CAP 6616 - Neuroevolution and Generative and Developmental Systems
Midterm Report

Anthony Wertz
James Schneider

October 29, 2012

1 Objective

To evolve a neural network that provides unique transformations of human faces.

2 Procedure

Using OpenCV we detect a face in any image and transform the image for input to the neural network. OpenCV is a toolkit for many computer vision tasks which is be used to identify and extract faces from an image. An extracted face is isolated and sent into the deforming network which applies some transformation. An interactive evolutionary process is used in order to generate the transformations used. The evolved neural network scans accepts a normalized (-1 to 1) coordinate, centered at the middle of the images, and returns a resulting coordinate indicating where on the original image pixel data should be pulled from. The evolutionary process takes place within a graphical user interface (GUI) which presents a small population of fifteen images to the user and allows him or her to select any number of preferred transformations. These are the basis for continued evolution.

The neuroevolutionary process is be implemented using the HyperNEAT C++ software as a basis while Python is be used as the main means of processing the results and presenting the options to the user. The two languages interact by providing Python bindings to the HyperNEAT code base. The interaction of the systems is depicted in figure 1.

As may be seen in figure 1, HyperNEAT C++ is be used to manage the internal processing of the main HyperNEAT components as that architecture already exists. However the Python system built is used to display the population (through a GUI built with Qt), evaluate the viability of offspring (e.g. determine the fitness of population individuals via operator input), and send the results back to the HyperNEAT C++ layer for evolution.

The current GUI interface may be seen in figure 2. This figure shows the general layout where a user is able to select an input image of a face to view and evolve the deformation networks. A network distortion rasterizer has been

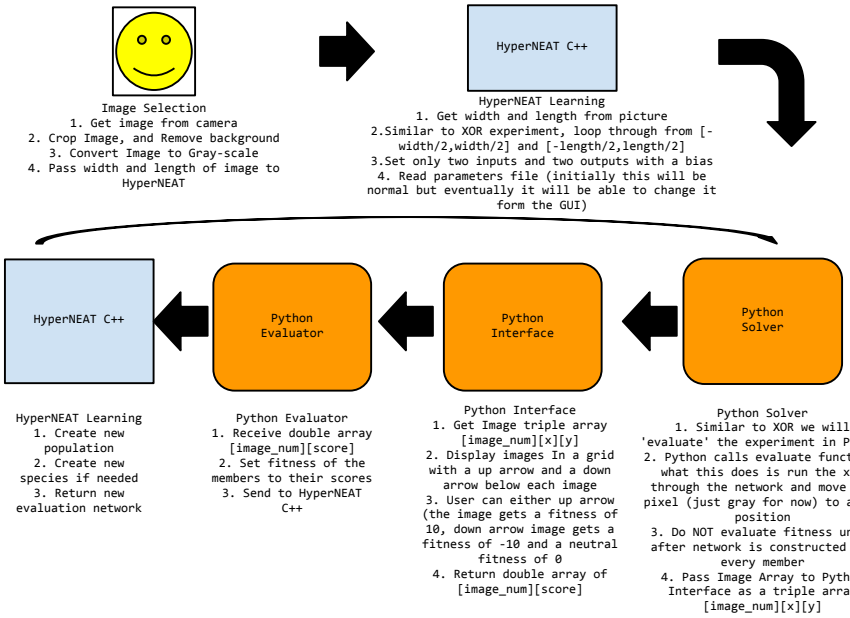

Figure 1: Architecture

built to utilize any given network representing a distortion in normalized coordinates to determine a new image. As shown, the networks represent unique and (sometimes) interesting distortions to the original image. The favored images may then be selected and evolved by clicking "Evolve" which prompts the evolution functionality in HyperNEAT.

As shown in the display, evolution parameters are be available to the operator to change on the fly during evolution. This allows adjustment of parameters throughout the process so a unique evolution can occur every time the application is used.

Current progress to this effect includes a large overhaul of the Python bindings available in the original HyperNEAT C++ code distribution to allow for Python tests to be built, the development of a new experiment module within HyperNEAT to support the necessary functionality, and the development of a graphical user interface to allow operator interaction with the evolutionary process. Most of the original work is presented in the source section in 8, though some minor changes were omitted for brevity in this document.

3 Questions

Many questions arose in the work on this project that remain to be answered.

1. Can feature extraction and image registration be used to generalize the deformation process across different images and allow deforming of faces

at different angles (even if warping to a neutral orientation before deformation is required)?

2. Will supplying information on the features of the face be useful to the evolution of deformations?

3. Can image features be used as a means of filtering out redundant transformations and supplying a criteria for a "novelty search" in the evolutionary process to supply the operator with unique transformations?

4. Can and should the network be seeded somehow to promote symmetry in the generated deformations?

5. Can the deformation process (including face detection, neutralizing, deformation, then rewarping) be completed in real time?

4 Output

The GUI provides visual feedback as to the progress of the evolutionary process. It also allows the user to output favored deformations (the image, not the network). The GUI will output the "best" networks as well as the entire last generation when the application is exited, but in the future support for exporting specific networks will be added.

As described in section 2, the GUI in figure 2 is the current operator interface.

5 Software

The following is a summary of the major software libraries used in the project:

1. HyperNeat v4.0 C++ (along with associated packages: TinyXMLDLL and JG template library

2. pyQt framework v4.2.8 for the GUI and processing

3. OpenCV v 2.4

4. Boost (especially for Python bindings)

6 Analysis

As seen in 2 the output of the current algorithm is extremely noisy. Presently, the most seen output example can be best described as an elf-like face distortion (pointed ears and chin) and an elongated neck (the dubbed "elf head Pez dispenser" transformation, see figure 3). The CPNN that produces this image is fairly simple for an odd transformation (as seen in figure 4) consisting of only three hidden nodes that only effect the x-axis. This image is so common amongst the output of the algorithm that we are attempting to determine why the bias toward this very specific deformation exits and analyzing methods to increase diversity within our evolved systems. We have observed the effect that when a user is evolving images the number of generations usually does not

