CMPT 414: Model Based Computer Vision
Term Project
Character Recognition Using a Neural Network
Cley Tang - 301116141
Jeff Hunter – 301128503

Introduction

The goal of this project is to implement a neural network that learns through the back-propagation technique. While a neural net can be used to recognize many general things, ours will be developed towards discrimination and recognition of various glyphs; in our case, the English alphabet. Our networks will be further constrained by having only layer based, as opposed to more general, topologies.

The network simulations were first built as simple two input neurons. We then had many stages of generalization and re-implementation of the higher levels: neurons to layers, layers to networks. At these intermediate stages, our networks were tested not to recognize glyphs, but to segment various hyper-planes (x,y in particular).

The final product was capable of recognizing small, relatively noise-free images of characters. More complex data-sets (databases of handwritten characters) were tested, but the learning rates and results of the networks under these conditions were much less than desired. Given more time to work on this project, we would probably spend our efforts constructing more elaborate training methods than simply iterating over a set of training data.

Development

The development of our project was geared towards a layer based neural network from the beginning. This allowed us to make many design simplifications at various levels. The two final objects we developed towards were a network capable of calculation and learning, and a trainer capable of teaching a network on various data-sets.

Neuron

The simplest object was developed first; a single neuron, first built with two inputs. Regardless of the number of inputs, a neuron in our model has four public functions: construction, calculation, learning and the returning of weights. While we considered having a selectable activation function, we decided that only a sigmoid function was necessary, as the back-prop method for learning we had chosen required it.

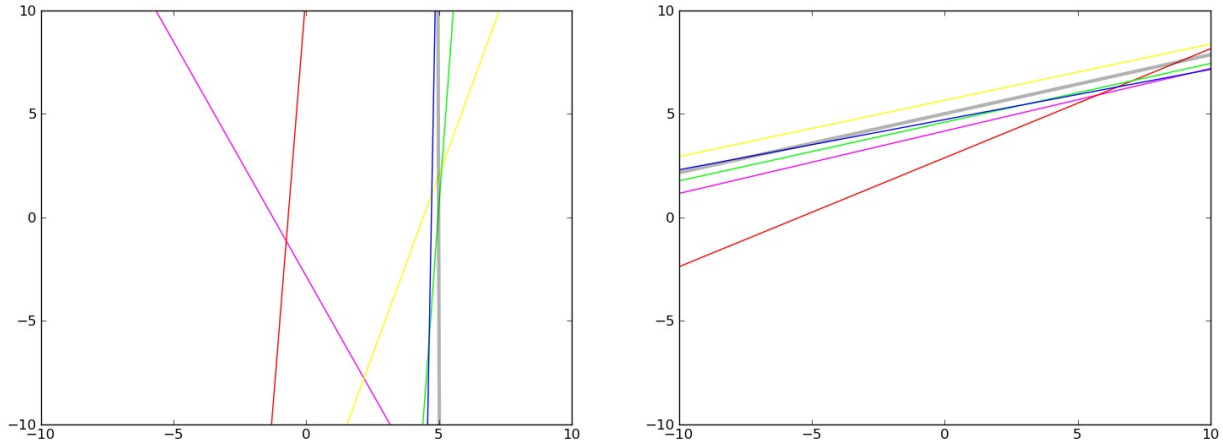
Construction is quite simple, required only three of four arguments. Argument that need to be specified are the number of inputs, the learning factor or “gain”, and the “inertia factor”. The optional argument gives a range of values that the neuron's weights and bias can be randomly selected from, the default being from 0.0 to 1.0.

Calculation is also quite simple, requiring only a list of inputs of proper length for this neuron (Ex: a four input neuron with complain when presented with five inputs for calculation). The inputs are then multiplied by their corresponding weights and summed together. A bias is then added to this subtotal, and the result is put through a sigmoid function. The result of the sigmoid then finally returned.

Learning is the only function that we had to redesign. Initially we had planned for two arguments, the inputs and desired outputs of that neuron. This design proved impossible to implement, as calculating the error terms for hidden neurons would require more information. It seemed necessary to use error term instead of desired output; testing revealed that it would be simple to calculate error terms outside of the neuron. Once an array of inputs and the corresponding error term are presented to the neuron, the weights and bias are adjusted in the standard way. For each input and associated weight, the input is multiplied by the error term and our gain factor, the weight is then adjusted by this amount. If a non-zero “inertia factor” was specified, then the weights are further adjusted by that inertia

multiplied by the last change in weights. The bias is adjusted in an identical way, the only exception being that the bias' “input” is always considered to be one.

