MANUAL DE USUARIO NEAT C++

Introducción

A través de este manual de usuario se busca que usted sea capaz de usar redes artificiales NEAT.

Antes de comenzar a programar se debe instalar el programa, usted puede seguir nuestras instrucciones de instalación para tener instalado el programa en forma de librería de C++.

Comenzaremos con un resumen didáctico de NEAT, luego se explicará el proceso estándar de programar una red NEAT y posteriormente se mostrará un programa de ejemplo paso a paso.

Una pincelada por NEAT:

Organismos, poblaciones, generaciones y evolución:

Cada red neuronal será tratada en NEAT como un organismo que pertenece a una población, cada población tiene la capacidad de reproducirse y generar una nueva generación de organismos.

Inicialmente se requiere que usted indique cómo debe ser un organismo, esencialmente esto significa identificar la cantidad de entradas y la cantidad de salidas del organismo, fíjese que a diferencia de otras redes neuronales usted no debe elegir la estructura pues NEAT encontrará la mejor.

Ya elegido el organismo neat comenzará con su algoritmo de evolución, la primera etapa consiste únicamente en generar una población inicial a partir del organismo que le ha sido otorgado.

A diferencia de otras redes neuronales NEAT evoluciona con algoritmos llamados neuroevolutivos, algoritmos que imitan la biología, y que **no utilizan herramientas matemáticas**. Particularmente NEAT funciona con un proceso parecido a la ley del más fuerte donde los organismos se reproducen según qué tan buenos son.

Evaluar un organismo es la parte esencial para programar un entrenamiento en NEAT. Esta evaluación debe ser entregada a cada organismo en su variable llamada **fitness**. Una vez que todos los organismos tienen su fitness pueden comenzar a reproducirse, la regla es sencilla entre más grande su fitness más probabilidades de reproducirse tiene ese organismo. Para lograr estructuras sofisticadas en NEAT además de el cruzamiento existe la posibilidad que ocurra una mutación en el hijo. Esa mutación puede corresponder a agregar una neurona, agregar una conexión o cambiar el peso sináptico de una conexión.

Instalación

Instalacion linux (sólo linux)

instalacion completa por terminal de linux:
Si no está instalado git se debe instalar del repositorio oficial de la distribución:
Ubuntu, Debian, Mint:

```
$ sudo apt-get install git
```

Arch:

```
$ sudo pacman -S git
```

Una vez instalado git:

```
$ git clone https://github.com/psigelo/NEAT
```

Con esto se descargará el proyecto en la ruta ./NEAT, Se entra a la carpeta:

```
$ cd ./NEAT
```

Se crean los objetos y se instala la libreŕía:

```
$ make
$ sudo make install
```

Con esto queda instalada la librería neat en su linux, la librearía queda instalada con el nombre neat y los headers son alcanzables como <NEAT>, por lo que para incluir los headers se debe agregar:
archivo.hpp (agregar):

```
#include <NEAT>
```

y en el makefile ó al momento de la compilación agregar:

```
-lneat
```

ejemplo:

```
g++ ejemplo.cpp -o ejecutable -lneat
```

OBSERVAR:
No es estrictamente necesario instalar la librería (el paso "$sudo make install") pero es más cómodo, en caso de no instalarla se deben agregar los objetos que resulten de "$make" en el makefile de su proyecto ya sea a través de enrutamiento o copiarlos a su carpeta de trabajo, en ese caso también es necesario que haga lo mismo con los headers (.hpp) que se encuentran en la carpeta headers.

Instalacion GUI

Requisito: Qt5

Pueden buscarlo en su repositorio de su distribución preferida. Alternativamente pueden compilar el código fuente desde git:

```
http://qt-project.org/wiki/Building_Qt_5_from_Git
```

Se sugiere la instalación de Qt creator por simplicidad el cual en general se encuentra en los repositorios oficiales.

EJEMPLOS

Como ejemplo se tienen 4 benchmarks típicos de las redes neuronales: XOR

Observación: Para probarlos no es necesario la instalación de la librería (en caso que no tuviera permisos de administrador) simplemente es necesario hacer los objetos con "$make" en la carpeta NEAT.