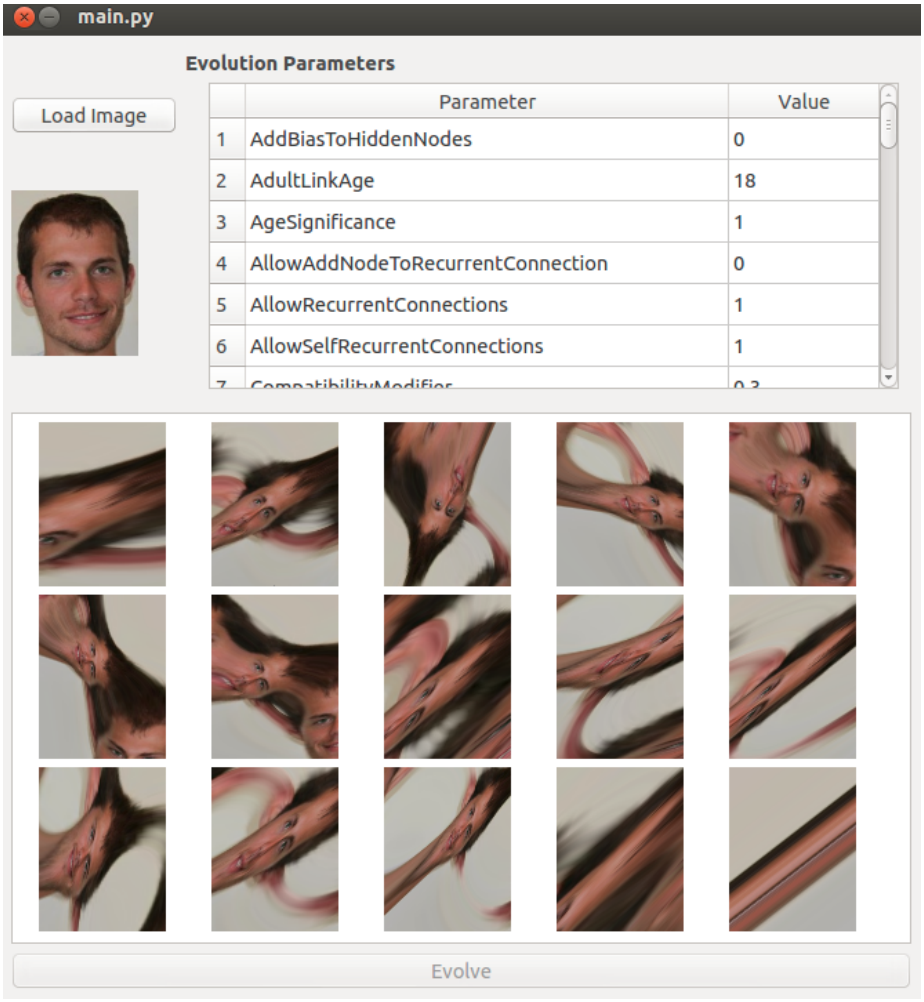

Figure 2: GUI Prototype

Figure 3: "Pez elf head" output

extend past 50, due in large part to the user quickly getting tired of picking images individually to evolve and eventually decaying into a pseudo-random type of search. This pseudo-random search makes it such that the CPNN's quickly start to converge on an answer as the species start to mimic each other in their genotype but vary in large ways with the phenotypes. Ideally these phenotype variations would be good, but because most of these phenotypes produce such a big distortion effect they are unlikely to be chosen by the human for evolution and left on the cutting room floor.

It is obvious what will happen in the situation described above as the slightest change to the genotype has a large impact on the phenotype: eventually those networks with larger variation will be evolved out due to user selection. This promotes a uniformity across the species with the phenotype expressed slightly differently, e.g. rotated along the y-axis as opposed to the x-axis. The uniformity eventually reaches endemic proportions with the all species converging on what that user preferred in initial populations. In this situation the first random population has a large effect on the final image. This represents a fundamental challenge in our evolutionary process that must be rethought for future versions. A novelty search or the use of prior information may be include in the algorithmic process in order to combat this unwanted convergence.

7 Timeline and Work Distribution

The following is the timeline and work distribution. Completed tasks are denoted in italics.

1. *9/12 - 9/24: Get build environment working between Windows and Ubuntu (hyperNEAT and C++ bindings); use environment to build XOR using python. Work Distribution: Independently get build working on respective platforms; independently implement XOR.*

2. *9/24 - 9/28: Build initial project architecture modules: (1) the python/hy-*

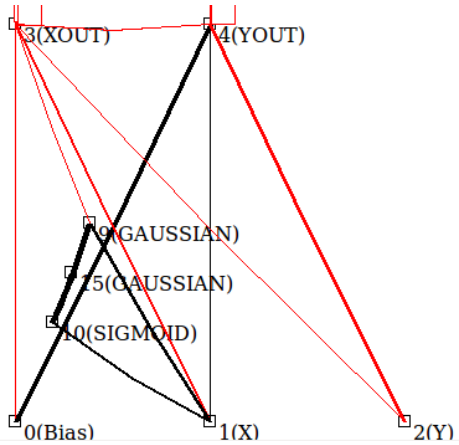

Figure 4: "Pez elf head" CPNN

perNEAT module that evolves CPPNs and sends the result to the user for evolution; (2) python/GUI module that displays the results sent by module 1, allows user selection, and returns the selection back to module 1 for further evolution. Work Distribution: Collaboratively determine python interface between modules; independently build first pass of respective modules; collaboratively work on implementation.

3. *9/28 - 10/3: Independently compile necessary documentation and graphics for respective modules; collaboratively compile documentation linking the two and considering the overall system architecture.*

4. *10/3 - 10/24: Experiment with evolution to attempt to evolve an interesting deformation (independently); through experimentation work out any software bugs; this is the baseline project implementation.*

5. *10/24 - 10/29: Consolidate interesting findings, issues, and experiences into the midterm report and presentation.*

6. 10/29 - 11/26: Use time as buffer to finish work on the baseline if good results weren't obtained. If acceptable results were found, attempt to increase applicability and complexity by looking at extensions including: detecting image features to be used in the transform instead of just pixel locations; integrating evolved neural nets in a video (real time).

7. 10/26 - 10/29: Complete Midterm report and Midterm Presentation.

8. 10/30 - 11/10: Implement "Haar Cascades" to detect facial features. These facial features will be the only ones evolved. The facial features will take the place of the whole image and instead of evolving everything on the image including head position we will just evolve these individual features.

9. 11/11 - 11/26: Collect images, videos and perform basic statistical analysis on the outputs. Perform a basic user study and develop an outline for the final paper and presentation. Write up analysis about project and discussion sections for final paper.

10. 11/27 - 12/2: Finish Final presentation and final paper.

11. 12/2 - 12/7: Complete final paper.