These simple neurons can make simple decisions, splitting an N dimensional space with an N-1 dimensional hyper-plane where N is the number of inputs. Some of the first tests of our project were attempting to get a single neuron to split the x, y plane across various lines. The following images represent decision boundaries for two tests, dividing the plane along the lines: x = 5 and y = (x / 4) + 5. Colours correspond to decision boundaries after a number of training steps: Violet = 10 training points, Red = 100 points, Yellow = 1000, Green = 10,000, Blue = 100,000. The thick grey lines are decision boundaries after the network has achieved 99.5 percent accuracy.



Layer

After our neurons were complete, we began to construct our layer object. A layer of neurons represents exactly what you would expect, a collection of neurons corresponding to a single layer of a possibly multi-layered neural network. Like a neuron, a layer can calculate, learn, be constructed, and return the weights of it's constituent neurons.

Construction of a layer required the number of inputs, the number of neurons, as well as gain and inertia factors. While a neuron can accept a range for weigh starting values, the layers use the default values. The gain and inertia factors are specified as floats, not as lists, ensuring that all neurons have these same factors. We tested small (2x2) networks with non-uniform factors, but found that results were not acceptable.

Calculation for a layer is incredibly simple. A list of inputs is passed in to the layer, if the size of the input list is inappropriate, the software complains. If the list is acceptable, then each neuron in the layer append it's calculation for the given input to a result list, which is returned after all neurons have contributed.

When asking a layer to learn, one must pass in a list of inputs and a list of error terms, one error for each neuron. The layer then passes those arguments down to the neurons individual learn functions. The choice to have error terms calculated outside the layer class arose from having to calculate error terms differently for hidden and output layers. We decided to have a more general layer class as opposed to differentiating between hidden and output layers.

While on it's own, a layer could be considered to be a simple form of neural network, we never implemented any testing of a layer on it's own. Having constructed the network and layer classes at the same time, we ended up testing layers through the proxy of networks.

Networks

The most problematic part of our development was implementing networks properly, perhaps this is because we pushed much of this project's computational responsibilities towards it. Our networks stumbled over two major bugs, and even when implemented properly, it was still troublesome to produce a network that was highly accurate for glyph recognition. Like our more simple classes, the networks available methods are calculation, learning and construction.

The construction of this network follows the pattern of the previous classes. One must provide the number of inputs, a list of integers representing the size of the constituent layers (the last of which being the number of outputs of this network), and the standard gain and momentum factors. Once these are provided, the necessary layers are created and stored, and the network is ready to be used.

Under this design, calculation is quite simple, with the only necessary arguments being a list of inputs that matches this network's number of inputs. This input is passed to the first layer to calculate, with subsequent results being passed to further layers and finally returned. This function proved quite easy to implement as we faced no significant errors because of it.

Our major problems were encountered when implementing the learning method. Our design calls for two arguments to this behaviour: an array of inputs as well as an array of desired outputs. Given this information, the network should adjust itself in a suitable way using back-propagation. A further complication is that this method needs to calculate suitable error terms for every constituent neuron.

