epochs = 0$), and the network will simply be run on the test sets. In this case, the statistics for the training set will all be zeros.

`-1 <test set 1 list>` - this option specifies a script file describing the first (or validation) test set. If this option is not specified, the statistics for test set 1 will all be zeros.

`-2 <test set 2 list>` - same as above, but for test set 2. The idea behind having two test sets is that one can be used as part of the train/test paradigm, in which training is stopped when performance on the test set begins to degrade. The other can then be used as a "real" test of the resulting network.

3.1.2. Interpreting the output of `nettrain`

When you run `nettrain`, it will first read in all the data files and print a bunch of lines regarding these operations. Once all the data is loaded, it will begin training. At this point, the network's training and test set performance is outlined in one line per epoch. For each epoch, the following performance measures are output:

```
<epoch> <delta> <trainperf> <trainerr> <t1perf> <t1err> <t2perf> <t2err>
```

These values have the following meanings:

`epoch` is the number of the epoch just completed; it follows that a value of 0 means that no training has yet been performed.

`delta` is the sum of all δ values on the hidden and output units as computed during backprop, over all training examples for that epoch.

`trainperf` is the percentage of examples in the training set which were correctly classified.

`trainerr` is the average, over all training examples, of the error function $\frac{1}{2}\sum(t_i - o_i)^2$, where t_i is the target value for output unit i and o_i is the actual output value for that unit.

`t1perf` is the percentage of examples in test set 1 which were correctly classified.

`t1err` is the average, over all examples in test set 1, of the error function described above.

`t2perf` is the percentage of examples in test set 2 which were correctly classified.

`t2err` is the average, over all examples in test set 2, of the error function described above.

3.2. Tips

Although you do not have to modify many parts of the code, you will need to know a little bit about the routines and data structures being used, so that you can easily implement new output encodings for your networks. The following sections describe each of the packages in a little more detail. You can look at the code to see how the routines are actually used.

In fact, it is probably a good idea to look over `nettrain.c` first, to see how the training process works. You will notice that `load_target()` is called to set up the target vector for training. You will also notice the routines which evaluate performance and compute error statistics, `performance_on_imagelist()` and `evaluate_performance()`. The first routine iterates through a set of images, computing the average error on these images, and the second routine computes the error and accuracy on a single image.

You will almost certainly not need to use all of the information in the following sections, so don't feel like you need to know everything the packages do. You should view these sections as reference guides for the packages, should you need information on data structures and routines.

Another fun thing to do, if you didn't already try it in the last question of the assignment, is to use the image package to view the weights on connections in graphical form; you will find routines for creating and writing images, if you want to play around with visualizing your network weights.

Finally, the point of this assignment is for you to obtain first-hand experience in working with neural networks; it is **not** intended as an exercise in C hacking. An effort has been made to keep the image package and neural network package as simple as possible. If you need clarifications about how the routines work, don't hesitate to ask.

3.3. The neural network package

As mentioned earlier, this package implements three-layer fully-connected feedforward neural networks, using a backpropagation weight tuning method. We begin with a brief description of the data structure, a `BPNN` (BackPropNeuralNet).

All unit values and weight values are stored as `doubles` in a `BPNN`.

Given a `BPNN *net`, you can get the number of input, hidden, and output units with `net->input_n`, `net->hidden_n`, and `net->output_n`, respectively.

Units are all indexed from 1 to n, where n is the number of units in the layer. To get the value of the `k`th unit in the input, hidden, or output layer, use `net->input_units[k]`, `net->hidden_units[k]`, or `net->output_units[k]`, respectively.

The target vector is assumed to have the same number of values as the number of units in the output layer, and it can be accessed via `net->target`. The `k`th target value can be accessed by `net->target[k]`.

To get the value of the weight connecting the `i`th input unit to the `j`th hidden unit, use `net->input_weights[i][j]`. To get the value of the weight connecting the `j`th hidden unit to the `k`th output unit, use `net->hidden_weights[j][k]`.

The routines are as follows:

```
void bpnn_initialize(seed)
    int seed;
```

This routine initializes the neural network package. It should be called before any other routines in the package are used. Currently, its sole purpose in life is to initialize the random number generator with the input `seed`.

```
BPNN *bpnn_create(n_in, n_hidden, n_out)
    int n_in, n_hidden, n_out;
```

Creates a new network with `n_in` input units, `n_hidden` hidden units, and `n_output` output units. All weights in the network are initialised to zero to make the output display more readable, however this can be changed to be randomly initialised to values in the range $[-1.0, 1.0]$. Returns a pointer to the network structure. Returns `NULL` if the routine fails.

```
void bpnn_free(net)
    BPNN *net;
```