8 Source Code

8.1 PyHyperNEAT.cpp

```
#include "NEAT.h"

#include <boost/python.hpp>
#include <boost/python/suite/indexing/vector_indexing_suite.hpp>

#include "HCUBE_ExperimentRun.h"
#include "Experiments/HCUBE_Experiment.h"
#include "ImageExperiment/ImageExperiment.h"

using namespace NEAT;

class PyHyperNEAT
{
    public:
        /**
         * Perform HyperNEAT initialization through Python.
         */
        static void initialize( void )
        {
            // No initialization necessary at this point.
        }

        /**
         * Perform HyperNEAT cleanup through Python.
         */
        static void cleanup( void )
        {
            NEAT::Globals::deinit();
        }

        /**
         * Returns the global parameters singleton.
         */
        static NEAT::Globals & getGlobalParameters( void )
        {
            return NEAT::Globals::getSingletonRef();
        }

        /**
         * Load a previously saved population from an XML file.
         *
         * @param filename The XML file.
         * @return The population.
         */
        static NEAT::GeneticPopulation * load( const string &
            filename )
        {
            cout << "Loading population file: " << filename << endl
                ;

            TiXmlDocument doc(filename);
            bool loadStatus = false;
```

```
51
        if (iends_with(filename,".gz"))
            loadStatus = doc.LoadFileGZ();
        else
            loadStatus = doc.LoadFile();

        if (!loadStatus)
            throw CREATE_LOCATEDEXCEPTION_INFO("Error trying to
                load the XML file!");

        TiXmlElement *element = doc.FirstChildElement();
61      NEAT::Globals* globals = NEAT::Globals::init(element);

        return new NEAT::GeneticPopulation(filename);
    }

    /**
     * Converts a Python list to a STL vector.
     *
     * @param list The list.
     * @return An STL vector.
71   */
    template<class T>
    static vector<T> convertListToVector( python::list * list )
    {
        vector<T> vec;
        for ( int a=0; a<python::len(*list); ++a )
        {
            vec.push_back(
                boost::python::extract<T>((*list)[a]) );
        }
81      return vec;
    }

    /**
     * Converts a STL vector to a Python list.
     *
     * @param vector The vector.
     * @return The Python list.
     */
    template<class T>
91  static python::list convertVectorToList( const std::vector<
        T> & vector )
    {
        python::object get_iter = python::iterator<std::vector<
            T> >();
        python::object iter = get_iter(vector);
        python::list l(iter);
        return l;
    }

    /**
     * Sets a substrate layer information.
101  */
    static void setLayerInfo(
         shared_ptr<NEAT::LayeredSubstrate<float> > substrate,
         python::list _layerSizes,
         python::list _layerNames,
         python::list _layerAdjacencyList,
         python::list _layerIsInput,
         python::list _layerLocations,
         bool normalize,
```

```
            bool useOldOutputNames )
111     {
            NEAT::LayeredSubstrateInfo layerInfo;

            for(int a=0;a<python::len(_layerSizes);a++)
            {
                layerInfo.layerSizes.push_back(
                    JGTL::Vector2<int>(
                    boost::python::extract<int>((_layerSizes)[a
                        ][0]),
                    boost::python::extract<int>((_layerSizes)[a
                        ][1])
                    )
121                 );
            }

            for(int a=0;a<python::len(_layerNames);a++)
            {
                layerInfo.layerNames.push_back(
                    boost::python::extract<string>(_layerNames[a])
                    );
            }

131         for(int a=0;a<python::len(_layerAdjacencyList);a++)
            {
                layerInfo.layerAdjacencyList.push_back(
                    std::pair<string,string>(
                    boost::python::extract<string>((
                        _layerAdjacencyList)[a][0]),
                    boost::python::extract<string>((
                        _layerAdjacencyList)[a][1])
                    )
                    );
            }

141         vector< bool > layerIsInput = convertListToVector<bool
                >(&_layerIsInput);
            for(int a=0;a<int(layerIsInput.size());a++)
            {
                layerInfo.layerIsInput.push_back(layerIsInput[a]);
            }

            for(int a=0;a<python::len(_layerLocations);a++)
            {
                layerInfo.layerLocations.push_back(
                    JGTL::Vector3<float>(
151                 boost::python::extract<float>((_layerLocations)
                        [a][0]),
                    boost::python::extract<float>((_layerLocations)
                        [a][1]),
                    boost::python::extract<float>((_layerLocations)
                        [a][2])
                    )
                    );
            }

            layerInfo.normalize = normalize;
            layerInfo.useOldOutputNames = useOldOutputNames;

161         substrate->setLayerInfo(layerInfo);
        }
```

```
        /**
         *
         */
        static void setLayerInfoFromCurrentExperiment(shared_ptr<
            NEAT::LayeredSubstrate<float> > substrate)
        {
            int experimentType = int(NEAT::Globals::getSingleton()
                ->getParameterValue("ExperimentType")+0.001);
            HCUBE::ExperimentRun experimentRun;
171         experimentRun.setupExperiment(experimentType,"");

            substrate->setLayerInfo(
                experimentRun.getExperiment()->getLayerInfo()
                );
        }

        /**
         * Returns the size of a substrate layer.
         */
181     static python::tuple getLayerSize(shared_ptr<NEAT::
            LayeredSubstrate<float> > substrate,int index)
        {
            return python::make_tuple(
                python::object(substrate->getLayerSize(index).x),
                python::object(substrate->getLayerSize(index).y)
                );
        }

        /**
         * Returns the substrate layer position.
191      */
        static python::tuple getLayerLocation(shared_ptr<NEAT::
            LayeredSubstrate<float> > substrate,int index)
        {
            return python::make_tuple(
                python::object(substrate->getLayerLocation(index).x
                    ),
                python::object(substrate->getLayerLocation(index).y
                    ),
                python::object(substrate->getLayerLocation(index).z
                    )
                );
        }

201     /**
         * Converts a tuple to an STL vector of floats.
         *
         * @param tuple The tuple.
         * @return A vector of floats.
         */
        static Vector3<float> tupleToVector3Float(python::tuple
            tuple)
        {
            return Vector3<float>(
                python::extract<float>(tuple[0]),
211             python::extract<float>(tuple[1]),
                python::extract<float>(tuple[2])
            );
        }

        /**
         * Converts a tuple to an STL vector of ints.
```

```
     *
     * @param tuple The tuple.
     * @return A vector of ints.
221  */
    static Vector3<int> tupleToVector3Int(python::tuple tuple)
    {
        return Vector3<int>(
            python::extract<int>(tuple[0]),
            python::extract<int>(tuple[1]),
            python::extract<int>(tuple[2])
        );
    }

231 /**
     * Configure an experiment to run. Call when the experiment
        will be mainly
     * conducted in Python.
     *
     * @param experiment_filename The filename of the
        experiment initial
     *      conditions.
     * @param output_filename The filename of the output file.
     * @return The ExperimentRun object.
     */
    static shared_ptr<HCUBE::ExperimentRun> setupExperiment(
241     const string & experiment_filename,
        const string & output_filename )
    {
        cout << "CONFIGURING EXPERIMENT:" << endl;
        cout << " : Globals : " << experiment_filename << endl;
        cout << " : Output   : " << output_filename << endl;

        NEAT::Globals::init(experiment_filename);
        int experimentType = int(NEAT::Globals::getSingleton()
            ->getParameterValue("ExperimentType")+0.001);

251     cout << "- Loading Experiment: " << experimentType << "
            ..." << flush;
        shared_ptr<HCUBE::ExperimentRun> experimentRun(new
            HCUBE::ExperimentRun());
        experimentRun->setupExperiment(experimentType,
            output_filename);
        cout << "Done" << endl;

        cout << "- Creating population..." << flush;
        experimentRun->createPopulation();
        experimentRun->setCleanup(true);
        cout << "Done" << endl;

261     cout << "- Setup complete" << endl;

        return experimentRun;
    }

    /**
     * Configure an experiment and then run it through the C++
        interface.
     * In this case the experiment will run in C++ and the
        results will
     * be returned to Python.
     *
```

```
271             * @param experiment_filename The filename of the
                    experiment initial
                *       conditions.
                * @param output_filename The filename of the output file.
                * @return The ExperimentRun object.
                */
               static shared_ptr<HCUBE::ExperimentRun>
                   setupAndRunExperiment(
                       const string & experiment_filename,