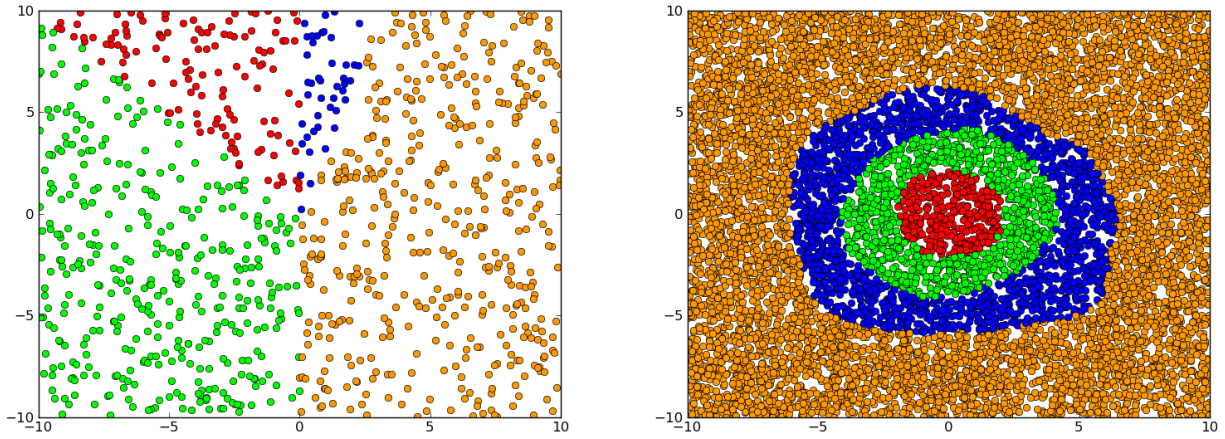
Our problems were simply in calculating error terms for hidden layers. After implementing this network, we tested it on various decisions. We noticed that any network with only one layer was functioning perfectly. As soon as an extra layer was needed (perhaps for deciding if x in x, y was between -5 and 5), the network absolutely failed, achieving accuracies in the 60-70 percent range, at best.

The first of the problems we discovered was a very simple error in calculating the error terms for hidden neurons. To properly calculate the error term for a hidden neuron, one must construct a weighted sum of error terms for all above neighbours. This weighted sum should then be multiplied by the output of the current neuron as well as one minus the same output. We had been using just the weighted sum as our error terms for hidden neurons, but correctly calculating the error terms for output layers. This would obviously cause any network of more than one layer to become quite unreliable. Upon fixing this error, our problem with multi-layered networks remained. Some error in training hidden layers still remained.

We finally discovered that we were adjusting the weights at an incorrect time in our algorithm. Our initial learning algorithm was two steps: Propagate the given input through the layers saving each intermediate output, then iterate backwards through the layers using the intermediate outputs to calculate errors and adjust weights. As the error term for a hidden neuron is based partly upon the weights above it, adjusting those weights before calculating this error term leads to an error. When the network was only one layer, this error was invisible, but for more than one layer, this bug prevented any proper functioning of the network. The proper algorithm is this: Propagate the input through the layers while remembering the outputs, iterate backwards through the layers using the outputs to calculate errors, then finally iterate over all the layers and adjust their weights. Once we repaired this error, out networks finally began performing properly.

Most of the testing done on these networks was in the form of decisions on two inputs, representing coordinates in the x, y plane. We would present various data-sets to train the networks on, and iterate them through the networks until a certain accuracy was reached. Once the training was complete, we would pass random points through the network and graph the resulting decisions. The following are graphs of the random test points on two differently trained networks, with colours representing different decision classes. The first network accepts two inputs and has two layers with

four neurons each, while the second network also accepts two inputs, but has thirty neurons in the first layer and four in the second.

