Project 1: Neural Network on Face Images

Due Date: April 17, 2017

Three most important things:

- Before you start this project, please read this document carefully, and make sure your submission meets all requirements.
- You may form a group of 1-3 members with one leader to accomplish this project and submit one report. Intra-group collaboration is encouraged, but inter-group "collaboration" in this project is not permitted.
- All the materials provided by this project are not permitted to be redistributed for any purpose.

1 Introduction

This assignment gives you an opportunity to apply neural network for learning the problem of face recognition. It is divided into two parts. For the first part you will do some experiments with a neural network program to train a sunglass recognizer and a facial expression recognizer. For the second part, which is optional, you have the option of studying some issues based on your own choice. The face images consist of faces of 20 persons. You will not need to do much coding for this assignment, but training the neutral networks may take some time.

It is recommended that you read this document selectively. Section 2 describes the face images you will use for the experiments. In Section 3, we give an introduction to the three projects, on one of which you need to make some modifications for the task. Section 4 gives the assignment together with some requirements. Submission details are given in Section 5. In appendix A, you can find

detailed documents on codes of the three projects. We suggest you read Section 4 first after you finished Section 2 and then refer to Section 3 when necessary because the description about the projects in that section is a little bit long and not so interesting.

2 The Face Images

The image data [1] can be found in `homedirectory/faceimages/faces` directory [1]. 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.

To view the images, you can use the program `IrfanView` or any other that can handle the PGM format in which the face images are stored. The `IrfanView` program is freely available at `http://www.irfanview.com/`.

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 can be 3 values: *empty, 2, 4*. *empty* 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). In this assignment, you need to use the full-resolution images for experiments.

[1] We recommend you first create a local directory (e.g., `C:\assignment`) on your hard disk, then download the .zip file and extract it into this directory. Thus `homedirectory` for you is the `src` folder in the assignment directory, i.e. `C:\assignment\src`.

If you look closely into 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 possible variations).

3 Three Projects

There are 3 projects [1] in the *homedirectory*, stored in *facetrain*, *hidtopgm*, *outtopgm* subdirectories respectively. The codes have been compiled and tested successfully on Windows 7 system, using Visual Studio 2015. Feel free if you wish to run them on some other platforms or in other development environments. Details of the routines, explanations of the source files, and related information can be found in Appendix A of this document.

Briefly, `facetrain` takes lists of image files as input, and uses them as training and testing sets for a neural network. `facetrain` can be used for training and/or recognition, and it also has the capability of saving networks to files. You can use the `hidtopgm` program to visualize the weights of the hidden units with respect to each pixel, and refer to Section 3.3 for details. The `outtopgm` program is used to look at the weights of the output unit. There is no need to make any changes to these two programs.

3.1 Tips

Although you do not have to modify the images 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 detail. You can refer to `imagenet.cpp`, `FaceTrain.cpp`, and `backprop.cpp` to see how the routines are actually used.

In fact, it is a good idea to look over `FaceTrain.cpp` first, to see how the training process works. You will notice that function `load_target()` in `imagenet.cpp` 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.

The purpose 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.

3.2 Neural network project

The project package contains C code for a three-layer fully-connected feedforward neural network which uses the backpropagation algorithm to tune its weights. It also provides you with an image package for accessing the face images, as well as the top-level program for training and testing, as a basis for you to modify.

3.2.1 Running `facetrain`

`facetrain` has several options which can be specified on the command line. Here we briefly describe 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.

Note that please, when you begin a completely new training task, remember to delete the old network file, i.e. `sunglasses.net`. Otherwise the file will be loaded and the network will be trained incrementally based on it, and thus the results might differ surprisingly from what you expected.

`-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. 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 running `facetrain` in IDE of Visual Studio 2015, you may need to make some modifications in its project settings, including executable, working directory and program arguments. Simpler ways are that you run it in DOS environment or just run the "`demo.bat`" file.

3.2.2 Interpreting the output of `facetrain`

When you run `facetrain`, it will first load 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 performance on training and test sets is evaluated and printed on the screen, 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 sequence 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 during backpropagation, 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 are 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 are correctly classified.

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

3.3 `Hidtopgm` project

It's unnecessary for you to change any code of this project. `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. It is the output file of running `facetrain`.

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.

3.4 `Outtopgm` project

It's unnecessary for you to change any code of this project. `outtopgm` takes the following fixed set of arguments:

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

This is the same as `hidtopgm`, but for output units instead of hidden units. Be sure you specify x to be 1 plus the number of hidden units, so that you can see the weight w_0 as well as weights associated with the hidden units. For example, to see the weights for the 2nd output unit of a network containing 3 hidden units, we can run the following command 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. It is the output file of running `facetrain`.

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 plus 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.

