Manual de uso de la librería gml_nn.h

La librería gml_nn.h ayuda al programador especificar y crear redes de neuronas artificiales de tipo perceptrón multicapa MLP con todas las neuronas de una capa conectadas a todas las de la siguiente, que posteriormente pueden ser entrenadas para solucionar variedad de problemas mediante el algoritmo de retropropagación. La librería permite la selección de múltiples funciones de activación, funciones de coste o error y otros métodos de optimización para poder crear el modelo deseado en C que permita ser guardado y cargado en otros códigos. También cuenta con un conjunto de funciones que permiten el visualizado y validación de resultados de la red, aunque no es el principal objetivo.

Esta librería forma parte del trabajo de fin de grado de Álvaro González Méndez, estudiante de ingeniería informática en Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid. El código y la documentación de esta se encuentra en el repositorio de GitHub: https://github.com/itsTwoFive/gml_nn.

Para el uso del código, es necesario usar la librería *matrix.h* que está incluida en el repositorio y para ciertas funcionalidades de visualizado se requiere tener instalado en el equipo *gnuplot*.

Para la lectura de datos de archivos con el formato .csv se da la opción de usar la también implementada *data_handler.h* que también se encuentra en el repositorio de GitHub junto a su manual de uso.

Podemos dividir los métodos contenidos en la cabecera de la librería por su funcionalidad en los diferentes campos:

- Creación y configuración de redes de neuronas artificiales:
  - Creación de la red.
  - Configuración de los parámetros de inicialización de variables.
  - Configuración de los parámetros de entrenamiento.
  - Configuración de la función de error de la red.
  - Configuración del optimizador usado por la red.
  - Configuración de las funciones de activación de las capas.
- Entrenamiento y ajuste de las variables de la red:
  - Entrenamiento simple de una sola época.
  - Entrenamiento de la red en varias épocas.
- Alimentado de la red y calculo del error cometido por esta:
  - Operación de alimentado “feedforward”.
  - Calculo del error cometido respecto a una función de error.
- Operaciones de guardado del estado y cargado de diferentes redes:
  - Guardado en fichero del estado de la red.
  - Cargado de red guardada previamente.
  - Consulta de los parámetros y variables.
- Visualizado de los datos obtenidos por la red:
  - Graficado de datos establecidos como entrada de la red.
  - Graficado de las predicciones hechas por la red.
  - Visualización de graficas de error tras el entrenamiento.
- Funciones Extras para validación y manejado de datos:

- Calculo del acierto de una red.
- Normalización de los datos de entrada.
- Discriminador de clases.

La librería hace uso de varios tipos de estructuras de datos `struct` para trabajar con las redes de neuronas y los datos. El primer tipo es `layer`, que guarda la información respecto a las neuronas contenidas en una capa de la siguiente forma:

```
typedef struct {
    int layer_width;
    int act_func;
    double (*c_act_func)(double);
    double alpha_rate;
    matrix *W;
    matrix *oW;
    matrix *dW;
    matrix *vW;
    matrix *cW;
    matrix *out;
} layer;
```

Donde se guarda el valor `layer_width` que indica el número de neuronas que forman dicha capa (ancho de capa). Seguido, existe un entero `act_func` que indica la función de activación que usaran las neuronas de la capa, en caso de que se especifique una función personalizada se usara el puntero a función `c_act_func` para especificar el funcionamiento de esta. Aparte, si la función de activación se puede modular con algún parámetro extra este se almacenará en el número real `alpha_rate`. Finalmente, los punteros al tipo `matrix` definido en *matrix.h* de distintas matrices que contienen los valores de los pesos `W`, valores calculados en pasos intermedios como `oW`, `dW`, `vW`, y `cW`, y por último la salida de las neuronas contenidas en la capa `out`.

El tipo definido como `data`:

```
typedef struct{
    int num_cases_train;
    double **train_input;
    double **train_output;
    int num_cases_test;
    double **test_input;
    double **test_output;
} data;
```

Que almacena tanto los datos de entrada como los de salida de los conjuntos de datos de prueba y entrenamiento. El entero `num_cases_train` guarda el número total de casos en el conjunto de entrenamiento. Los punteros `train_input` y `train_output` apuntan respectivamente a las tablas donde se encuentran guardadas las entradas y salidas del conjunto de casos de entrenamiento. De la misma forma para con el conjunto de pruebas, se guardan en el entero `num_cases_test` y los punteros `test_input` y `test_output` el número de casos del conjunto de prueba, las entradas de este y sus salida. Nótese que las “matrices” o “tablas” de datos guardan los valores de números reales en un array de arrays en lugar del uso de la librería *matrix.h* ya que como no se necesita realizar operaciones matriciales y solo se quiere acceder a los datos, se evita el uso con el fin de ahorrar coste computacional.

