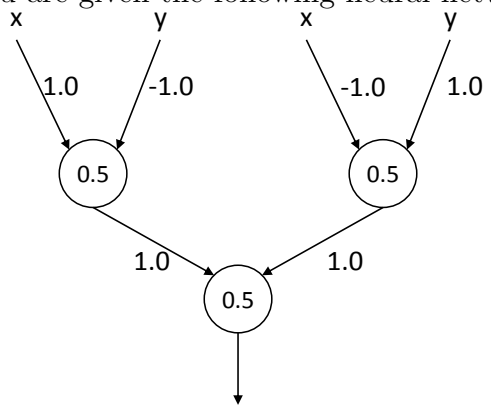
CSCI 360 – Project #2

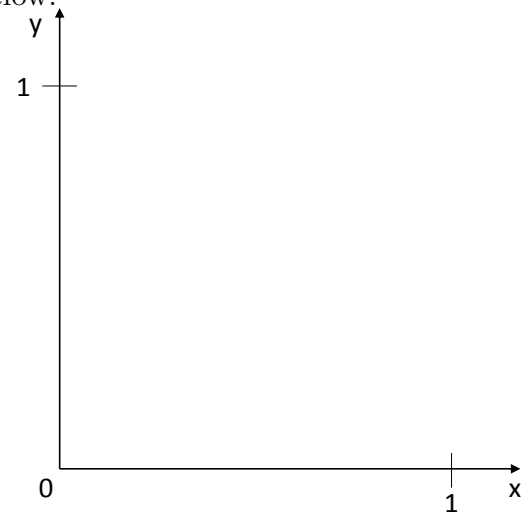
Unlike the first project, this project has three parts. This project will likely take more time to complete than Project 1, so start working on it as soon as possible.

Part 1 (Theoretical): Neural Networks

You are given the following neural network:



Numbers labeling the edges are weights, and numbers labeling the nodes are thresholds for a threshold function as activation function. Draw the area that contains the points (x, y) (x and y can take any real value in range [0,1]) for which the given neural network will output a 1, using a coordinate system like the one given below:



Part 2 (Programming): Local Search

In this part of the project, you will use the local search methods you have learned in class (such as hill-climbing and simulated annealing) to generate puzzles that are optimized to be challenging and entertaining. We provide you with code that generates and evaluates the puzzles (with respect to some value function that we describe below), but you will have to implement the local search part that finds puzzles with high values. Your program needs to terminate within 1 minute of wall clock time and report the best puzzle it has found.

The Puzzle

The puzzle consists of a grid with r rows and c columns of cells. Each cell contains exactly one integer in the range from i to j (inclusive), where i and j are positive integers. The player of the puzzle has to start in the upper-left cell (= the start cell) and move with the smallest number of actions to the lower-right cell (= the goal cell). If the player is in a cell that contains integer x, then they can perform one of at most four actions, namely either move x cells to the left (L), x cells to the right (R), x cells up (U), or x cells down (D), provided that they do not leave the grid. An example puzzle of size 5×5 is given below:

3	2	1	4	1
3	2	1	3	3
3	3	2	1	4
3	1	2	3	3
1	4	4	3	G

The shortest solution for the instance above is 19 moves: R D L R U D R L R L U D L D R R U D D.

The Value Function for the Puzzle

Below are some of the features that are expected from a 'good' puzzle that is both challenging and entertaining. We list these features just to give you an idea of the kind of puzzles that we are trying to generate. You are not expected to program anything related to these features. The code that we provide takes care of that.

- The puzzle has a solution.
- The puzzle has a unique shortest solution.
- The puzzle contains as few *black holes* (dead ends) as possible. Define a *reachable cell* as a cell that can be reached from the start with a sequence of actions. Define a *reaching cell* as a cell from which the goal can be reached with a sequence of actions. A cell is a black hole if and only if it is a reachable, non-reaching cell.