4 Assignment

The assignment is divided into two parts. The first part is required and the second is optional, but it may help you score higher for this assignment if your answers are perfect. Nevertheless, the highest score is limited by the maximum given to this assignment. We encourage the students who have interest in neural network or face recognition to have a try.

4.1 Part I (Required)

4.1.1 Tasks

- **The provided code** is currently set up to learn to recognize "sunglasses"; i.e., given an image as input, to recognize whether the face in the image is wearing sunglasses, or not. Issue the following command (or just double click `demo.bat`) in your home directory to observe the training process:

```
facetrain\release\facetrain -n sunglasses.net -t trainset\
all_scale1_train.list -1 trainset\all_scale1_test1.list
-2 trainset\all_scale1_test2.list -e 200
```

- **Your task** is to modify this code to implement a "facial expression" recognizer; i.e. implement a neural network which, when given an image as input, can indicate whether the person in the image is *angry, happy, neutral or sad*. You need to implement a different output encoding for this task. Find the "`load_target`" function in the project and try to make changes to this code.You will also need to modify the performance evaluation routines e.g. `evaluate_performance()`. If you need to change the structure of the network, you may also need to modify the parameters used when creating the network in "`backprop_face`" function.

4.1.2 Questions You Need to Answer

For the tasks listed above, conduct the experiments and answer the following questions:

- How did you encode the outputs? Change the parameters and retry the experiments: the number of hidden units, learning rate and momentum, the number of epochs, the initial weights, etc. 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? Draw the curves for the experiment outputs. We suggest you put `trainperf`, `t1perf` and `t2perf` in one figure, and `trainerr`, `t1err` and `t2err` in another. The x-axis denotes the number of epochs and the y-axis denotes the corresponding outputs.

- Now, try taking a look at how backpropagation tuned the weights of the hidden units with respect to each pixel. Use *hidtopgm* and *outtopgm* to visualize the weights of hidden units and the output units. The lowest weights are mapped to pixels with the value of zero (in terms of grayscale), and the highest mapped to 255. Use the PGM viewer program to display the weights. What kind of conclusions can you draw from the weights learned? Do the hidden units seem to weight particular regions of the image greater than others?

- Do you encounter any difficulty in the experiments? How did you solve the

problem? Can you describe your findings during the experiment or anything interesting?

4.2 Part II (Optional)

Now that you know your way around `facetrain`, it's time to have some fun. Pick some interesting topic of your own choice – be creative! Run some experiments, and prepare a short write-up about your group's ideas, experimental results, and any conclusions you draw. Some possible points for experiment (please don't let the list limit your innovation, you can try anything) are:

4.2.1 Tasks

- Implement a 1-of-20 face recognizer; i.e. implement a neural network that accepts an image as input, and outputs the userid of the person. To do this, you will also 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).
- Use the output of the facial expression 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.
- How do networks perform if they are 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?
- What if the network, which was trained using the data set from one application environment, is tested by the data generated in another environment which is totally irrelevant? (for example, consider the scenario that we train the classifier using the images in your dormitory, and then test it using the images in the lab) Why the results are good or not? Hint: you may find other face recognition data via Internet.

4.2.2 Questions You Need to Answer

- Vary the parameters and try the experiments, with the goal of getting the greatest possible test set accuracy (i.e., what is the best performance you can achieve). Draw the curves for the experiment outputs.

- Use the image package, weight visualization utility, and/or anything else you might be available to try to understand better what the network has actually learned. Using this information, what do you think the network is learning?
- Write something interesting you encounter during the experiments as well as your conclusions.

5 Submission

Please submit your report to the course website before **23:59:59, April 17, 2017**. You can write it either in English or Chinese. The report should be limited within 3 pages using the given template, provided in `template` folder, including all figures and tables. Both LaTeX and Microsoft Word templates are provided, you can find them in the `template` folder. The file format of your report must be **Portable Document Format(.pdf)**. Moreover, if your experiment results are very convincing and the methods you proposed are of novelty, you can refuse to provide the details in your report. Please note that, for fairness, there will be punishment on late submission: the report, which is worth full credit by the end of the due date, will be worth only half credit for the next week (23:59:59, April 24, 2017) and **ZERO** credit after that. Therefore, do finish your work on time please.

References

[1] T. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

A Documentation on the Codes

A.1 `facetrain` project

- `pgmimage.cpp`, `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.cpp`, `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.cpp`: 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.cpp`: the top-level program which uses all of the modules above to implement a 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 pose recognizers.

- `hidtopgm.cpp`, `outtopgm.cpp`: the weight visualization utility for hidden units and output units respectively. It's not necessary to 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.cpp` and `facetrain.cpp`, 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.

A.2 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 $[-0.1, 0.1]$. 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` if failed.

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

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

A.3 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 when success, 0 when 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`.