La estructura más compleja y clave para el funcionamiento de la librería es el tipo `neural_net`, que define y almacena todos los parámetros globales (que se usan por igual en todas las capas)

y la lista de capas. Aunque se pueden modificar estos campos accediendo a ellos como cualquier otra estructura, se recomienda no hacerlo y usar los métodos de la librería que permiten cambiar esos valores. Esta viene dada por la siguiente especificación de tipo:

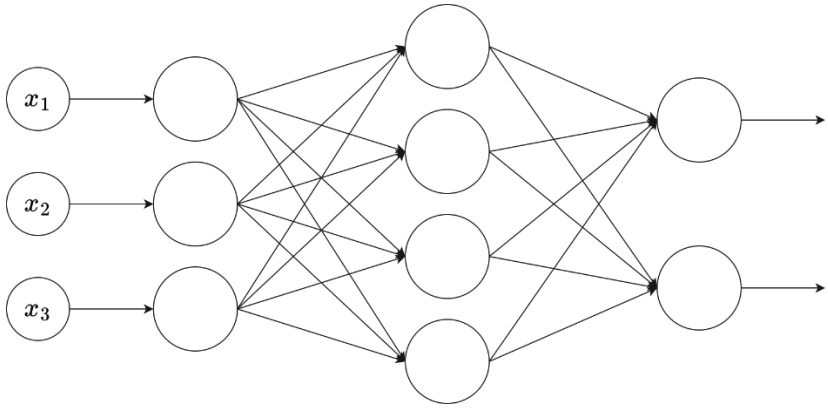
```
typedef struct{
    int input_count;
    int err_func;
    double (*c_err_func)(double,double);
    int layer_count;
    double learning_rate;
    double decay_rate;
    double epsilon_rate;
    int rand_seed;
    int batch_size;
    int cost_output;
    int console_out;
    layer **layers;
    data *dataset;
}neural_net;
```

El entero `input_count` almacena el numero da entradas que la red admite para su alimentación. Al igual que en las capas a la hora de seleccionar función de activación, en la red se almacena el valor que indica la función de error en el entero `err_func`, en caso de que se especifique que se quiere usar una función de error personalizada, el puntero a función `c_err_func` almacenara la dirección de dicha función. El siguiente número entero `layer_count`, indica el número de capas que existen en la red y el puntero de capas `layers`, almacena una lista de punteros a estructuras de tipo `layer` que almacenan la información acerca de las capas del modelo. El "learning rate" o tasa de aprendizaje se guarda en el valor de doble coma flotante `learning_rate`, al igual que la tasa de decadencia en `decay_rate` y el tamaño de salto usado para el método de diferencias finitas en `epsilon_rate`. En caso de que se quiera usar una semilla de generación personalizada para la inicialización de los pesos, esta se debe guardar en el entero `rand_seed`. Si el programador desea usar optimización por mini-lotes, el tamaño del lote se guarda en la variable entera `batch_size`. Las opciones de impresión de coste por consola y visualizado del error son almacenadas en `cost_output` y `console_out`. Por último, los datos necesarios para el entrenamiento y las pruebas cuando se usa la función de entrenamiento simplificado son almacenadas en la estructura de tipo `data` apuntada por `dataset`. El siguiente número entero `layer_count`, indica el número de capas que existen en la red.