- The puzzle contains as few *white holes* as possible. A cell is a white hole if and only if it is a reaching, non-reachable cell.
- The puzzle contains as few *forced forward moves* as possible. A forced forward move occurs when there is only one action that leaves a reachable cell.
- The puzzle contains as few *forced backward moves* as possible. A forced backward move occurs when there is only one action that reaches a reaching cell.

Using these features, we develop the following value function to evaluate a given puzzle:

- We multiply the length of a shortest solution by 5.
- We add $r \times c$ (r = number of rows, c = number of columns) points if there is a unique shortest solution.
- We subtract 2 points for each white or black hole.
- We subtract 2 points for each forced forward or backward move.
- We subtract $r \times c \times 100$ points if the puzzle does not have a solution.

Generating Good Puzzles

We provide you with a code that generates puzzles (but not necessarily good ones). The program reads the parameters r, c, i and j from the command line and outputs a single puzzle of size $r \times c$ where each cell contains an integer between i and j (inclusive). Running 'make' compiles and runs the code using the parameters `5 5 1 4`, which should generate an output similar to the one given below. To run the program with different parameters, you can modify the makefile to change the default parameters, or execute the program directly from the command line.

```
./PuzzleGenerator 5 5 1 4
Generating a 5x5 puzzle with values in range [1-4]

Puzzle:
3 2 1 4 2
3 4 1 2 1
4 3 2 2 1
3 2 1 3 4
1 4 2 3 0

Solution: Yes
Unique: Yes
Solution length: 14
# of black holes: 1
```

```
# of white holes: 3
# of forced forward moves: 5
# of forced backward moves: 9
Puzzle value: 59

Total time: 4.901224 seconds
```

The provided code has three classes. The `Timer` class is used to measure the wall clock time in seconds. The `Puzzle` class has a simple interface for generating random puzzles, generating a random successor or all possible successors of a puzzle (by changing the value of a single cell), and calculating the value of a puzzle. The `PuzzleGenerator` class utilizes the `Puzzle` class to generate 'good' puzzles. You will be modifying the `GeneratePuzzle()` member function of the `PuzzleGenerator` class to generate a 'good' puzzle using local search. We have already implemented a member function for `PuzzleGenerator`, called `RandomWalk`, for generating better than average puzzles. You should look at the source code of `RandomWalk()` to understand how to use the `Timer` and `Puzzle` classes. You are free to add new member functions and variables to the `PuzzleGenerator` class, but you should not modify any other source files.

Hints

- You are free to implement whichever local search algorithm you like. Note that a crossover function is not provided in the puzzle, so genetic algorithms might not be a good idea.
- Implementing a simple hill climbing algorithm would be pretty easy and it would get you some points (and generate decent puzzles when the puzzle size is small) but it is likely that it won't find very high quality puzzles (especially for larger puzzles, where the number of successors at each iteration can be very large). Nevertheless, it might be a good starting point.
- Your program has plenty of time to generate a good puzzle. Try including random restarts (and storing the best solution you have found so far), to try and find higher quality puzzles. If your current search for a good puzzle does not seem promising, you might want to consider terminating it early to save some time.
- You might not want to evaluate all the successors of a state. Hill climbing considers all the successors (for a 10×10 puzzle with cell values between 1-9, there are around 800 successors), which takes a lot of time to process. Simulated annealing randomly picks one and decides whether to move to that one. You can also 'sample' your successors by randomly generating a small number of them and picking the best one. It is also possible to change your successor selection strategy dynamically as the search progresses.

- A randomly generated puzzle might not be solvable. In this case, its score will be extremely low (you can also check if a puzzle is solvable by using the `HasSolution` member function of the `Puzzle` class). You can either discard it and try to generate a puzzle with a solution, or you can start searching. Even if the puzzle is unsolvable, its score can get higher with respect to the other quality metrics, and, once you reach a configuration that is solvable, its score will jump up.

- Most of the hints that we have listed above are about reminding you of the different things that you can do. It is up to you to mix and match the different techniques and test them out to create a powerful local search algorithm for generating high-quality puzzles.

Grading

Your code should compile and run on `aludra.usc.edu` (which we will use for grading). We have tested and verified that the provided code (and a solution to the project that extends the provided code) compiles and runs on `aludra.usc.edu`.