                       const string & output_filename )
               {
                   // Setup experiment.
281                shared_ptr<HCUBE::ExperimentRun> experimentRun =
                       setupExperiment(experiment_filename,
                       output_filename);

                   // Run experiment.
                   cout << "- Running experiment..." << endl;
                   cout << "=============================================="
                       << endl;
                   experimentRun->start();
                   cout << "=============================================="
                       << endl;
                   cout << "- Experiment finished" << endl;

                   // Finished, now return.
291                return experimentRun;
               }

               /**
                * Returns the type of experiment being run.
                */
               static int getExperimentType()
               {
                   return int(NEAT::Globals::getSingleton()->
                       getParameterValue("ExperimentType")+0.001);
               }
301
               /**
                * Returns the maximum number of generations for the
                    experiment.
                */
               static int getMaximumGenerations()
               {
                   return int(NEAT::Globals::getSingleton()->
                       getParameterValue("MaxGenerations"));
               }
    };

311 /**
     * The following code section will register the bindings from C++
         to Python.
     *
     * Specifying classes with python::no_init will prevent Python from
         creating
     * new objects of that type.
     *
     * Specifying classes with boost::noncopyable will prevent Python
         from copying
     * objects instead of passing references.
     *
     * Both of these should be used to avoid oddities.
```

```
321  */
     BOOST_PYTHON_MODULE(PyHyperNEAT)
     {
             //To prevent instances being created from python, you add
                 boost::python::no_init to the class_ constructor
         python::class_<HCUBE::ExperimentRun , shared_ptr<HCUBE::
             ExperimentRun>,boost::noncopyable >("ExperimentRun",python
             ::no_init)
                     .def("produceNextGeneration", &HCUBE::ExperimentRun
                         ::produceNextGeneration)
                     .def("finishEvaluations", &HCUBE::ExperimentRun::
                         finishEvaluations)
                     .def("preprocessPopulation", &HCUBE::ExperimentRun
                         ::preprocessPopulation)
                     .def("pythonEvaluationSet", &HCUBE::ExperimentRun::
                         pythonEvaluationSet)
             .def("saveBest", &HCUBE::ExperimentRun::saveBest)
331      ;

         python::class_<NEAT::GeneticPopulation , shared_ptr<NEAT::
             GeneticPopulation> >("GeneticPopulation",python::init<>())
                     .def("getIndividual", &NEAT::GeneticPopulation::
                         getIndividual)
                     .def("getGenerationCount", &NEAT::GeneticPopulation
                         ::getGenerationCount)
                     .def("getIndividualCount", &NEAT::GeneticPopulation
                         ::getIndividualCount)
             ;

         python::class_<NEAT::GeneticGeneration , shared_ptr<NEAT::
             GeneticGeneration>,boost::noncopyable >("GeneticGeneration"
             ,python::no_init)
                     .def("getIndividual", &NEAT::GeneticGeneration::
                         getIndividual)
341                  .def("getIndividualCount", &NEAT::GeneticGeneration
                         ::getIndividualCount)
                     .def("cleanup",&NEAT::GeneticGeneration::cleanup)
                     .def("sortByFitness",&NEAT::GeneticGeneration::
                         sortByFitness)
             ;

             python::class_<NEAT::GeneticIndividual , shared_ptr<NEAT::
                 GeneticIndividual>,boost::noncopyable >("
                 GeneticIndividual", python::no_init)
                 .def("spawnFastPhenotypeStack", &NEAT::
                     GeneticIndividual::spawnFastPhenotypeStack<float>)
             .def("getNodesCount", &NEAT::GeneticIndividual::
                 getNodesCount)
             //.def("getNode", &NEAT::GeneticIndividual::getNode)
             .def("getLinksCount", &NEAT::GeneticIndividual::
                 getLinksCount)
351          //.def("getLink", &NEAT::GeneticIndividual::getLink)
             .def("linkExists", &NEAT::GeneticIndividual::linkExists)
             .def("getFitness", &NEAT::GeneticIndividual::getFitness)
             .def("getSpeciesID", &NEAT::GeneticIndividual::getSpeciesID
                 )
             .def("isValid", &NEAT::GeneticIndividual::isValid)
             .def("printIndividual", &NEAT::GeneticIndividual::print)
             .def("reward",&NEAT::GeneticIndividual::reward)
         ;
```

```
        python::class_<std::vector<shared_ptr<NEAT::
            GeneticIndividual> > >("GeneticIndividualVector")
361             .def(python::vector_indexing_suite<std::vector<
                    shared_ptr<NEAT::GeneticIndividual> > >())
        ;

    python::class_<NEAT::FastNetwork<float> , shared_ptr<NEAT::
        FastNetwork<float> > >("FastNetwork",python::init<>())
            .def("reinitialize", &NEAT::FastNetwork<float>::
                reinitialize)
            .def("update", &NEAT::FastNetwork<float>::update)
            .def("updateFixedIterations", &NEAT::FastNetwork<
                float>::updateFixedIterations)
            .def("getValue", &NEAT::FastNetwork<float>::
                getValue)
        .def("setValue", &NEAT::FastNetwork<float>::setValue)
            .def("hasLink", &NEAT::FastNetwork<float>::hasLink)
371         .def("getLinkWeight", &NEAT::FastNetwork<float>::
                getLinkWeight)
    ;

        python::class_<NEAT::LayeredSubstrate<float> , shared_ptr<
            NEAT::LayeredSubstrate<float> > >("LayeredSubstrate",
            python::init<>())
                .def("populateSubstrate", &NEAT::LayeredSubstrate<
                    float>::populateSubstrate)
            .def("setLayerInfo", &PyHyperNEAT::setLayerInfo)
            .def("setLayerInfoFromCurrentExperiment", &PyHyperNEAT
                ::setLayerInfoFromCurrentExperiment)
                .def("getNetwork", &NEAT::LayeredSubstrate<float>::
                    getNetwork, python::return_value_policy<python
                    ::reference_existing_object>())
                .def("getNumLayers", &NEAT::LayeredSubstrate<float
                    >::getNumLayers)
                .def("setValue", &NEAT::LayeredSubstrate<float>::
                    setValue)
381             .def("getLayerSize", &PyHyperNEAT::getLayerSize)
                .def("getLayerLocation", &PyHyperNEAT::
                    getLayerLocation)
                .def("getWeightRGB", &NEAT::LayeredSubstrate<float
                    >::getWeightRGB)
                .def("getActivationRGB", &NEAT::LayeredSubstrate<
                    float>::getActivationRGB)
                .def("dumpWeightsFrom", &NEAT::LayeredSubstrate<
                    float>::dumpWeightsFrom)
                .def("dumpActivationLevels", &NEAT::
                    LayeredSubstrate<float>::dumpActivationLevels)
        ;

        python::class_<HCUBE::ImageExperiment , shared_ptr<HCUBE::
            ImageExperiment>,boost::noncopyable >("ImageExperiment"
            ,python::no_init)
                .def("setReward",&HCUBE::ImageExperiment::setReward
                    )
391     ;

        /*python::class_<HCUBE::EvaluationSet , shared_ptr<HCUBE::
            EvaluationSet>,boost::noncopyable >("EvaluationSet",
            python::no_init)
                .def("runPython", &HCUBE::EvaluationSet::runPython)
                .def("getExperimentObject", &HCUBE::EvaluationSet::
                    getExperimentObject)
```

```
        ;*/

        python::class_<Vector3<int> >("NEAT_Vector3",python::init
            <>())
                .def(python::init<int,int,int>())
        ;
401
    python::class_<NEAT::Globals, boost::noncopyable >("Globals",
        python::no_init)
        .def("getParameterCount", &NEAT::Globals::getParameterCount
            )
        .def("getParameterName", &NEAT::Globals::getParameterName,
            python::return_value_policy<python::
                copy_const_reference>() )
        .def("getParameterValue", &NEAT::Globals::getParameterValue
            )
        .def("setParameterValue", &NEAT::Globals::setParameterValue
            )
    ;

        python::def("load", PyHyperNEAT::load, python::
            return_value_policy<python::manage_new_object>());
411     python::def("initialize", PyHyperNEAT::initialize);
        python::def("cleanup", PyHyperNEAT::cleanup);
    python::def("getGlobalParameters",
        PyHyperNEAT::getGlobalParameters,
        python::return_value_policy<python::
            reference_existing_object>() );
        python::def("tupleToVector3Int", PyHyperNEAT::
            tupleToVector3Int, python::return_value_policy<python::
            return_by_value>());
        python::def("setupExperiment", PyHyperNEAT::setupExperiment
            );
    python::def("setupAndRunExperiment", PyHyperNEAT::
        setupAndRunExperiment);
    python::def("getExperimentType", PyHyperNEAT::getExperimentType
        );
    python::def("getMaximumGenerations", PyHyperNEAT::
        getMaximumGenerations);
421 }
```