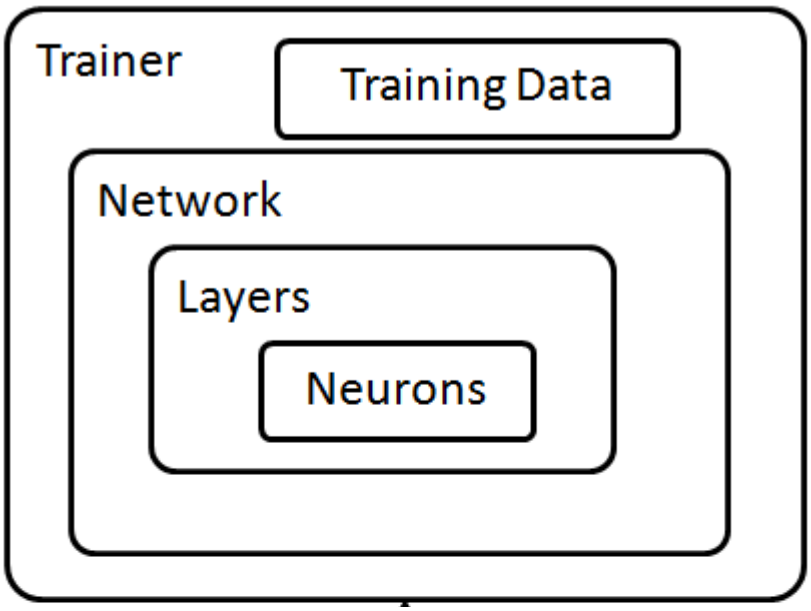


Trainer

The trainers we implemented were merely a small set of software designed to train a neural network on a given data-set. The trainer will judge the performance of the network and cease training under certain conditions, such as a given accuracy or a failure to converge. These trainers we a simple generalization of the various tests previously used on our networks and neurons. Besides the difficulty in finding a proper network topology for a given problem, we faced no significant hurdles in our trainers. As you can imagine, a trainer is not a very exciting piece of software, being just a small loop and a few checks.

Some plans were made for more elaborate trainers, such as trainers that would search through different topologies, or trainers that would breed accurate networks instead of using back-propagation, but there plans were never follow through on due to lack of time.

Design



Unsuccessful Implementation

The first data set we attempted to use for actual character recognition was one found at http://ftp.ics.uci.edu/pub/machine-learning-databases/optdigits/ . This data-set consisted of both training and testing data for handwritten Arabic numerals. This data-set represented 64 by 64 images by counting the number of "on" pixels in non-overlapping 4 by 4 squares such that you have an array of 64 integers (from 0 to 16) with the 65^{th} integer being the numeral represented (0 to 9).

Given this data and various topologies, the best accuracy we could achieve was ten percent. We are still unsure as to what is causing this lack of accuracy, but it there are a few clues that seem to imply that our network learns very slowly on this data-set. The first is that the changes in output over successive iterations of the data set were very small. The second is that the number of errors when iterating over the training set is constant. If 3000 of 3800 classifications are incorrect on the first pass through the set, on the second pass through the set, there are also exactly 3000 errors.

We decided that perhaps immediately testing on such an elaborate data set was not the wisest course of action. Our efforts then shifted to constructing a much simpler set of training images that, while not handwritten, would at least be able to recognize simple characters.

Successful Implementation

Overview

Our specific implementation of the network has the structure of a single-layer perceptron with N number of output nodes. N is the number of training images used to train the network. Our current image library consists of 5 images ranging from A to E. Each training image is 5x5 pixels in resolution with each image corresponding to 1 desired output, or class (versus many images for each desired output, or class).

The main reason why our system has only a single layer and processes 5x5 images is because, after much testing, that the system can be trained much faster and quite accurately given the size of our images. Also, since any image inputted through our program is automatically converted to a 5x5 binary image, our system is more resistant to small differences in images when determining a suitable match.

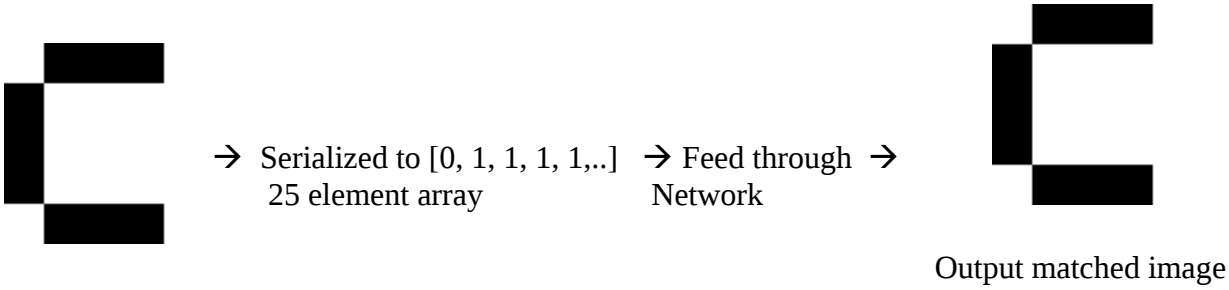
Our image library is then used to train our network. Once the training has been completed, an image (of any size) can be serialized into a 1 dimensional vector and fed through our network. The network will output a list of percentages or confidence values indicating the most likely match from the library.

