A Neural Networks Mini-project: Image Processing

Note: This document is adapted from the Homework 3 for the 10-701/15-781 Machine Learning course which was held in 2003 at CMU during the fall semester.

This assignment gives you an opportunity to formulate and solve a machine learning problem using a real-world data set.

What you will do.

- Find a partner to work on this project (two person teams are encouraged, though you may work alone if you prefer. No three person teams please.)
- The image data and a neural network package is available at http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/mlc/hw3/face/.

  This directory provides an implementation of the Backpropagation algorithm described in Chapter 4 of the Tom Mitchell's "Machine Learning" textbook. It also provides an image dataset discussed in Section 4.7 of the book, containing over 600 face images of students from CMU Machine Learning classes. This code and data are only supported under the Unix and Linux operating systems.

  There are four files with extension `tar.gz` under this directory. Copy each to your home directory and unpack with the following command:

  ```
  tar -xvf XXX.tar.gz
  ```

  Now you should find four subdirectories under your home directory: `<./faces>`, `<./faces_4>`, `<./code>` and `<./example>`.
- Define your learning task. The image data contains approximately 30 different images from each of twenty people, with each image labeled by several properties including the person ID, whether they are wearing sunglasses, etc. Therefore, classifying images to determine whether the person is wearing sunglasses is one possible choice.
- Perform the work. As a guideline, we expect each student to spend 7–12 hours on this homework over the course of two weeks (remember you are working in pairs, so you can do a fairly substantial project).
- Turn in a 2–4 page write-up. Your write-up should describe *precisely* your learning task(s), your learning method(s) including how you represented the data for input to the learner, your experiments, results, and any conclusions your draw from this. The clarity and content of your write-up will have a primary impact on your grade. The reports must not be more than 4 pages, 11-point font, including figures. Each two-person team must hand in a single write-up.

Grading and determining when you have done enough.

A good strategy for this homework is to divide your effort into two one-week blocks. During week one you might complete training a classifier for image classification using the provided code, and experimenting with some with the parameters to determine what works best. Then during the second week you could make up your own follow-on question to study (perhaps trying a competing method, or studying some issue you noticed during the first week). A project that does a solid job applying the given code and carefully evaluating and describing it might get 75–80% credit. A project that in addition pursues an interesting second approach or alternative problem might get 90–100% credit.

Be creative! Exploring your own interesting ideas and comparing them with the baseline approaches will receive credit whether they beat the baseline or not.

1 Face Images

The `<./faces>` directory contains 20 subdirectories, one for each person, named by userid. Each of these directories contains several different face images of the same person. The image is stored in PGM format with the following naming convention:

`<userid>_<pose>_<expression>_<eyes>_<scale>.pgm`

- `<userid>` is the user id of the person in the image, and this field has 20 values: an2i, at33, boland, bpm, ch4f, cheyer, choon, danieln, glickman, karyadi, kawamura, kk49, megak, mitchell, night, phoebe, saavik, steffi, sz24, and tammo.
- `<pose>` is the head position of the person, and this field has 4 values: straight, left, right, up.
- `<expression>` is the facial expression of the person, and this field has 4 values: neutral, happy, sad, angry.
- `<eyes>` is the eye state of the person, and this field has 2 values: open, sunglasses.
- `<scale>` is the scale of the image, and this field has 3 values: 1, 2, and 4. 1 indicates a full-resolution image (128 columns $\times$ 120 rows); 2 indicates a half-resolution image (64×60); and 4 indicates a quarter-resolution image (32×30).

If you've been looking closely in the image directories, you may notice that some images have a `.bad` suffix rather than the `.pgm` suffix. As it turns out, 16 of the 640 images taken have glitches due to problems with the camera setup; these are the `.bad` images. Some people had more glitches than others, but everyone who got "faced" should have at least 28 good face images (out of the 32 variations possible, discounting scale).

`<./faces_4>` is a subset of the face data, which contains only the one-quarter sized images. You may consider using these images in order to keep training time to a manageable level.

2 Neural Network Code

The `<./code>` directory contains C code for a three-layer fully-connected feedforward neural network which uses the backpropagation algorithm to tune its weights. To compile the code, type the following commands

```
cd ./code
make
```

When the compilation is done, you should have one executable program `facetrain` under the source code directory. Copy it to the directory where training/test is carried out, say your home directory.

```
cp facetrain ../
```

Briefly, `facetrain` takes lists of image files as input, and uses these as training and test sets for a neural network. `facetrain` can be used for training and/or recognition, and it also has the capability to save networks to files. The code you have been given is initially set up to learn to recognize the person with userid `glickman`. You can modify code in `imagenet.c` and `facetrain.c` to implement various recognition tasks. Refer to the beginning of Section 4 for an overview of how to make changes to this code.