8.2 ImageExperiment.h

```
#ifndef IMAGEEXPERIMENT_H_
#define IMAGEEXPERIMENT_H_

#include "Experiments/HCUBE_Experiment.h"

namespace HCUBE
{
        class ImageExperiment : public Experiment
 9      {
                public:
                        ImageExperiment( string _experimentName,
                            int _threadID );
                        virtual ~ImageExperiment() { }

                        virtual NEAT::GeneticPopulation*
                            createInitialPopulation(int
                            populationSize);
```

```
                virtual void processGroup(shared_ptr<NEAT::
                    GeneticGeneration> generation);

                virtual void processIndividualPostHoc(
                    shared_ptr<NEAT::GeneticIndividual>
                    individual);
19
                virtual bool performUserEvaluations()
                {
                        return false;
                }

                virtual inline bool
                    isDisplayGenerationResult()
                {
                        return displayGenerationResult;
                }
29
                virtual inline void
                    setDisplayGenerationResult(bool
                    _displayGenerationResult)
                {
                        displayGenerationResult=
                            _displayGenerationResult;
                }

                virtual inline void
                    toggleDisplayGenerationResult()
                {
                        displayGenerationResult=!
                            displayGenerationResult;
                }
39
                virtual Experiment * clone();

                virtual void resetGenerationData(shared_ptr
                    <NEAT::GeneticGeneration> generation)
                    {}

                virtual void addGenerationData(shared_ptr<
                    NEAT::GeneticGeneration> generation,
                    shared_ptr<NEAT::GeneticIndividual>
                    individual) {}

                void setReward(shared_ptr<NEAT::
                    GeneticIndividual> individual,double
                    reward) {}

            private:
49              double calculate_reward( NEAT::FastNetwork<
                    float> & network );

            protected:
               /* NEAT::LayeredSubstrate<float> substrate;
                shared_ptr<const NEAT::GeneticIndividual>
                    substrateIndividual;
                NEAT::LayeredSubstrateInfo layerInfo;*/

        };
    }
59
```

```
#endif /* IMAGEEXPERIMENT_H_ */
```