For example:

Input Image:	E
Library:	[A, B, C, D, E]
Output:	[0.02, 0.03, 0.02, 0.01, 0.98]
Binary Output:	[0, 0, 0, 0, 1]

We then find the maximum value in the "natural" output list. This element is changed to one while all others are changed to zero to produce a "binary" output list (In this case indicating E is the

most likely candidate). This binary output allows for easy comparison to the desired inputs, which also have a binary nature.



Output matched image

Training Preparation

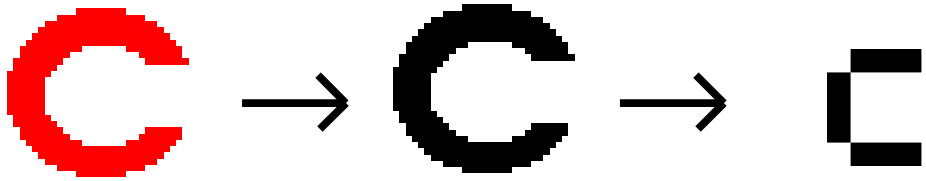
Our training data is a set of training inputs each corresponding to one desired output:

Inputs	Desired Output
[1, 0, 0, 0, 0, 1, 1, 1, 0, 0.... 1]	[1, 0, 0, 0, 0]
[1, 0, 0, 1, 0, 0, 1, 0, 0.... 1]	[0, 1, 0, 0, 0]
...	

Each input is a list of 25 binary values which represent a 5x5 image. Before any image can be run through our system, they need to be re-sized into a 5x5 image. If the images are in colour, they are then convert to a grey-scale image in which each pixel will be rounded to either 1 or 0 to produce a binary image.

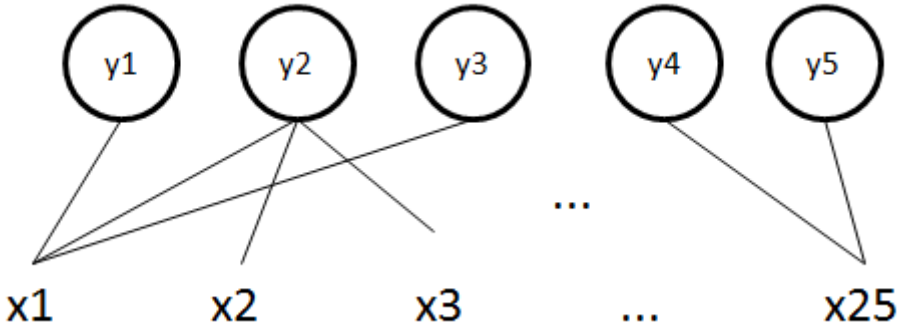
Once our array of training images has been put into the form shown above they are then used to train the system.