We will test your program with several benchmarks using different parameters. For each benchmark, if the quality of your puzzle is above a certain threshold, you get full points. Otherwise, you get a grade based on the ratio of your puzzle's quality to the threshold quality. You can assume that r and c will be between 5 and 10 (inclusive). Puzzles that do not have a solution do not get any marks at all. Programs that exceed the time limit (1 minute) get points deducted for each second that they are over the time limit (rounded up).

For each benchmark, the five highest quality puzzles will get bonus points. We haven't decided yet how to give the bonus points yet, but it will be so that, if a program generates the highest quality puzzle for each benchmark, it will get a total of 30 bonus points.

Part 3: Hand Gesture Recognition

In this part of the project, you are given an implementation of a back-propagation neural network that can be trained to recognize hand gestures from images. You will modify the parameters of the neural network and make observations about its training process and accuracy. We first describe the neural network and the code.

Design Decisions for Neural Networks

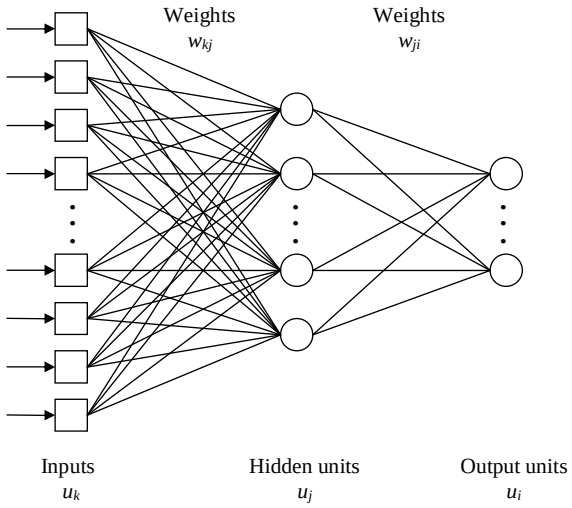


Figure 1: One-Hidden-Layer Feedforward Neural Network

We make the following design decisions:

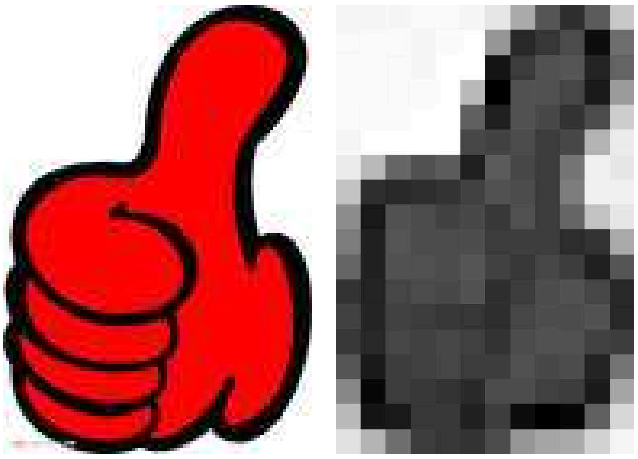
- **Topology:** We use a one-hidden-layer feedforward neural network, that consists of one layer of hidden units and one layer of output units, as shown in Figure 1. One-hidden-layer feedforward neural networks are able to represent all continuous functions if they have a sufficiently large number of hidden units and thus are more expressive than single perceptrons.
- **Activation Function:** We use the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

as activation function for all units. The derivative of the sigmoid function is

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

- **Input Encoding:** We subsample the images and then represent them as matrixes of intensity values, one per pixel, ranging from 0 (= black) to 255 (= white). Figure 2 (a) shows the original image of a hand gesture, Figure 2 (b) shows a subsampled image of size 18 × 12, and Figure 2 (c) shows the intensity values of the pixels of the subsampled image. Each intensity value is then linearly scaled to the range from 0 to 1 and becomes one input of the neural network.



