Homework 3: Neural networks and face images

15-681, 15-781: Machine Learning
Tom Mitchell
Carnegie Mellon University

Due: October 15, 1998

1. Introduction

This assignment gives you an opportunity to apply neural network learning to the problem of face recognition. The face images you will use are faces of students from earlier Machine Learning classes.

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.

1.1. The face images

The image data can be found in `/afs/cs/project/theo-8/faceimages/faces`. This directory contains 20 subdirectories, one for each person, named by userid. Each of these directories contains several different face images of the same person.

You will be interested in the images 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); 4 indicates a quarter-resolution image (32×30). For this assignment, you will be using the quarter-resolution images for experiments, to keep training time to a manageable level.

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).

1.2. Viewing the face images

To view the images, you can use the program `xv`. This is available as `/usr/local/bin/xv` on Andrew machines, and `/usr/misc/.X11-others/bin/xv` or `/usr/local/bin/xv` on CS machines. `xv` handles a variety of image formats, including the PGM format in which our face images are stored. While we won't go into detail about `xv` in this document, we will quickly describe the basics you need to know to use `xv`.

To start `xv`, just specify one or more images on the command line, like this:

```
xv /afs/cs/project/theo-8/faceimages/faces/glickman/glickman_straight_happy_open_4.pgm
```

This will bring up an X window displaying the face. Clicking the right button in the image window will toggle a control panel with a variety of buttons. The `Dbl Size` button doubles the displayed size of the image every time you click on it. This will be useful for viewing the quarter-resolution images, as you might imagine.

You can also obtain pixel values by holding down the left button while moving the pointer in the image window. A text bar will be displayed, showing you the image coordinates and brightness value where the pointer is located.

To quit `xv`, just click on the `Quit` button or type `q` in one of the `xv` windows.

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 also supplying you with an image package for accessing the face images, as well as 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 `/afs/cs/project/theo-8/faceimages/code`. Copy all of the files in this area to your own directory, and type `make`. Note: take care to use `cp *` instead of `cp *.*` in order to ensure that you get the `Makefile`. When the compilation is done, you should have one executable program: `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 has been compiled and tested successfully on CS-side Alphas, DecStations, Sun SPARC-2s, and IBM PowerPC's, and Andrew-side DecStations and Sun SPARC-5s. 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.

2. The Assignment

2.1. Part I (required)

Turn in a short write-up of your answers to the questions found in the following sequence of initial experiments.

1. Issue the following command in your home directory to obtain the training and test set data for this assignment:

   `cp /afs/cs/project/theo-8/faceimages/trainset/*.list .`

2. The code you have been given is currently set up to learn to recognize the person with userid `glickman`. Modify this code to implement a "sunglasses" recognizer; i.e., train a neural net which, when given an image as input, indicates whether the face in the image is wearing sunglasses, or not. Refer to the beginning of Section 3 for an overview of how to make changes to this code.

3. Train a network using the default learning parameter settings (learning rate 0.3, momentum 0.3) for 75 epochs, with the following command:

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

   `facetrain`'s arguments are described in Section 3.1.1, but a short description is in order here. `shades.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 (namely those whose filenames are in the `straightrnd_train.list` file), 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`).

4. What code did you modify? What was the maximum classification accuracy achieved on the training set? How many epochs did it take to reach this level? How about for the validation set? The test set? Note that if you run it again on the same system with the same parameters and input, you should get exactly the same results because, by default, the code uses the same seed to the random number generator each time. You will need to read Section 3.1.2 carefully in order to be able to interpret your experiments and answer these questions.

5. Now, implement a 1–of–20 face recognizer; i.e. implement a neural net that accepts an image as input, and outputs the userid of the person. To do this, you will need to implement a different output encoding (since you must now be able to distinguish among 20 people). (Hint: leave learning rate and momentum at 0.3, and use 20 hidden units).