Interfaz de uso (listado de métodos)

```
neural_net nn_create(int act_func, int layer_count, int layer_widths[], int
input_count);
```

La rutina **`nn_create`** permite instanciar una red de neuronas que acepta `input_count` número de valores de entrada. Esta contara con un número fijo `layer_count` de capas ocultas y de salida (ya que la capa de entrada se genera dependiendo del número de entradas, no hace falta contar con ella en la inicialización de la red) siendo el tamaño especificado en la cadena de enteros `layer_widths` de forma ordenada. Es importante que el tamaño de la lista de enteros `layer_widths` sea igual al número de capas en la red. El primer parámetro `act_func` determina que función de activación se desea implantar en todas las capas por defecto. De forma gráfica, si se quisiera crear la siguiente red:



Se debería especificar un `input_count` de 3, un `layer_count` de 2 (número de capas que no sean de entrada), y como lista `layer_widths` se entregaría a la función [4,2]. La función devolverá la estructura creada tipo `neural_net`.

```
void layer_print(neural_net nn, int layer_num);
```

El método **`layer_print`** permite mostrar por consola los pesos y el sesgo de cada neurona de la capa número `layer_num` en la red `nn`. La salida del método muestra neurona a neurona por la consola de comandos el valor de los parámetros de esta siguiendo la siguiente estructura:

```
wN.0 0.000000
wN.1 0.000000
      …
wN.M 0.000000
bN   0.000000
```

En el caso de la neurona número N de la capa seleccionada, con $M + 1$ numero de neuronas en la siguiente capa.

```
void nn_set_learning_rate(neural_net *nn, double learning_rate);
```

La función **`nn_set_learning_rate`** permite al programador cambiar el valor de la tasa de aprendizaje aplicada a toda la red de neuronas `nn` al descrito en la llamada `learning_rate`. Por defecto las redes de neuronas generadas con el método `nn_create` cuentan con un valor para la tasa de aprendizaje $\eta = 0.1$.

```
void nn_set_decay_rate(neural_net *nn, double decay_rate);
```

La rutina **`nn_set_decay_rate`** permite al programador cambiar el valor de la tasa de decadencia aplicada a toda la red de neuronas `nn` al descrito en la llamada `decay_rate`. Este valor por defecto viene asociado $\beta = 0.0$ ya que, si se le proporciona un valor positivo, el optimizador de la red de neuronas usara momento para acelerar el descenso de gradiente como es descrito en el método de la bola pesada o descenso con momento. Si se quiere volver a desactivar este, basta con volver a cambiarlo a `0.0f`.

```
void nn_set_epsilon(neural_net *nn, double epsilon_value);
```

El método **`nn_set_epsilon`** permite al programador cambiar el valor épsilon usado en el método de las diferencias finitas para aproximar la derivada de una función. Este valor `epsilon_value` es aplicado a toda la red de neuronas `nn`. Por defecto viene dado como $\varepsilon = 10^{-3}$.

```
void nn_set_batch_size(neural_net *nn, int size);
```

La función **nn_set_batch_size** permite al programador cambiar el modo en el que el optimizador de la red neuronal `nn` trabaja con los casos de entrenamiento durante este. Dependiendo del valor del entero `size`, la red de neuronas se comportará de diferente manera:

- `size` = 0 – La red usara todos los casos de prueba en cada entrenamiento.
- `size` = 1 – La red usara el descenso de gradiente estocástico (SGD) como optimizador, tomando solo una muestra cada entrenamiento.
- `size` > 1 – La red usara mini-lotes seleccionados de forma pseudoaleatoria durante el descenso de gradiente de cada entrenamiento.

Si se establece un valor para `size` mayor que el número de casos en el set de entrenamiento se causara un error por desbordamiento de memoria.

```
void layer_set_act_func(neural_net nn, int layer_pos, int act_func);
```

El método **layer_set_act_func** permite al programador establecer una función de activación denotada por su macro en el campo `act_func` a la capa número `layer_pos` de la red de neuronas `nn`. Existen varios tipos de funciones de activación preprogramadas en la librería, si se quiere usar una de estas, se debe de pasar como parámetro `act_func` una de las siguientes macros:

ACT_NONE – Indica que no se desea emplear ninguna función de activación en la capa.
ACT_SIGMOID – Indica que se desea utilizar la función logística o sigmoidal.
ACT_TANH – Indica que se desea utilizar la función tangente hiperbólica o tanh.
ACT_OPSIGMOID – Indica que se quiere usar una variante optimizada de la función logística para el uso de datos con un rango [-1, 1] de salida esperada. Esta función viene dada por la siguiente expresión:

$$\sigma(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$$

ACT_RELU – Indica que se quiere usar el rectificador ReLU o unidad de rectificador lineal como función de activación.
ACT_LRELU - Indica que se quiere usar el rectificador Leaky ReLU o unidad de rectificador lineal con fuga como función de activación. Esta es una variante del ReLU que toma un valor α como segundo parámetro para modular la parte negativa de la función. Si se quiere modificar este valor que viene por defecto como $\alpha = 0.1$ se debe usar el método `layer_set_alpha`.
ACT_SOFTPLUS – Indica que se quiere usar la función softplus (aproximación suavizada de ReLU).
ACT_HEAVISIDE – Indica que se quiere usar la función escalón. No recomendable su uso ya que al carecer de derivada el entrenamiento falla.