247	247	248	250	250	248	231	170	88	48	92	200
250	252	251	249	254	255	151	41	55	76	15	112
252	252	250	249	255	255	22	59	84	87	37	100
252	252	250	252	255	190	1	85	83	70	26	157
250	252	252	255	255	96	32	85	74	53	75	207
248	255	255	255	255	73	69	77	61	73	180	237
238	185	103	85	93	31	93	73	61	119	240	233
195	47	41	49	53	67	85	75	60	126	243	242
143	24	83	75	68	74	57	80	60	102	194	232
124	34	86	79	77	79	60	64	62	45	51	171
103	34	83	80	79	88	69	54	71	64	31	88
53	35	70	75	74	85	49	61	78	84	70	42
67	39	67	61	57	54	46	71	79	78	85	41
56	46	70	67	70	65	53	77	83	81	67	64
87	28	64	65	70	60	64	80	74	79	37	116
137	9	62	57	52	44	77	78	65	62	29	168
189	23	27	56	58	34	63	14	0	9	95	217
228	181	114	62	56	53	78	153	168	172	241	246

(a) *(b)* *(c)*

Figure 2: Input Encoding

- **Output Encoding:** Each output of the neural network corresponds to a combination of the values of its output units. Imagine that a neural network has to decide whether an image contains the hand gesture "thumbs up." In this case, we can use one output unit and map all output values greater than 0.5 to "yes" and all output values less than or equal to 0.5 to "no."

- **Error Function:** We use the sum of squared errors as error functions. Consider a single training example e, where $t_i[e]$ represents the desired output and o_i the actual output of output unit u_i. Then, the difference of the desired and actual output is calculated as

$$E[e] = \frac{1}{2} \sum_i (t_i[e] - o_i)^2.$$

The Back-Propagation Algorithm

The pseudocode of the back-propagation algorithm repeatedly iterates over all training examples, as shown Figure 3. Each iteration is called an epoch. For each training example e, it executes two parts of code. The first part of the code propagates the inputs forward through the neural network to calculate the outputs of all units [Lines

```
1  procedure back-propagation(trainingset, neuralnetwork, α)
2  inputs
3    trainingset: the training examples, each specified by the inputs x_k[e] and desired outputs t_i[e];
4    neuralnetwork: a one-hidden-layer feedforward neural network with weights w_kj and w_ji
5    α: the learning rate
6  repeat
7    for each e in trainingset do
8      /* propagate the input forward */
9      for each hidden unit u_j do
10       a_j := Σ_k w_kj x_k[e]
11       o_j := 1/(1 + e^{-a_j})
12     for each output unit u_i do
13       a_i := Σ_j w_ji o_j
14       o_i := 1/(1 + e^{-a_i})
15     /* propagate the error backward */
16     for each output unit u_i do
17       δ_i := o_i(1 − o_i)(t_i[e] − o_i)
18       for each hidden unit u_j do
19         w_ji := w_ji + α · δ_i · o_j
20     for each hidden unit u_j do
21       δ_j := o_j(1 − o_j) Σ_i w_ji δ_i
22       for each input unit u_k do
23         w_kj := w_kj + α · δ_j · x_k[e]
24 until the termination condition is satisfied
```

Figure 3: Back-Propagation Algorithm

8-14]. Line 10 calculates the weighted sum a_j of the inputs of a hidden unit u_j from the inputs $x_k[e]$ of the neural network. Line 11 calculates the output o_j of the hidden unit by applying the sigmoid function to the weighted sum a_i of its inputs. Similarly, Line 13 calculates the weighted sum of the inputs of an output unit u_i from the outputs o_j of the hidden units. Line 14 calculates the output o_i of the hidden unit (which is an output of the neural network) by applying the sigmoid function to the weighted sum of its inputs. The second part of the code propagates the error backward through the neural network and updates the weights of all units using gradient descent with a user-provided learning rate α [Lines 15-23], which is typically a small positive constant close to zero. The available examples are usually partitioned into the training examples and the cross validation examples, and the back-propagation algorithm terminates once the error over all cross validation examples increases (to prevent overtraining), where the error of a set of examples is the average of the errors of the examples contained in the set.