To help explore what the nets actually learn, you'll also find a utility program `hidtopgm` for visualizing hidden-unit weights as images. The utility can be compiled on your system by issuing the command

```
make hidtopgm
```

There are two other compiling options. The `facetrain_init0` program (compile with '`make facetrain_init0`') initializes the hidden unit weights of a new network to zero, rather than random values. The `outtopgm` (compile with '`make outtopgm`') is the same as `hidtopgm`, for output units instead of input units.

Details of the routines, explanations of the source files, and related information can be found in Section 4 of this handout.

3 Example

The `<./example>` directory is a simple example of using the provided program to recognize the person with userid `glickman`. You will find two executable files `rainbow` and `hidtopgm` under this directory which is precompiled on the Andrew-side Unix machine. Replace it with the program you compiled from the source code if necessary.

To get started, switch to the `<./example>` directory and type the following command on a single line:

```
./facetrain -n face.net -t straightrnd_train.list -1 straightrnd_test1.list
                                    -2 straightrnd_test2.list -e 75
```

The command trains a network using the default learning parameter settings (learning rate 0.3, momentum 0.3) for 75 epochs. `facetrain`'s arguments are described in Section 4.1, but a short description is in order here. `face.net` is the name of the network file which will be saved when training is finished. `straightrnd_train.list`, `straightrnd_test1.list`, and `straightrnd_test2.list` are text files which specify the training set (70 examples)

and two test sets (34 and 52 examples), respectively.

This command creates and trains your net on a randomly chosen sample of 70 of the 156 "straight" images, and tests it on the remaining 34 and 52 randomly chosen images, respectively. One way to think of this test strategy is that roughly $\frac{1}{3}$ of the images (`straightrnd_test2.list`) have been held over for testing. The remaining $\frac{2}{3}$ 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 (`straightrnd_train.list`) and $\frac{1}{3}$ are being used for the validation set to decide when to halt training (`straightrnd_test1.list`).

The difference between the `straightrnd_*.list` and the `straighteven_*.list` sets is that while the former divides the images purely randomly among the training and test sets, the latter ensures a relatively even distribution of each individual's images over the sets. Because we have only 7 or 8 "straight" images per individual, failure to distribute them evenly would result in testing our network the most on those faces on which it was trained the least. If you have access to a fast machine, then you are welcome to do these experiments on the entire set (replace `straight` with `all` in the command above). Otherwise, stick to the "straight" images.

To see which images the net may have failed to classify, type

```
./facetrain -n face.net -T -1 straighteven_test1.list -2 straighteven_test2.list
```

To visualize the weights of hidden unit n, type

`./hidtopgm face.net` *image-filename* `32 30` n

Invoking `xv` on the image *image-filename* should then display the range of weights, with the lowest weights mapped to pixel values of zero, and the highest mapped to 255.

4 Documentation

The code for this assignment is broken into several modules:

- `pgmimage.c`, `pgmimage.h`: the image package. Supports read/write of PGM image files and pixel access/assignment. Provides an `IMAGE` data structure, and an `IMAGELIST` data structure (an array of pointers to images; useful when handling many images).
- `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.
- `imagenet.c`: 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 your own recognition tasks, to set up appropriate target vectors for the output encodings you choose.

- facetrain.c: the top-level program which uses all of the modules above to implement a "glickman" recognizer. 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 your own tasks.

- 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.

4.1 facetrain

facetrain has several options which can be specified on the command line. Here we briefly describes how each option works. A very short summary of this information can be obtained by running facetrain 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. The default seed is 102194 (guess what day it was when I wrote this document). 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 text file which contains a list of image pathnames, one per line, that will be used for training. 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 text file which contains a list of image pathnames, one per line, that will be used as a 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.

When you run `facetrain`, 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.

4.2 Tips

Although you do not have to modify the image or network packages, you will need to know a little bit about the routines and data structures in them, 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 `imagenet.c` and `facetrain.c` to see how the routines are actually used.

In fact, it is probably a good idea to look over `facetrain.c` first, to see how the training process works. You will notice that `load_target()` from `imagenet.c` 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, 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.

4.3 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 `double`s 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 kth 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 randomly initialized 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.

4.4 The image package

The image package provides a set of routines for manipulating PGM 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_open(filename)
    char *filename;
```

Opens the image given by `filename`, loads it into a new `IMAGE` data structure, and returns a pointer to this new structure. Returns `NULL` on failure.

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

Creates an image in memory, with the given filename, of dimensions `nrows` × `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`.

4.5 `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.

4.6 `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.