Los ejemplos se encuentran en la carpeta ./NEAT/experiments/ y se pueden compilar y ejecutar con el makefile de la carpeta:

XOR

El benchmark XOR corresponde a un xor (or excluyente), la idea es que la red neuronal aprenda a funcionar como xor.

Primero se debe entrar a la carpeta del experimento xor

```
cd ./NEAT/experiments/
```

compilación:

```
make xor
```

ejecución:

```
make runXor
```

PASO A PASO:

La siguiente secuencia de pasos busca mostrar de forma sencilla la manera de crear y utilizar un organismo dentro de NEAT.

Paso 1: Creación de una red neuronal inicial.

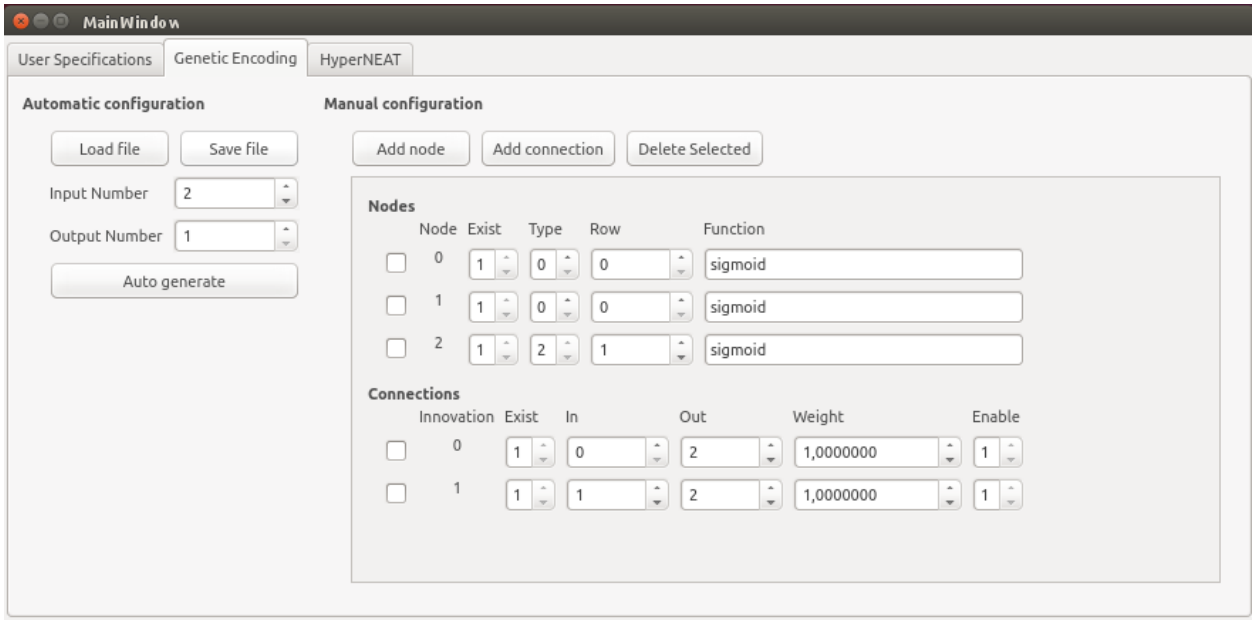
En este documento se entiende que un código genético a una red neuronal.

(1) Mediante GUI

Primero se debe ejecutar el código de la GUI el cual se encuentra en la carpeta Gui. para ejecutarlo puede ejecutar qtcreator, y desde qtcreator abrir el archivo QHyperNEAT.pro que se encuentra dentro de la carpeta QHyperNEAT. Luego apretar el botón run.