Existe también la macro **ACT_CUSTOM**, que indica que se está usando una función personalizada por el programador. Si se quiere hacer uso de una función personalizada se debe usar el método `layer_custom_act_func`.

```
void layer_set_alpha(neural_net nn, int layer_pos, double alpha);
```

La función **layer_set_alpha** permite establecer el valor α usado como modulador en funciones de activación como Leaky ReLU que necesitan este valor. Por defecto $\alpha = 0.1$, pero si se usa esta función, puede establecerse al valor en punto flotante `aplha` a la capa número `layer_pos` de la red `nn`.

```
void layer_custom_act_func(neural_net nn, int layer_pos, double (*func)(double));
```

La rutina **layer_custom_act_func** permite especificar un puntero `func` a otro método que use como entrada un solo parámetro para que sea usado como función de activación para la capa `layer_pos` de la red de neuronas `nn`. Como no se puede saber analíticamente la derivada de esta, la red usara el método de las diferencias finitas para calcular la derivada de esta usando el valor ε establecido con la función `nn_set_epsilon`. El método de diferencias finitas calcula la derivada de la siguiente forma:

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

A parte, se actualizará el entero que indica la función de activación a `ACT_CUSTOM`.

```
void nn_set_err_func(neural_net *nn, int err_func);
```

El método **nn_set_err_func**, al igual que `layer_set_act_func`, permite al programador establecer una entre las ya predefinidas funciones de error descritas por el entero `err_func` a toda la red de neuronas `nn`. Las funciones de error que el programador puede usar vienen definidas por los siguientes macros:

ERR_SQRDIFF – Es el valor por defecto al crear una red, esta indica que se quiere usar el método de la media de errores cuadráticos.
ERR_HSQRDIFF – Indica que se quiere utilizar la mitad de la media de errores cuadráticos.
ERR_SIMPDIFF – Indica que se quiere usar la media de errores simples. Al igual que la función de activación de escalón, no es recomendable su uso ya que no sirve para entrenar a la red.

Existe también la macro **ERR_CUSTOM**, que indica que se está usando una función personalizada por el programador. Si se quiere hacer uso de una función de error personalizada se debe usar el método `nn_custom_err_func`.

```
void nn_custom_err_func(neural_net *nn, double (*func)(double,double));
```

La rutina **nn_custom_err_func** permite especificar un puntero `func` a otro método que use como entrada de dos parámetros (valor obtenido y esperado) para que sea usado como función de error en toda la red de neuronas `nn`. Como no se puede saber analíticamente la derivada de esta, la red usara el método de las diferencias finitas para calcular la derivada de esta usando el valor ε establecido con la función `nn_set_epsilon`. El método de las diferencias finitas se calcula de igual forma que en el método `layer_custom_act_func`. A parte, se actualizará el entero que indica la función de error a `ERR_CUSTOM`.

```
void nn_set_rand_seed(neural_net *nn, int seed);
```

La rutina **nn_set_rand_seed** permite establecer una semilla `seed` para la inicialización del valor de los pesos de toda la red neuronal nn. Si se establece como valor 0, la semilla se generará en función a la fecha y el tiempo de la maquina donde corra el código.

```
void nn_weight_randf(neural_net *nn);
```

El método **nn_weight_randf** inicializa todos los pesos de la red `nn` de forma pseudoaleatoria, usando el método `rand` de la librería estándar de C. Si se desea cambiar el valor de la semilla de generación, es recomendable hacerlo mediante `nn_set_rand_seed` en vez de usar el método `srand` porque así permite a la hora de guardar la red en un fichero también encapsular este valor.

```
matrix *cost(neural_net nn, int data_length, double **data, double **results);
```

La función **cost** permite calcular el coste usando la función de error establecida en la red nn. Esta necesita como entrada el conjunto de datos de entrada data y los resultados esperados de esta results. También es necesario especificar el tamaño del conjunto.

```
matrix *feed_forward(neural_net nn, double data[], int data_size);
```