Example



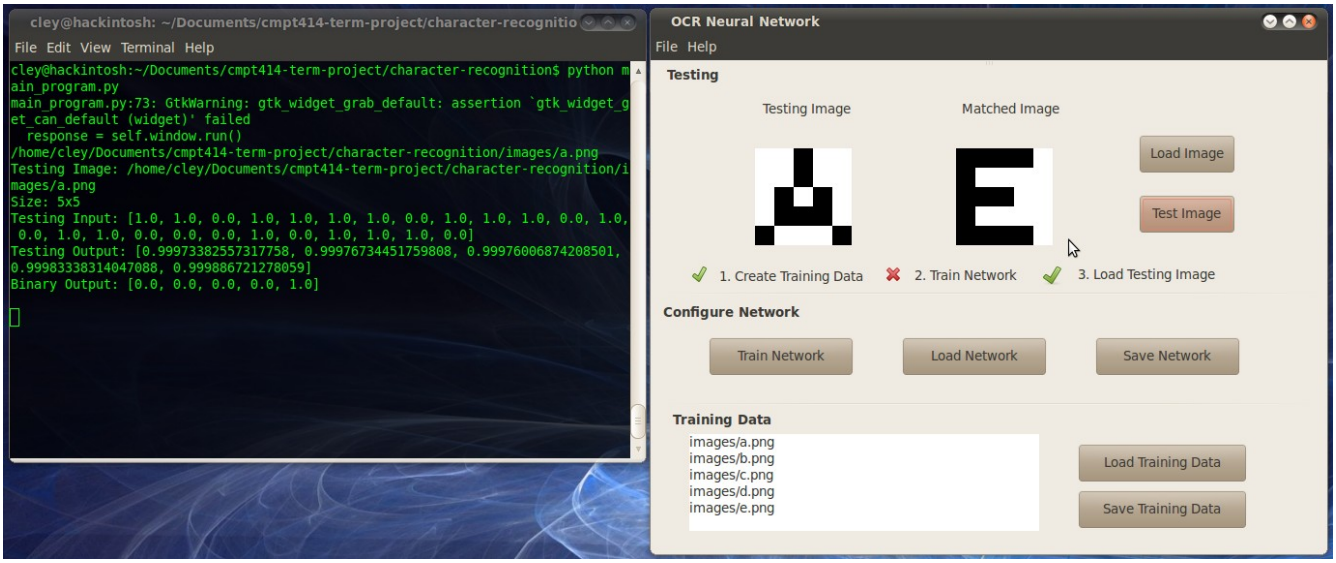
Topology

The network we have implemented for our demo can be represented as a bipartite graph. “x” represents inputs and y represents outputs. After several tests we found this network structure to be the most accurate and fastest to train given the training data and input size.