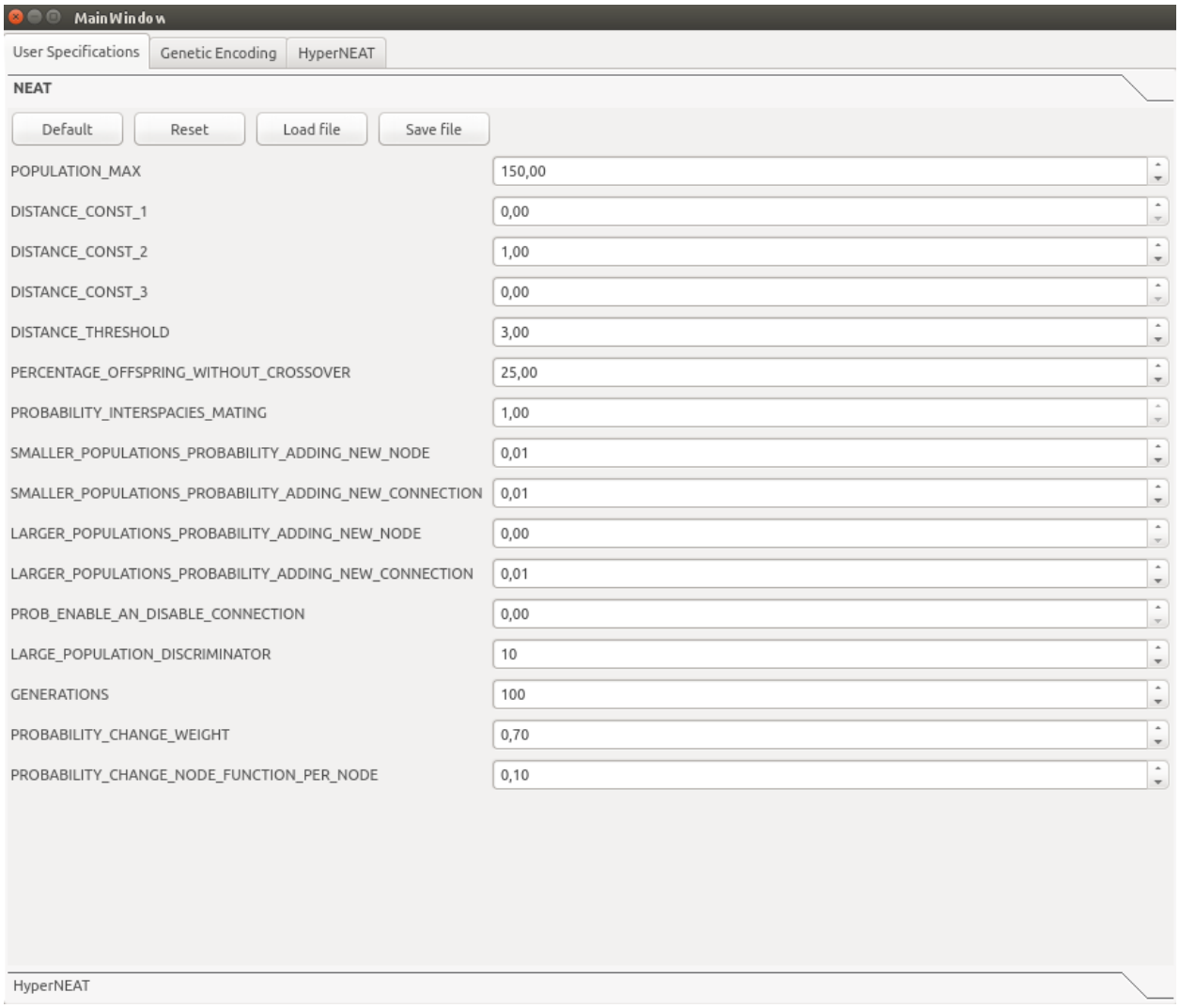
A continuación se expondrán todas las secciones en la que se descompone la interfaz, junto a una breve explicación de cada una.

NEAT: Estructura



Para crear un código genético inicial basta con saber la cantidad de entradas y salidas, luego se aprieta en autogenerar y finalmente en guardar.

NEAT: Especificaciones de usuario



Para cargar o guardar un archivo con los parámetros del usuario, de extensión "*user_definition"*, utilice los botones *'Load'* y *'Save'*, respectivamente.

Los parámetros por defecto son suficientes para la mayoría de los escenarios. Para mayor información sobre algún parámetro puntual refiérase a [1].

Paso 2: Realización del entrenamiento.

Para la realización de un entrenamiento nuevo se tiene un archivo template.cpp en la carpeta Template/ listo para que usted comience a programar, a este archivo le falta únicamente especificar cómo se debe calcular el fitness de los organismos.