8.3 ImageExperiment.cpp

```
#include "HCUBE_Defines.h"
#include "ImageExperiment/ImageExperiment.h"

using namespace NEAT;
namespace HCUBE
{
        ImageExperiment::ImageExperiment(string _experimentName, int
            _threadID) :
        Experiment(_experimentName, _threadID)
    {
        /*layerInfo = NEAT::LayeredSubstrateInfo();

        //Piece input layer (a)
        layerInfo.layerSizes.push_back(JGTL::Vector2<int>(1024,1));
        layerInfo.layerValidSizes.push_back(JGTL::Vector2<int
            >(1024,1));
        layerInfo.layerNames.push_back("Input");
        layerInfo.layerIsInput.push_back(true);
        layerInfo.layerLocations.push_back(JGTL::Vector3<float
            >(0,0,0));

        //OutputLayer (e)
        layerInfo.layerSizes.push_back(JGTL::Vector2<int>(1,1));
        layerInfo.layerValidSizes.push_back(JGTL::Vector2<int>(1,1)
            );
        layerInfo.layerNames.push_back("Output");
        layerInfo.layerIsInput.push_back(false);
        layerInfo.layerLocations.push_back(JGTL::Vector3<float
            >(0,8,0));

        //inputs connect to hidden
        layerInfo.layerAdjacencyList.push_back(std::pair<string,
            string>("Input","Output"));

        layerInfo.normalize = true;
        layerInfo.useOldOutputNames = false;

        substrate = NEAT::LayeredSubstrate<float>();
        substrate.setLayerInfo(layerInfo);*/
    }

        NEAT::GeneticPopulation * ImageExperiment::
            createInitialPopulation(
        int populationSize)
    {
        cout << "Creating Image Experiment initial population..."
            << endl;
        GeneticPopulation *population = new GeneticPopulation();
        vector<GeneticNodeGene> genes;

        genes.push_back(GeneticNodeGene("Bias", "NetworkSensor",
                0, false));
        genes.push_back(GeneticNodeGene("X",    "NetworkSensor",
                0, false));
        genes.push_back(GeneticNodeGene("Y",    "NetworkSensor",
                0, false));
        //genes.push_back(GeneticNodeGene("Gauss","HiddenNode",
                 0.5, false, ACTIVATION_FUNCTION_GAUSSIAN));
```

```
        genes.push_back(GeneticNodeGene("XOUT", "NetworkOutputNode"
            , 1, false, ACTIVATION_FUNCTION_SIGMOID));
        genes.push_back(GeneticNodeGene("YOUT", "NetworkOutputNode"
            , 1, false, ACTIVATION_FUNCTION_SIGMOID));

        for (int a=0;a<populationSize;a++)
        {
            shared_ptr<GeneticIndividual> individual(new
                GeneticIndividual(genes,true,1.0));

            for (int b=0;b<0;b++)
            {
                individual->testMutate();
            }

            population->addIndividual(individual);
        }

        cout << "Finished creating population\n";
        return population;
    }

    double ImageExperiment::calculate_reward( NEAT::FastNetwork<
        float> & network )
    {}

    void ImageExperiment::processGroup(shared_ptr<NEAT::
        GeneticGeneration> generation)
    {
        NEAT::FastNetwork<float> network = group[0]->
            spawnFastPhenotypeStack<float>();

        double reward = calculate_reward( network );
        group[0]->reward(reward);
    }

    void ImageExperiment::processIndividualPostHoc(shared_ptr<NEAT
        ::GeneticIndividual> individual)
    {
        NEAT::FastNetwork<float> network = individual->
            spawnFastPhenotypeStack<float>();

        double reward = calculate_reward( network );
        const double max_reward = 8*8 + 10.0 + 9.0;
        cout << "POST_HOC_ANALYSIS: " << reward << "/" <<
            max_reward << endl;
    }

    void setReward(shared_ptr<NEAT::GeneticIndividual> generation,
        double reward){
        generation->reward(reward);
    }

    Experiment* ImageExperiment::clone()
    {
        ImageExperiment * experiment = new ImageExperiment(*this);

        return experiment;
    }
}
```