6. As before, train the network, this time for 100 epochs:

   ```
   facetrain -n face.net -t straighteven_train.list -1 straighteven_test1.list
   -2 straighteven_test2.list -e 100
   ```

   You might be wondering why you are only training on samples from a limited distribution (the "straight" images). The essential reason is training time. If you have access to a very fast machine (anything slower than an Alpha or Sun4 may be too slow), 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.

   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.

7. Which parts of the code was it necessary to modify this time? How did you encode the outputs? What was the maximum classification accuracy achieved on the training set? How many epochs did it take to reach this level? How about for the validation and test set?

8. Now let's take a closer look at which images the net may have failed to classify:

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

   Do there seem to be any particular commonalities between the misclassified images?

9. Implement a pose recognizer; i.e. implement a neural net which, when given an image as input, indicates whether the person in the image is looking straight ahead, up, to

the left, or to the right. You will also need to implement a different output encoding for this task. (Hint: leave learning rate and momentum at 0.3, and use 6 hidden units).

10. Train the network for 100 epochs, this time on samples drawn from all of the images:

    ```
    facetrain -n pose.net -t all_train.list -1 all_test1.list
    -2 all_test2.list -e 100
    ```

    Since the pose-recognizing network should have substantially fewer weights to update than the face-recognizing network, even those of you with slow machines can get in on the fun of using all of the images. In this case, 260 examples are in the training set, 140 examples are in test1, and 193 are in test2.

11. How did you encode your outputs this time? What was the maximum classification accuracy achieved on the training set? How many epochs did it take to reach this level? How about for each test set?

12. 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:

    `hidtopgm pose.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. If the images just look like noise, try retraining using `facetrain_init0` (compile with `make facetrain_init0`), which initializes the hidden unit weights of a new network to zero, rather than random values.

13. Do the hidden units seem to weight particular regions of the image greater than others? Do particular hidden units seem to be tuned to different features of some sort?

2.2. Part II. (optional!)

This part of the assignment is absolutely not required. However, given that you now know your way around the code, you might be interested in exploring some more advanced issues. Feel free to work along or form a team with one or two other students, and pick some interesting topic of your own choice – be creative!! Run some experiments, and prepare a short write-up of what your group's idea, experimental results, and any conclusions you draw (a few pages should be sufficient). For this part of the assignment, your group can turn in a single group writeup.

Some possibilities for experimentation are given below (but please don't let this list limit your thinking):

- Use the output of the pose recognizer as input to the face recognizer, and see how this affects performance. To do this, you will need to add a mechanism for saving the output units of the pose recognizer and a mechanism for loading this data into the face recognizer.

- Learn the *location* of some feature in the image, such as eyes. You can use `xv` to tell you the coordinates of the feature in question for each image, which you can then use as your target values.

- Take a look at the additional data from an earlier year's class in

  `/afs/cs.cmu.edu/project/theo-8/ml94faces`.

  What techniques can you employ to train nets to generalize better from one dataset to the other?

- How do nets perform if trained on more than one concept at once? Do representations formed for multiple concepts interfere with each other in the hidden layer, or perhaps augment each other?

- Use the image package, weight visualization utility, and/or anything else you might have available to try to understand better what the network has actually learned. Using this information, what do you think the network is learning? Can you exploit this information to improve generalization?

- Change the input or output encodings to try to improve generalization accuracy.

- Vary the number of hidden units, the number of training examples, the number of epochs, the momentum and learning rate, or whatever else you want to try, with the goal of getting the greatest possible discrepancy between train and test set accuracy (i.e., how badly can you make the network overfit), and the smallest possible discrepancy (i.e., what is the best performance you can achieve).

3. 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). **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 not need to modify any code in this module to complete the assignment.**

- `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 the face recognizer and the pose recognizer, 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 "TA" 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 face and pose recognizers.

- `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.

Although you'll only need to modify code in `imagenet.c` and `facetrain.c`, 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. `facetrain`

3.1.1. Running `facetrain`

`facetrain` 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 `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.

3.1.2. Interpreting the output of `facetrain`

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.

3.2. 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`, `facetrain.c`, and `facerec.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, 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 `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 `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 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.

3.4. 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` $\times$ `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.