El código siguiente corresponde a template.cpp el cual será el punto de partida para realizar un experimento.

```
#include <NEAT>
using namespace ANN_USM;
using namespace std;

double fitness(geneticEncoding organism){
       double result;
       //Este es un ejemplo de como usar a un organismo
       vector <double> inputs;
       vector <double> outputs;

       outputs = organism.eval(inputs);
       return result;
}
int main(int argc, char** argv){
       std::srand(std::time(0));
       Population poblacion(argv[1],argv[2], argv[3] ,argv[4]);
       for (int i = 0; i < poblacion.GENERATIONS; ++i){
              for (int i = 0; i < (int)poblacion.organisms.size(); ++i)
              {
              poblacion.organisms.at(i).fitness = fitness(poblacion.organisms.at(i));
              }
              poblacion.epoch();
       }
       poblacion.saveChampion(argv[3],argv[4]);
       return 0;
}
```

Para realizar un entrenamiento es necesario buscar la manera de evaluar al organismo, o sea asignarle un fitness, para eso hay que solamente programar el método fitness y nada más.

Programando el método fitness:

Primero que nada recordar que usted ya eligió un organismo por lo tanto sabe la cantidad de entradas que éste espera y la cantidad de salidas que éste devolverá, es importante que esto corresponda con el entrenamiento que se realizará.

Para simplificar el entendimiento se explicará paso a paso cómo se realiza un entrenamiento para que una red neuronal aprenda a ser un or excluyente a través del código comentado.

Sólo se mostrará el extracto donde se encuentra el método fitness, el resto del programa sigue siendo el mismo de antes.

```
...
double fitness(geneticEncoding organism){
    /*
                El or exluyente corresponde a un operador de logica
                el cual tiene 2 entradas y una salida

                la logica corresponde a
                0 XOR 0 = 0
                0 XOR 1 = 1
                1 XOR 0 = 1
                1 XOR 1 = 0
    */

        // Se define los vectores de entradas al organismo y salidas del organismo
        vector <double> inputs(2,0);
        vector <double> outputs;
        double error_total(0.0);
        /*
        La idea es medir que tan buen XOR es cada organismo y se medira a partir
        de cuanto error comete en realizar las cuatro operaciones.

        para cada una de las 4 combinaciones de entradas posibles se medira el error
        cometido y luego se calcula el error total del organismo, previamente sabemos
        que el error maximo que puede cometer el organismo es equivocarse en las cuatro
        entradas por lo tanto el error maximo que puede cometer es 4

        finalmente calcularemos el fitness del organismo de la siguiente forma
```

```
        (ERROR MAX - ERROR COMETIDO)^2

        o sea

        (4 - error cometido)^2

        */

        //Procedemos a calcular los errores.

        // 0 XOR 0 = 0
        inputs.at(0) = 0;
        inputs.at(1) = 0;
        outputs = organism.eval(inputs);
        /*Como se espera que el output sea 0 dado que ambas entradas son 0
        y queremos que el maximo error por entrada sea 1
        Ademas sabemos que la salida de las redes neuronales son normalizadas a [-1,1]
        Pero deseamos que sea en el intervalo [0,1] podemos usar el valor
       absoluto a la salida*/
        error_total += abs(0 - abs(outputs.at(0)));
        // 0 XOR 1 = 1
        inputs.at(0) = 0;
        inputs.at(1) = 1;
        outputs = organism.eval(inputs);
        //Como se espera que el output sea 1
        error_total += abs(1 - abs(outputs.at(0)));
        // 1 XOR 0 = 1
        inputs.at(0) = 1;
        inputs.at(1) = 0;
        outputs = organism.eval(inputs);
        //Como se espera que el output sea 1
        error_total += abs(1 - abs(outputs.at(0)));
        // 1 XOR 1 = 0
        inputs.at(0) = 1;
        inputs.at(1) = 0;
        outputs = organism.eval(inputs);
        //Como se espera que el output sea 0
        error_total += abs(0 - abs(outputs.at(0)));

        double fitness = pow(4.0-error_total,2);
        return fitness;
}
...
```

Algunas observaciones:

geneticEncoding corresponde a la clase que representa organismos, el nombre genetic encoding es introducido en [1] y corresponde a la codificación de un organismo, por eso en NEAT geneticEncoding es la clase que representa organismos, particularmente tiene el método std::vector <double> geneticEncoding.eval() el cual tiene que argumento de entrada un vector que corresponde a las entradas del organismo y retorna un vector con las salidas del organismo.