Takes a pointer to a network, and frees all memory associated with the network.

```
void bpnn_train(net, learning_rate, momentum, erro, errh)
    BPNN *net;
```

```
    double learning_rate, momentum;
    double *erro, *errh;
```

Given a pointer to a network, runs one pass of the backpropagation algorithm. Assumes that the input units and target layer have been properly set up. `learning_rate` and `momentum` are assumed to be values between 0.0 and 1.0. `erro` and `errh` are pointers to doubles, which are set to the sum of the δ error values on the output units and hidden units, respectively.

```
void bpnn_feedforward(net)
    BPNN *net;
```

Given a pointer to a network, runs the network on its current input values.

```
BPNN *bpnn_read(filename)
    char *filename;
```

Given a filename, allocates space for a network, initializes it with the weights stored in the network file, and returns a pointer to this new `BPNN`. Returns `NULL` on failure.

```
void bpnn_save(net, filename)
    BPNN *net;
    char *filename;
```

Given a pointer to a network and a filename, saves the network to that file.

3.4. The image package

The image package provides a set of routines for manipulating images. An image is a rectangular grid of pixels; each pixel has an integer value ranging from 0 to 255. Images are indexed by rows and columns; row 0 is the top row of the image, column 0 is the left column of the image.

```
IMAGE *img_creat(filename, nrows, ncols)
    char *filename;
    int nrows, ncols;
```

Creates an image in memory, with the given filename, of dimensions `nrows` $\times$ `ncols`, and returns a pointer to this image. All pixels are initialized to 0. Returns `NULL` on failure.

```
int ROWS(img)
    IMAGE *img;
```

Given a pointer to an image, returns the number of rows the image has.

```
int COLS(img)
    IMAGE *img;
```

Given a pointer to an image, returns the number of columns the image has.

```
char *NAME(img)
    IMAGE *img;
```

Given a pointer to an image, returns a pointer to its base filename (i.e., if the full filename is `/usr/joe/stuff/foo.pgm`, a pointer to the string `foo.pgm` will be returned).

```
int img_getpixel(img, row, col)
    IMAGE *img;
    int row, col;
```

Given a pointer to an image and row/column coordinates, this routine returns the value of the pixel at those coordinates in the image.

```
void img_setpixel(img, row, col, value)
    IMAGE *img;
    int row, col, value;
```

Given a pointer to an image and row/column coordinates, and an integer `value` assumed to be in the range $[0, 255]$, this routine sets the pixel at those coordinates in the image to the given value.

```
int img_write(img, filename)
    IMAGE *img;
    char *filename;
```

Given a pointer to an image and a filename, writes the image to disk with the given filename. Returns 1 on success, 0 on failure.

```
void img_free(img)
    IMAGE *img;
```

Given a pointer to an image, deallocates all of its associated memory.

```
IMAGELIST *imgl_alloc()
```

Returns a pointer to a new `IMAGELIST` structure, which is really just an array of pointers to images. Given an `IMAGELIST *il`, `il->n` is the number of images in the list. `il->list[k]` is the pointer to the `k`th image in the list.

```
void imgl_add(il, img)
    IMAGELIST *il;
    IMAGE *img;
```

Given a pointer to an imagelist and a pointer to an image, adds the image at the end of the imagelist.

```
void imgl_free(il)
    IMAGELIST *il;
```

Given a pointer to an imagelist, frees it. Note that this does not free any images to which the list points.

```
void imgl_load_images_from_textfile(il, filename)
    IMAGELIST *il;
    char *filename;
```

Takes a pointer to an imagelist and a filename. `filename` is assumed to specify a file which is a list of pathnames of images, one to a line. Each image file in this list is loaded into memory and added to the imagelist `il`.

3.5. `hidtopgm`

`hidtopgm` takes the following fixed set of arguments:

`hidtopgm` *net-file image-file x y n*

net-file is the file containing the network in which the hidden unit weights are to be found.

image-file is the file to which the derived image will be output.

x and y are the dimensions in pixels of the image on which the network was trained.

n is the number of the target hidden unit. n may range from 1 to the total number of hidden units in the network.

3.6. `outtopgm`

`outtopgm` takes the following fixed set of arguments:

`outtopgm` *net-file image-file x y n*

This is the same as hidtopgm, for output units instead of input units. Be sure you specify x to be 1 plus the number of hidden units, so that you get to see the weight w_0 as well as weights associated with the hidden units. For example, to see the weights for output number 2 of a network containing 3 hidden units, do this:

outtopgm pose.net pose-out2.pgm 4 1 2

net-file is the file containing the network in which the hidden unit weights are to be found.

image-file is the file to which the derived image will be output.

x and y are the dimensions of the hidden units, where x is always 1 + the number of hidden units specified for the network, and y is always 1.

n is the number of the target output unit. n may range from 1 to the total number of output units for the network.