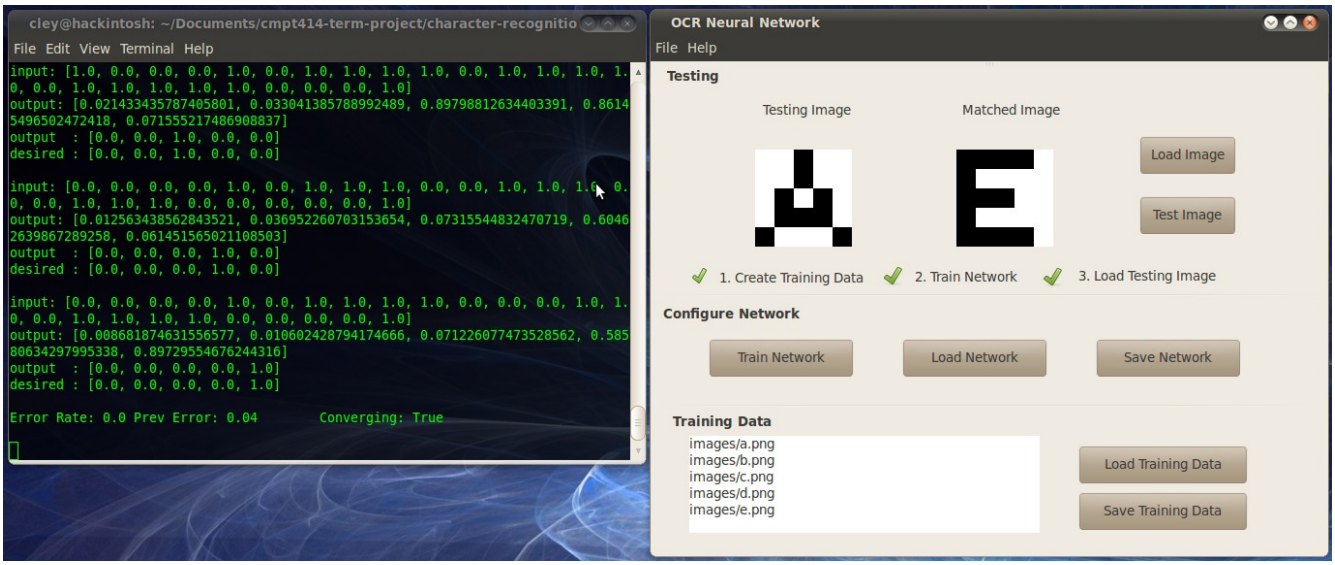


Program Usage

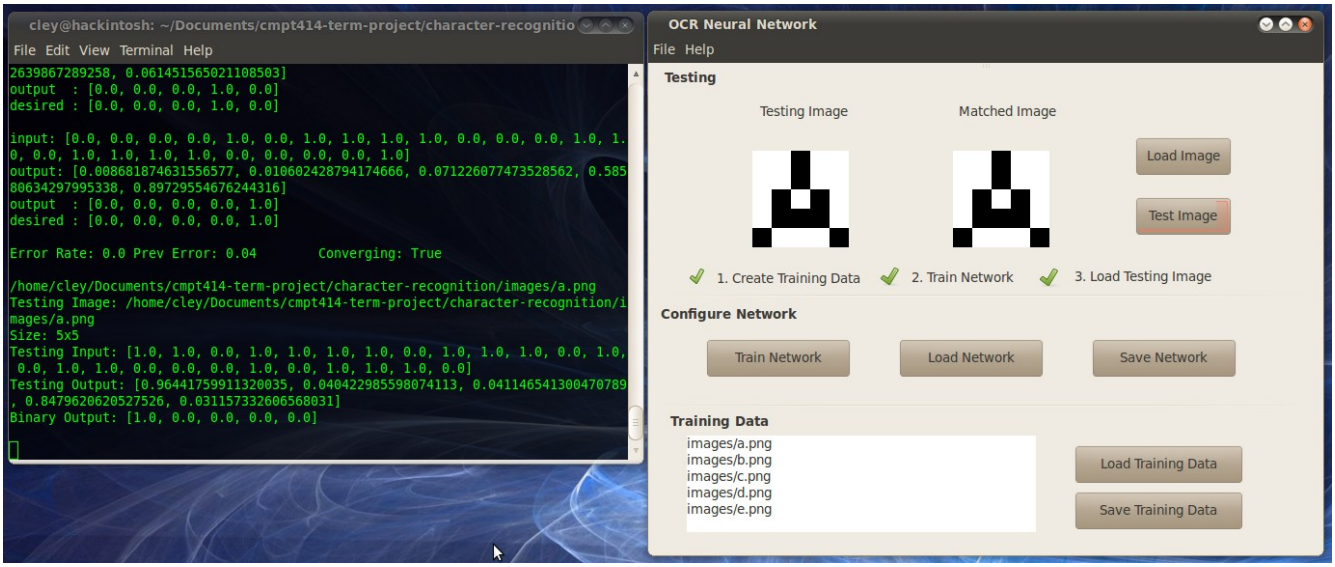
Upon starting the program you will see our main program’s GUI. You will notice that sample training data has already been added for you. This training data is a list of images that you will use to train your network. Immediately if you load an image for comparison, you may not get correct matched image outputted since the network is not trained at this point.



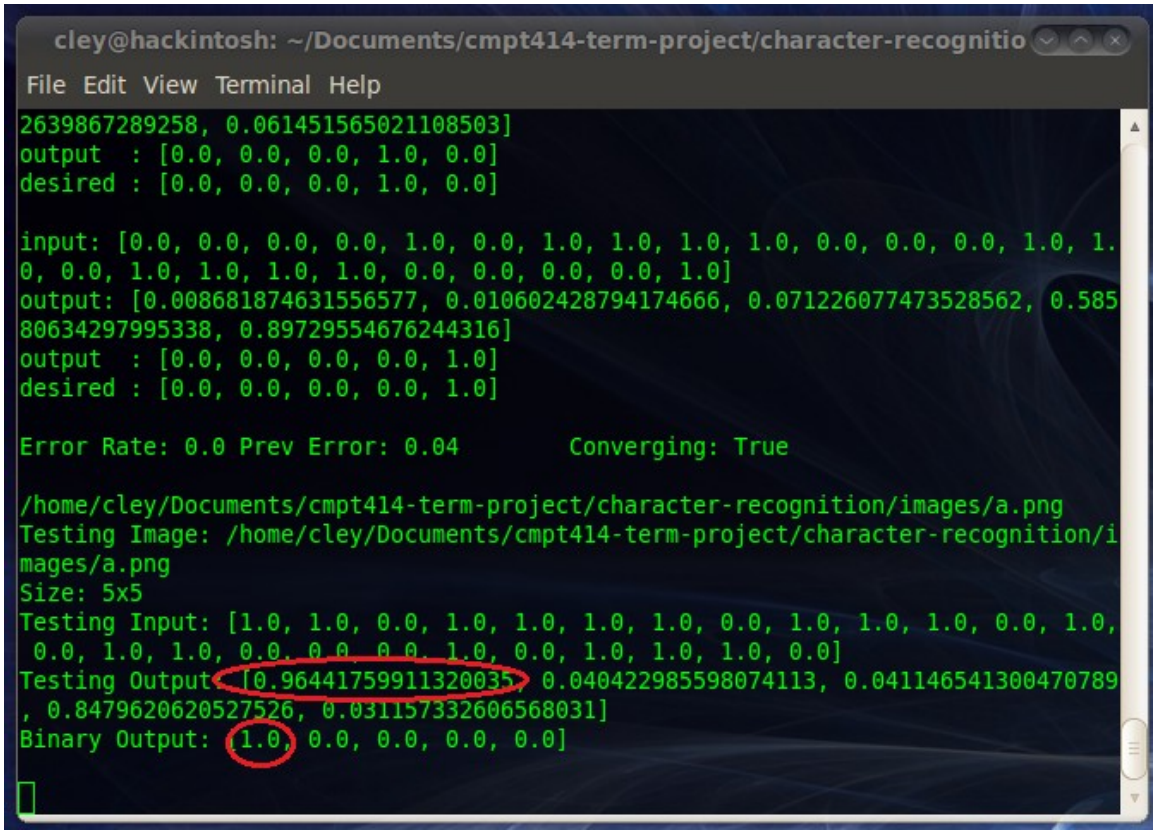
After training the network you will see your terminal output information from each training step. It displays each inputted image in binary form, the raw output as a list of floats representing confidence values for each training image, the output at that step and the desired output.