Con el fitness listo queda totalmente programado el entrenamiento, es de esperar que NEAT automáticamente encuentre un organismo campeón quien será el que mejor fitness después de todas las generaciones.

Ahora es ideal guardar al organismo campeón para posteriormente poder usarlo en cualquier código que se desee usar esta red neuronal. Para eso se debe revisar minuciosamente el método main.

```
int main(int argc, char** argv){
       /*
        srand es la semilla para que las probabilidades
        Recordar que sin esta semilla cada vez que se realiza
        una ejecución de NEAT resultaría exactamente los mismos
        organismos
       */
        std::srand(std::time(0));
        /*
        Se crea la poblacion de organismos inicial con el contructor de Population
        argv[1]: Es el archivo de definiciones de NEAT con la definicion de
        todas las probabilidades que gobiernan al algoritmo.
        argv[2]:genetic_encoding es el archivo que define al organismo base del
        entrenamiento
        argv[3]: Es el nombre de NEAT para que posteriormente al guardar
        al campeon este posea un nombre en el disco duro.
        argv[4]: es la ruta en que se desea guardar al campeon en el
        disco duro.
        */
        Population poblacion(argv[1],argv[2], argv[3] ,argv[4]);
        /*
        El siguiente bloque for anidado corresponde a :
        primero: evaluar cada organismo
        segundo: reproducir a los mejores y que pase la epoca
        entiendase como que pase a la siguiente generacion
        */
        for (int i = 0; i < poblacion.GENERATIONS; ++i){
                for (int i = 0; i < (int)poblacion.organisms.size(); ++i)
                {
                poblacion.organisms.at(i).fitness = fitness(poblacion.organisms.at(i));
```

```
                }
                poblacion.epoch();
        }
        /*
        una vez terminado el entrenamiento se debe guardar el campeon
        */
    poblacion.saveChampion();
    return 0;
}
```

Una vez programado el experimento se procede a compilarlo.

```
$ g++ template.cpp -o ejecutable -lneat
```

Para que este código compile usted debe haber seguido las instrucciones para instalarlo como librería.

Para ejecutarlo

```
$./ejecutable       ruta_al_user_definition       ruta_al_genetic_encoding       nombre_prueba
ruta_para_guardar_campion
```

Una vez terminado el entrenamiento usted puede desear usar al campeón que fue entrenado dentro de su propio programa, para eso usted debe cargar al organismo campeón directamente como se muestra en el siguiente ejemplo de carga de campeón del ejemplo anterior.

```
#include <NEAT>
#include <iostream>
using namespace std;
int main(int argc, char** argv){
        geneticEncoding organismo_campeon;
        /*
        Se debe cargar al campeon desde el disco duro
        argv[1] debe ser la ruta hasta el campeon ya entrenado
        */
        organismo_campeon.load(argv[1]);
        /*
        Una vez cargado el campeon se puede usar el metodo eval()
        igual que en el entrenamiento, por ejemplo como prueba
        podríamos ver qué tan bien realiza la prueba el campeon.
        */
        vector <double> inputs(2,0);
```

```
        cout << "XOR(0,0): " << organism.eval(inputs).at(0) << endl;
        inputs.at(0)=0;
        inputs.at(1)=1;
        cout << "XOR(0,1): " << organism.eval(inputs).at(0) << endl;
        inputs.at(0)=1;
        inputs.at(1)=0;
        cout << "XOR(1,0): " << organism.eval(inputs).at(0) << endl;
        inputs.at(1)=1;
        inputs.at(1)=1;
        cout << "XOR(1,1): " << organism.eval(inputs).at(0) << endl;
        return 0;
}
```

la compilación de este programa de ejemplo:

```
$ g++ programa.cpp -o salida -lneat
```

Y la ejecución:

```
$ ./salida ruta_al_campeon_entrenado
```

Bibliografía

[1] Kenneth O. Stanley and Risto Miikkulainen (2002). "Evolving Neural Networks Through Augmenting Topologies". Journal Evolutionary Computation, Vol. 10, No. 2, pp 99127.