We now derive the update rule for the weights w_{ji} of the output units. The update rule of the weights w_{kj} of the hidden units is similar. The weights w_{ji} of output unit u_i are updated using gradient descent with learning rate α, that is, changed by a small step against the direction of the gradient of the error function to decrease the error $E[e]$ quickly:

$$w_{ji} := w_{ji} - \alpha \cdot \frac{\partial E[e]}{\partial w_{ji}}.$$

The input of unit u_i is $a_i = \sum_j w_{ji} o_j$ [Line 13], resulting in

$$\frac{\partial E[e]}{\partial w_{ji}} = \frac{\partial E[e]}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_{ji}} = \frac{\partial E[e]}{\partial a_i} \cdot o_j = \frac{\partial E[e]}{\partial o_i} \cdot \frac{\partial o_i}{\partial a_i} \cdot o_j.$$

The error is $E[e] = \frac{1}{2}\sum_i (t_i[e] - o_i)^2$, resulting in

$$\frac{\partial E[e]}{\partial o_i} = \frac{\partial}{\partial o_i}\frac{1}{2}(t_i[e] - o_i)^2 = -(t_i[e] - o_i).$$

The output of unit u_i is $o_i = \sigma(a_i)$ [Line 14], resulting in

$$\frac{\partial o_i}{\partial a_i} = \sigma(a_i)(1 - \sigma(a_i)) = o_i(1 - o_i).$$

Thus, the update rule for the weights w_{ji} [Line 19] is

$$\begin{aligned} w_{ji} &:= w_{ji} - \alpha \cdot \frac{\partial E[e]}{\partial w_{ji}} \\ w_{ji} &:= w_{ji} - \alpha(-(t_i[e] - o_i)o_i(1 - o_i)o_j) \\ w_{ji} &:= w_{ji} + \alpha(t_i[e] - o_i)o_i(1 - o_i)o_j \\ w_{ji} &:= w_{ji} + \alpha\delta_i o_j, \end{aligned}$$

where we define $\delta_i = (t_i[e] - o_i)o_i(1 - o_i)$ for simplification.

`gesturetrain`

You will use `gesturetrain`, which is a modified version of a program by Tom Mitchell and his students from Carnegie Mellon University. The next two sections contain modified descriptions of that program. We thank Tom Mitchell for allowing us to use this material.

Running `gesturetrain`

`gesturetrain` has several options that can be specified on the command line. A short summary of these options can be obtained by running `gesturetrain` with no arguments.

- `-n <network file>` - This option either loads an existing network file or creates a new one with the given filename. The neural network is saved to this file at the end of training.
- `-e <number of epochs>` - This option specifies the number of epochs for training. The default is 100.

`-T` - This option suppresses training but reports the performance for each of the three sets of examples. The misclassified images are listed along with the corresponding output values.

`-s <seed>` - This option sets the seed for the random number generator. The default seed is 102194. This option allows you to reproduce experiments, if necessary, by generating the same sequence of random numbers. It also allows you to try a different sequence 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 the network is only saved at the end of training if you train the network for 100 epochs (also the default).

`-t <train set list>` - This option specifies the filename of a text file that contains a list of image pathnames, one per line, that is used as training set. If this option is not specified, training is suppressed. The statistics for this set of examples is all zeros but the performance is reported for the other sets of examples.

`-1 <test set 1 list>` - this option specifies the filename of a text file that contains a list of image pathnames, one per line, that is used as non-training set (typically the cross validation set). If this option is not specified, the statistics for this set of examples is all zeros.

`-2 <test set 2 list>` - this option specifies the filename of a text file that contains a list of image pathnames, one per line, that is used as non-training set (typically the test set). If this option is not specified, the statistics for this set of examples is all zeros.

Output of `gesturetrain`

`gesturetrain` first reads all data files and prints several lines about these operations. It then begins training and reports the performance on the training, cross validation and test sets on one line per epoch:

```
<epoch> <delta> <trainperf> <trainerr> <test1perf> <test1err>
<test2perf> <test2err>
```

These values have the following meanings:

`epoch` is the number of epochs completed (zero means no training has taken place so far).