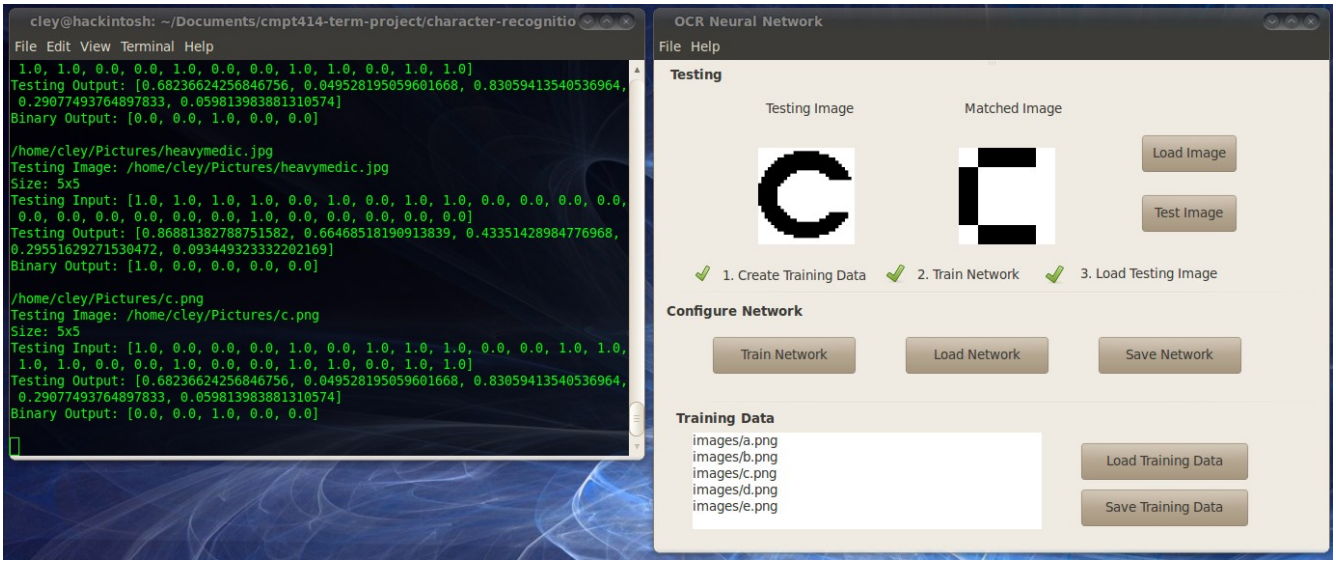
Once training is complete, if we test the system again we find that the system is now able to determine the correct letter.



If we take a closer look at the terminal output we can see that our system is 96% confident that the image tested is the letter A.



What about images that resemble a C not in our library? We input a 30x30 image for comparison resembling the letter C. Since this image is scaled down to a 5x5 image, our program is able to make a comparison and correctly identify the letter.



Now we can try a colour image and find that the matched image could be any from the library.