8.4 GUIWindow.py

```
from PyQt4.QtGui import *
from PyQt4.QtCore import *
from PopulationModel import *
import os.path
import sys
import PyHyperNEAT as neat
from datetime import datetime

class Window(QMainWindow):
    def __init__( self, parent = None ):
        super(Window, self).__init__( parent )

        ### Experiment Configuration
        # ATW: TODO: When we start getting reasonable distortions, there
        # needs to be a method to reload an old population. For now we'll
        # always make a new one.

        self.experiment_data_dir  = "../external/HyperNEAT/NE/HyperNEAT/out/data"
        self.date_specifier = datetime.now().strftime("%y%m%d_%H%M%S")

        # Initialize HyperNEAT.
        neat.initialize()

        #see if there is an output directory
        if not os.path.exists(os.getcwd() + "/output"):
            os.mkdir(os.getcwd() + "/output")

        # Load the image experiment.
        self.experiment = neat.setupExperiment(
            "%s/ImageExperiment.dat" % self.experiment_data_dir,
            "output/imageExp_out_%s.xml" % self.date_specifier )

        # Grab the global parameters.
        self.globals = neat.getGlobalParameters()
        population_size = int(self.globals.getParameterValue('PopulationSize'))

        ### GUI Configuration

        # Set default window size.
        self.setFixedSize( 670, 700 )

        # Initialize the list view for the distorted images.
        lv = QListView( )
        lv.setViewMode( QListView.IconMode )
        lv.setUniformItemSizes( True )
        lv.setSelectionRectVisible( True )
        lv.setMovement( QListView.Static )
        lv.setSelectionMode( QListView.MultiSelection )
        lv.setEditTriggers( QListView.CurrentChanged )
        lv.setResizeMode( QListView.Adjust )
        lv.setIconSize( QSize(120, 120) )
        lv.setMinimumSize( 500, 385 )
        lv.setSpacing( 5 )
        lv.setEnabled( False )
        self.population_list = lv

        #pop-up menu
```

```
        lv.setContextMenuPolicy(Qt.CustomContextMenu)
        lv.connect(lv, SIGNAL('customContextMenuRequested (const
            QPoint&)'), self.onContext)

        # Create the population model.
        pm = PopulationModel( population_size )
        self.population_model = pm
64      lv.setModel( pm )
        for i in xrange(population_size):
            self.population_model.update_item( i, DummyNetwork( i %
                4 + 1 ) )

        # Monitor population list selection changes.
        self.connect(
            lv.selectionModel(),
            SIGNAL('selectionChanged(const QItemSelection &, const
                QItemSelection &)'),
            self.handle_listview_change )

74      # Initialize widgets for displaying the graphic and for
            choosing
        # a new base image.
        btn_select_image = QPushButton( "Load Image" )
        self.connect( btn_select_image, SIGNAL('released()'), self.
            select_image )
        lbl_image = QLabel( "Nothing loaded" )
        lbl_image.setFixedSize( 120, 120 )
        self.original_image_label = lbl_image
        image_layout = QVBoxLayout()
        image_layout.addWidget( btn_select_image )
        image_layout.addWidget( lbl_image )
84
        # Initialize a horizontal layout for the parameters.
        gb = QGroupBox( "Evolution Parameters" )
        gb.setSizePolicy( QSizePolicy.Expanding, QSizePolicy.
            Expanding )

        tw = QTableWidget( 1, 2 )
        tw.setHorizontalHeaderLabels(QString("Parameter;Value").
            split(';'))
        tw.setColumnWidth( 0, 350 )
        tw.horizontalHeader().setStretchLastSection( True )
        self.connect( tw, SIGNAL('itemChanged(QTableWidgetItem *)')
            , self.handle_parameter_change )
94      self.parameter_table = tw

        tw_layout = QVBoxLayout()
        tw_layout.addWidget( tw )
        gb.setLayout( tw_layout )

        param_layout = QHBoxLayout()
        param_layout.addLayout( image_layout )
        param_layout.addWidget( gb )

104     # Grab the global parameters.
        parameter_count = self.globals.getParameterCount()
        tw.setRowCount( parameter_count )
        for p in xrange(parameter_count):
            parameter_name = self.globals.getParameterName( p )
            tw.model().setData( tw.model().index( p, 0 ),
                parameter_name )
```

```
                tw.model().setData( tw.model().index( p, 1 ), self.
                    globals.getParameterValue( parameter_name ) )
                index = tw.item( p, 0 )
                index.setFlags( index.flags() ^ Qt.ItemIsEditable )

        # Initialize a horizontal layout for the evolve button.
        btn_evolve = QPushButton( "Evolve" )
        self.connect( btn_evolve, SIGNAL('released()'), self.
            evolve_image )
        btn_evolve.setEnabled( False )
        self.btn_evolve = btn_evolve

        control_layout = QHBoxLayout()
        control_layout.addWidget( btn_evolve )

        # Initialize vertical central layout.
        central_layout = QVBoxLayout()
        central_layout.addLayout( param_layout )
        central_layout.addWidget( lv )
        central_layout.addLayout( control_layout )
        central_widget = QFrame()
        central_widget.setLayout( central_layout )

        self.setCentralWidget( central_widget )

        ### Initialization

        # Handle command line parameters.
        if len(sys.argv) >= 2:
            self.select_image(sys.argv[1])

        # Generate first population.
        self.get_next_generation( initializing=True )

    # Destructor.
    def __del__( self ):
        # Save best population.
        self.experiment.saveBest()

        # Cleanup HyperNEAT.
        neat.cleanup()

    # Update a parameter value.
    def handle_parameter_change( self, parameter ):
        column = parameter.column()
        if column == 1:
            row = parameter.row()
            parameter_name = self.globals.getParameterName( row )
            current_value = self.globals.getParameterValue(
                parameter_name )
            (new_value, is_a_float) = parameter.data(Qt.EditRole).
                toFloat()
            if is_a_float:
                self.globals.setParameterValue( parameter_name,
                    new_value )
            else:
                parameter.setText( "%.2f" % current_value )

    # Choose a new image to work with.
    def select_image( self, file_name = None ):
        if file_name == None:
            file_name = QFileDialog.getOpenFileName(
```

```
                self,
                "Select Image", "",
                "Image Files (*.png *.jpg *.bmp)" );
        if os.path.isfile(file_name):
            print "Loading image: %s ..." % file_name
            self.population_list.setEnabled( True )
            self.original_image = QPixmap( file_name )
            scaled_image = self.original_image.scaled(120, 120, Qt.
                KeepAspectRatio, Qt.SmoothTransformation)
            self.population_model.set_original_image( scaled_image
                )
            self.original_image_label.setPixmap( scaled_image )

    # Handles listview changes. We only care if elements are
        selected.
    def handle_listview_change( self, selected, deselected ):
        evolve_btn_enabled = self.population_list.selectionModel().
            hasSelection()
        if self.btn_evolve.isEnabled() != evolve_btn_enabled:
            self.btn_evolve.setEnabled( evolve_btn_enabled )

    # Get next generation.
    def get_next_generation( self, initializing = False ):
        if not initializing:
            self.experiment.produceNextGeneration()
        self.experiment.preprocessPopulation()
        self.population = self.experiment.pythonEvaluationSet()

        # Update population model.
        if self.population.getIndividualCount() != self.
            population_model.rowCount():
            print "WARNING! Discrepency between population size and
                model size! Things might blow up. Wear a hardhat."

        for i in xrange(self.population_model.rowCount()):
            print "Updating network %2d with new network..." % i,
            index = self.population_list.model().index(i, 0)
            self.population_list.selectionModel().select(index,
                QItemSelectionModel.Select)
            individual = self.population.getIndividual(i)
            network = individual.spawnFastPhenotypeStack()
            self.population_model.update_item(i, network)
            self.population_list.selectionModel().select(index,
                QItemSelectionModel.Deselect)
            self.population_list.repaint()
            print "Done"

    # Evolve the image with the selected individuals.
    def evolve_image( self ):
        # Add a reward to all selected elements.
        indices = self.population_list.selectionModel().
            selectedRows()
        for index in indices:
            self.population.getIndividual(index.row()).reward( 100
                )

        # Deselect elements.
        self.population_list.selectionModel().clearSelection()

        # Finish evaluation.
        self.experiment.finishEvaluations()
```

```
            # Get next generation.
            self.get_next_generation()


        def onContext(self,point):
224         # Create a menu
            menu = QMenu("Menu", self)
            save_image = menu.addAction("Save Selected Images")
            save_network = menu.addAction("Save Selected Neural Network"
                )
            # Show the context menu.
            action = menu.exec_(self.population_list.mapToGlobal(point))
            if action == save_image:
                self.population_model.save_image(self.population_list.
                    selectionModel().selectedRows())
            elif action == save_network:
                self.experiment.saveBest()
```