La rutina **feed_forward** permite alimentar a la re de neuronas nn con los datos data para un caso específico. Es importante especificar en el campo data_size el número de atributos con los que cuenta el caso con el que estamos alimentando a la red. Tras haber realizado todas las operaciones necesarias, el método devuelve un tipo matrix de tamaño $1 \times N$ siendo N el numero de neuronas en la última capa. También se puede recoger el resultado del último **feed_forward** accediendo al atributo de tipo matrix out de la última capa layer de la red de neuronas nn.

```
void train_network_epoch(neural_net nn, int data_length, double** data, double**
results);
```

El método **train_network_epoch** permite realizar el entrenamiento de una sola época en base a los parámetros y valores establecidos previamente en la red de neuronas nn. Las entradas de la rutina de tipo doble puntero: data y results apuntan directamente al conjunto de datos con los que se quiere entrenar a la red por completo y los resultados esperados de cada uno de los casos. También es necesario especificar el número de casos con el que cuenta el conjunto por el valor data_length.

Este método es el recomendado para un aprendizaje mas controlado y personalizado, ya que el programador puede acceder a cualquier dato entre época y época para establecer diferentes métodos de visualización o control de los resultados.

Si lo que se desea es una forma más sencilla y generalizada de entrenar a dicha red, es recomendable el uso de la función train_network, que permite hacer n número de entrenamientos con una sola llamada y contiene diferentes modos de visualización del estado del entrenamiento.

```
void nn_set_training_data(neural_net nn, int num_cases, double **train_input,
double **train_output);
```

La rutina **nn_set_training_data** permite cargar el conjunto de datos de entrenamiento a la red neuronal nn para su posterior entrenamiento usando el método de entrenamiento simplificado train_network. Es necesario especificar el número de casos en el entero num_cases y por supuesto pasar el conjunto de entradas del set y sus respectivas salidas a través de los dos punteros train_input y train_output respectivamente.

```
void nn_set_testing_data(neural_net nn, int num_cases, double **test_input,
double **test_output);
```

La rutina **nn_set_testing_data**, de la misma forma que nn_set_training_data permite cargar el conjunto de datos de pruebas a la red neuronal nn para su posterior cálculo de coste usando el método de entrenamiento simplificado train_network. Es necesario especificar el número de casos en el entero num_cases y por supuesto pasar el conjunto de entradas del set y sus respectivas salidas a través de los dos punteros test_input y test_output respectivamente.

```
void train_network(neural_net nn, int epochs, int print_cost_each, int
which_cost);
```

El método **train_network** facilita al programador la forma de entrenar a la red de forma simple. Su funcionamiento depende antes del cargado correcto de datos usando `nn_set_training_data` y `nn_set_testing_data`. Se debe especificar el número de épocas o iteraciones en el entero `epochs` y cada cuanto se quiere calcular el coste de la red durante entrenamiento en el entero `print_cost_each`. También se debe especificar con qué conjunto de datos se quiere imprimir el coste. Esto se hace usando una macro como entrada para el valor `which_cost`. Existen los siguientes modos:

COST_NONE – No se realizará ningún calculo de coste durante el entrenamiento.
COST_TRAIN – Solo se calculará el coste de los datos de entrenamiento.
COST_TEST – Solo se calculará el coste de los datos de prueba.
COST_BOTH – Se calculará tanto el coste de los datos de prueba como los de entrenamiento por separado.

El mostrado de los datos por defecto se hace mediante la salida estándar donde se ejecute el código que usa la librería. Haciendo uso del método `nn_set_cost_output` este comportamiento puede ser cambiado, permitiendo la salida de datos en formato csv o un graficado usando *gnuplot*.

