A Farewell to Structural Rigidity: Preliminary Domain Code

Christopher Ross, Jonathan Brant, Zak Roessler

October 12, 2015

# Overview

The ultimate goal of our research is to leverage NEAT to evolve the starting weights and the hidden layers of autoencoders in combination with traditional backpropagation for weight refinement. As the MNIST dataset has long served as a benchmark for image classification tasks, we will be training and evaluating our autoencoders on samples from that dataset and analyzing both the quantitative and qualitative aspects of the evolved features. In order to facilitate the aforementioned analysis, the following tasks need to be completed (excluding the experiment code itself):

1. Backpropagation over arbitrary topologies
2. Evaluator code for training/validation of NEAT-generated autoencoders using backpropagation and the calculation of the resulting fitness score.
3. Routines for input/output of source and result MNIST images

All of the above have been implemented and validated (save modifications to the fitness function as we run our experiments). As a side issue, preliminary tests on MNIST images have proven to be quite computationally intensive. In order to achieve more realistic run times, we may have to reduce the image resolution. Our analysis of the results will be both quantitative and qualitative. More concretely, the quantitative aspect of our analysis will largely include a study of how well the evolved autoencoders reproduce the input that’s presented (indicating that the evolved features were indeed relevant). The qualitative analysis will take a more subjective approach in that it will be concerned with an producing an appraisal of how much the output produced “looks” like the respective MNIST digit.

# autoencoder training and fitness evaluation

For each individual in each generation, our evaluation loop proceeds as follows:

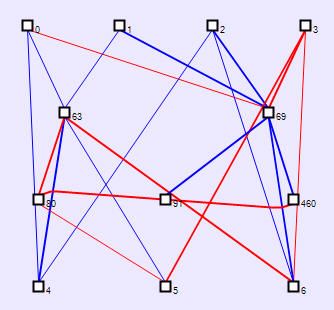
1. Load the input neurons using the normalized grayscale [0,255], normalized to [0,1] and activate the network.
2. Calculate the network error, which will run backpropagation, update the weights, and return the SSE (sum of squared error).
   1. The above process is repeated until either a given error is reached or the maximum number of backpropagation iterations have been run (both of which are tunable parameters).
3. Steps 1 and 2 is repeated for every sample in the training set.
4. The same process as step 2 is repeated for the validation set, except the network is only activated once for each sample and the SSE of each sample summed.
5. The fitness is then calculated as the difference between the number of validation samples and the sum of the errors for each validation image.
   1. Note that this may change as we experiment with other fitness functions.
6. Optionally, the utility method for image generation is given the network’s denormalized output for serialization to a bitmap representation.

# backpropagation over arbitrary topologies

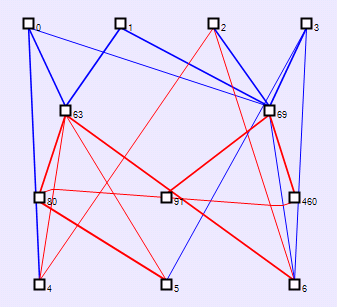
Given that NEAT doesn’t conform to the notion of strictly layered, fully connected networks, our backpropagation algorithm needs to have the ability to calculate weights updates over arbitrary topology with the hidden layers. In order to constrain the scope of this effort, we’ve restricted our backpropagation implementation to sigmoidal activations only on acyclic networks (specifically, SharpNEAT’s “FastAcyclicNetwork” class). In order to accomplish this, structural dependencies of the network are analyzed in a depth-first manner, creating sort of ad-hoc “layers” as necessitated by said connectivity. After that analysis (which is performed when the next generation of genomes are spawned) is complete, backpropagation can proceed as normal using the automatically generated “layers”.

The above implementation was unit tested independently of SharpNEAT’s typical evolutionary loop, allowing us to step through the layering and backpropagation process and verify the accuracy of each step. Given a serialized genome (which was generated from previous runs of NEAT), we were able to run several thousand iterations of backpropagation and ensure that the error continually decreased and the outputs moved toward the given inputs. One example is shown below:

An arbitrary ANN before training:



The same ANN that has been trained to output 101:



Even at this stage of our experiment, there has been evidence to suggest that some topologies are better suited as autoencoders than others. Several of our examples were able to generate the desired output after running backpropagation a small number of times, while the ANN above took several orders of magnitude more iterations of backpropagation.

# input/output of MNIST images

The path for the training data is contained in “Autoencoder.experiments.xml” along with other experiment parameters. The training and evaluation dataset consists of 28x28 pixel images. Each input pixel in the training data is represented by a greyscale value between 0 and 255. The XML tag < AutoencoderSize> is set to the width multiplied by the height in pixels of the training data. The XML tag <NumImageSamples> contains the number of images in the file to use for training/validation. Using these parameters, the EvolvedAutoencoderEvaluator constructor creates a list of data type double arrays where each array entry represents a training image.

A method called ReadImage (in ImageIoUtils) is passed the images file path, the resolution of the image (which comes from the <AutoencoderSize> configuration parameter), the number of image samples to extract for training/validation (which comes from the <NumImageSamples> configuration parameter), and the maximum pixel intensity (which is 255 for this experiment). A Binary reader is used to iterate through every sample, creating a double array the size of the image resolution to store each pixel intensity value (normalized to a value between 0 and 1). Each array is then added to the list of samples. This process is repeated for each sample image until the desired number of samples has been loaded.

To produce the output image, a function called Write Image (also in ImageIoUtils) is invoked. The parameters for Write Image include the output path and a single double array (which is the normalized activation of each ANN output neuron). Write Image denormalizes each value in the array and writes the result out as a bitmap (BMP) file. A BMP file for output was chosen to enable easy serialization and inspection of the results produced.

Below is a screenshot of the first 5 training digits for the digit 0 as seen from a raw pixel viewer (http://rawpixels.net/).



Below is the bitmap image produced as output by the Image I/O Utility class.



Through visual inspection, it is apparent that the reader and writer are producing the same image.

Below is a screenshot of the first 5 training digits for the digit 9 as seen from a raw pixel viewer(http://rawpixels.net/).

Below is the bitmap image produced as output in the Image I/O Utility class.



Again, it’s clear that the reader and writer are producing the same image.