`delta` is the sum of all δ values of the hidden and output units over all training examples.

`trainperf` is the percentage of training examples that were correctly classified.

`trainerr` is the average of the errors of all training examples.

`test1perf` is the percentage of cross validation examples that were correctly classified.

`test1err` is the average of the errors of all cross validation examples.

`test2perf` is the percentage of test examples that were correctly classified.

`test2err` is the average of the errors of all test examples.

Note that accuracy is the percentage of instances that are classified correctly (the algorithm makes a binary decision, based on whether the output of the network is above 0.5), whereas the error refers to the error function defined in the "Back-Propagation Algorithm" section.

The last line of the output shows the training time of the network.

Modifying and Running `gesturetrain`

To change the parameters of the network, you only need to modify the following two lines in the `gesturetrain.c` file:

```
#define BPNN_NUM_HIDDEN_UNITS 4
#define BPNN_LEARNING_RATE 0.3
```

The values listed above are the **default values of the parameters** of the network (whenever you are asked to change one of the parameters, first restore the default values for the parameters, then make the change). The first parameter determines the number of hidden units in the neural network. The second parameter determines the value of α.

We have modified the `makefile` for your convenience. Executing the command `make testgesturetrain` compiles the code (which is necessary after you make changes to the parameters of the network), removes the previously saved network (so that the program generates a new network for you with unique initial weights determined by the random seed), and runs `gesturetrain` for 100 epochs, using predetermined training, validation, and test sets.

We have tested and verified that the provided code compiles and runs on `aludra.usc.edu`. You can also run `gesturetrain` on a different platform if you like, since we are only interested in the output of the program (and you will not submit any source files). However, we will not be able to help you if you have problems when trying to run `gesturetrain` on a different platform.

Questions

Question 1

Train a neural network using the default training parameter settings by executing the command `make testgesturetrain`.

The data for hand gesture recognition contain 156 images that are split randomly into three sets. The test set contains roughly 1/3 (= 52) of the examples. The test examples are stored in `downgesture_test2.list` and are used to evaluate the error of the neural network after training. Test examples are typically unnecessary during

training. We need them in this project to determine the error of the neural network on unseen examples after training. The remaining examples are used for training. They are split randomly into a training set that contains roughly 2/3 (= 70) of the remaining examples and a cross validation set that contains roughly 1/3 (= 34) of the remaining examples. The training examples are stored in `downgesture_train.list` and are used by the update rule to adjust the weights of all units. The cross validation examples are stored in `downgesture_test1.list` and are used to terminate training once the error over them increases. `gesture.net` contains the neural network after training.

Explain why the backpropagation algorithm uses a training and a cross validation set and terminates once the error over all cross validation examples (instead of, say, all training examples) increases. Read the section "Output of `gesturetrain`" carefully. Then, graph the error on the training set, cross validation set and test set as a function of the number of epochs.

Explain your experimental results (including your observations, detailed explanations of the observations and your overall conclusions).

Question 2

Change the number of hidden units in the neural network to 2 and train it (`make testgesturetrain`). Compare the training time and the accuracy and error of the resulting network (100^{th} epoch) to the network with the default values for the parameters.

Explain your experimental results (including your observations, detailed explanations of the observations and your overall conclusions).

Question 3

Change the learning rate, α, of the neural network to 0.8 and train it (`make testgesturetrain`). Graph the error on the training set, cross validation set and test set as a function of the number of epochs and compare it with the graph from Question 1. What can you say about the influence of α on the training process?

Explain your experimental results (including your observations, detailed explanations of the observations and your overall conclusions).

Submission

You need to submit your solution of Parts 1 and 3 (in separate PDF files named 'Part1.pdf' and 'Part3.pdf', respectively) as well as your code (PuzzleGenerator.h and PuzzleGenerator.cpp) for Part 2 through the blackboard system by Wednesday, Oct. 14, 11:59pm.

Questions

We created a discussion board specifically for this project on the blackboard system. Please also feel free to make use of the TA.