8.5 PopulationModel.py

```
#import PyHyperNEAT as neat
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import cStringIO
import Image
import os
import sys

# Represents a dummy network for testing.
class DummyNetwork:
    # Network distortion types.
    OneToOne = 1
    HFlip    = 2
    VFlip    = 3
    Flip     = 4

    def __init__( self, network_type = OneToOne ):
        self.network_type = network_type
        self.x_in = 0.0
        self.y_in = 0.0

    def reinitialize( self ):
        pass

    def setValue( self, name, value ):
        if name == 'X':
            self.x_in = float(value)
        elif name == 'Y':
            self.y_in = float(value)

    def update( self ):
        pass

    def getValue( self, name ):
        if name == 'XOUT':
            ret_val = self.x_in
            if self.network_type == DummyNetwork.HFlip:
                ret_val = -ret_val
        elif name == 'YOUT':
            ret_val = self.y_in
            if self.network_type == DummyNetwork.VFlip:
                ret_val = -ret_val
```

```
            if self.network_type == DummyNetwork.Flip:
                return -ret_val
46          else:
                return ret_val

    # Represents one element in a popluation.
    class PopulationItem:
        def __init__( self ):
            self.network = None
            self.distorted_image = None
            self.icon = None

56      def update_distortion( self, image, network = None ):
            if network != None:
                self.network = network

            if (image != None) and (self.network != None):
                self.distorted_image = self.distort( image, self.
                    network )
                self.icon = QIcon(self.distorted_image)

        #@profile
        def distort( self, image_map, network ):
66          #return QPixmap( QImage(image_map) )
            # Convert whatever's sent in to the appropriate image
                format.
            image = QImage( image_map )
            image.convertToFormat( QImage.Format_RGB32 )

            # Extract channel data from the image.
            buffer = QBuffer( )
            buffer.open( QBuffer.ReadWrite )
            image.save( buffer, 'PNG' )
            strio = cStringIO.StringIO()
76          strio.write( buffer.data( ) )
            buffer.close( )
            strio.seek( 0 )
            pil_image = Image.open( strio )
            image_chan_i = pil_image.split()
            image_chan = [
                image_chan_i[0].load(),
                image_chan_i[1].load(),
                image_chan_i[2].load()]

86          # Create a new image of the same size.
            distorted_image = QImage( image.width(), image.height(),
                QImage.Format_RGB32 )

            # Determine normalized coordinates.
            x_norm = image.width() / 2.0
            y_norm = image.height() / 2.0

            # For each pixel, run the network with the destination
                point to determine
            # which pixels to blend.
            for y in xrange(image.height()):
96              for x in xrange(image.width()):
                    y_norm_in = (y - y_norm) / y_norm
                    x_norm_in = (x - x_norm) / x_norm

                    # Evaluate network.
```

```
                    network.reinitialize( )
                    network.setValue( 'X', x_norm_in )
                    network.setValue( 'Y', y_norm_in )
                    network.setValue( 'Bias', .5 )
                    network.update( )

                    x_norm_out = network.getValue( 'XOUT' )
                    y_norm_out = network.getValue( 'YOUT' )

                    # Determine pixel coordinates and clamp to the
                        image boundaries.
                    y_out = y_norm_out * y_norm + y_norm
                    x_out = x_norm_out * x_norm + x_norm
                    if y_out < 0.0:
                        y_out = 0.0
                    elif y_out > image.height() - 1:
                        y_out = image.height() - 1
                    if x_out < 0.0:
                        x_out = 0.0
                    elif x_out > image.width() - 1:
                        x_out = image.width() - 1

                    # Determine row and column pixels and weights.
                    x_o1 = int(x_out)
                    x_o2 = x_o1 + 1 if x_o1 < image.width() - 1 else
                        x_o1
                    y_o1 = int(y_out)
                    y_o2 = y_o1 + 1 if y_o1 < image.height() - 1 else
                        y_o1
                    x_w = x_out - x_o1
                    y_w = y_out - y_o1

                    # Combine pixels.
                    p = [0, 0, 0]
                    for i in xrange(3):
                        p1 = int(round(
                            image_chan[i][x_o1, y_o1]*(1-x_w) +
                            image_chan[i][x_o2, y_o1]*(x_w) ))
                        p2 = int(round(
                            image_chan[i][x_o1, y_o2]*(1-x_w) +
                            image_chan[i][x_o2, y_o2]*(x_w) ))
                        p[i] = p1*(1-y_w) + p2*(y_w)

                    # Set value.
                    distorted_image.setPixel(
                        x, y, qRgb(p[0], p[1], p[2]) )

            return QPixmap(distorted_image)

        def get_distorted_image( self ):
            return QVariant() if self.distorted_image == None else self
                .distorted_image

        def get_icon( self ):
            return QVariant() if self.icon == None else self.icon

    # A simple class for handling a population model.
    class PopulationModel(QAbstractListModel):
        def __init__( self, population_size, parent = None ):
            super(PopulationModel, self).__init__( parent )
            self.image_number = 0
            self.population_size = population_size
```

```
        self.original_image = None
        self.population = [PopulationItem() for x in xrange(self.
            population_size)]

    def set_original_image( self, image ):
        self.original_image = image
        for i in xrange(self.population_size):
            self.update_item( i )

    def update_item( self, index, network = None ):
        self.population[index].update_distortion( self.
            original_image, network )

    def rowCount( self, parent = QModelIndex() ):
        return len(self.population)

    def data( self, index, role ):
        if (not index.isValid()) or (index.row() >= self.
            population_size):
            return None

        if role == Qt.DecorationRole:
            return self.population[index.row()].get_icon( )
        elif role == Qt.SizeHintRole:
            return QSize(120, 120)
        else:
            return QVariant()

    def save_image(self, indices):
        image_path = os.getcwd() + "/savedimages/"

        sys.stdout.write("Saving Images...")

        if not os.path.exists(image_path):
            os.mkdir(image_path)

        for index in indices:
            fileloc = image_path + str(self.image_number)
            image = QImage( self.population[index.row()].
                get_distorted_image() )
            image.convertToFormat( QImage.Format_RGB32 )
            image.save(fileloc, 'PNG')
            self.image_number += 1

        sys.stdout.write("Images Saved\n")
```