Cambiar la salida de los costes no impide que se siga mostrando por consola la información sobre estos. Tanto si se quiere silenciar como si solo se quiere mostrar la época en la que se encuentra la red durante el entrenamiento se debe usar el método `nn_set_console_out`.

```
void nn_set_cost_output(neural_net *nn, int cost_out);
```

La rutina **nn_set_cost_output** debe ser usada para indicar a la red de neuronas que se desea mostrar el resultado de los cálculos de coste. Dependiendo del valor entregado al parámetro entero `cost_out` la red de neuronas actuara de forma distinta durante la ejecución del método de entrenado simplificado `train_network`. Puede adquirir los siguientes comportamientos al usar las macros:

COUT_ONLY_CONSOLE – Valor por defecto. Indica que solo se quiere mostrar la información por la salida estándar.
COUT_GNUPLOT – Indica que se desea dibujar una grafica mostrando los costes calculados por época. Si se calculan los de ambos conjuntos, se mostrarán en la misma grafica. Si la salida de la red de neuronas es múltiple, se calcula una media de los costes de todas las salidas para dibujar la media de coste.
COUT_CSV – Indica que se desea guardar los resultados del coste en una tabla de un fichero con formato csv. El nombre del fichero generado será `costs.csv`.

```
void nn_set_console_out(neural_net *nn, int console_out);
```

La función **nn_set_console_out** permite silenciar la salida de los costes por consola cuando se usa la función `train_network` sobre la red `nn`. Existen varios tipos de silenciado y estos se seleccionan dándole valor al entero `console_out` con una de las siguientes macros:

PRT_CONSOLE – Valor por defecto. Muestra todo cálculo de costes por la salida estándar.
PRT_NOCONSOLE – No muestra ningún mensaje de costes por la salida estándar.
PRT_ONLYEPOCH – Solo muestra el número de iteración en la que se encuentra el entrenamiento.

```
void nn_save(neural_net nn, char* name);
```

La rutina **nn_save** permite el guardado de la red apuntada por el puntero `nn` en un fichero llamado `name` son el sufijo .nn. Todas las redes son guardadas en la carpeta "saved" y siguen el siguiente formato:

```
Input Number: 2
Learning Rate: 0.10000000
Decay Rate: 0.00000000
Epsilon: 0.00100000
Batch Size: 0
Error Function: 1
Random Seed: 0
Number of Layers: 1
Layer Widths: 3

*Layer
       Width: 3
       Alpha Value: 0.10000000
       Activation Function: 5
  -Weights
       0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000
```

Este tipo de ficheros permite consultar y editarlos parámetros de la red sin necesidad de ejecutar código. Para cargarlos basta con hacer uso del método `nn_load`. Cabe destacar que si se estaba usando alguna función de activación o de error personalizada habrá que volver a cargarla cuando se vuelva a cargar la red.

```
neural_net nn_load(char* filename);
```

La función **nn_load** como se puede intuir por su nombre carga una red de neuronas previamente guardada mediante el método `nn_save` por otro Código. Esta toma una cadena `filename` y busca dentro de la carpeta "saved" si existe alguna red con ese nombre. En caso de que no exista buscara tomando la cadena como una dirección relativa. Si tampoco encuentra el fichero .nn mostrara un error por consola y parara la ejecución del Código. Si el método encuentra la red a cargar en el sistema de ficheros devolverá una instancia `neural_net` con la información de la red de neuronas cargada.

Se debe tener en cuenta que si se estaba usando alguna función de activación o de error personalizada habrá que volver a cargarla cuando se vuelva a cargar la red usando el método correspondiente `nn_custom_err_func` o `layer_custom_act_func`.

```
void plot_2d_data_for_binary(double **data_in, double **data_out, int num_cases,
int num_out, double *ranges);
```

La rutina **plot_2d_data_for_binary** crea un proceso ejecutando *gnuplot* conectado mediante un pipe que muestra datos de un set con dos atributos dibujados con colores distintos dependiendo de la clase a la que pertenezcan. Se deben especificar dos punteros a los datos con sus correspondientes salidas `data_in` y `data_out`. También se debe pasar como parámetro `num_cases` el número de casos que se quiere dibujar y el número de clases a las que pueden pertenecer los distintos casos `num_out`. Finalmente, también se debe entregar a la función un array de reales que especifican los límites de la gráfica `ranges`, siendo para el eje x el mínimo `ranges[0]` y el máximo `ranges[1]` y para el eje y `ranges[2]` y `ranges[3]`.

```
void show_areas_2d_plot(neural_net nn, int num_out, double step, double *ranges);
```

El método **show_areas_2d_plot** al igual que plot_2d_data_for_binary muestra crea un pipe a otro proceso ejecutando *gnuplot* que mostrara un mapa de áreas en las que un clasificador usando una red neuronal clasificaría los puntos que se encuentren en dichas áreas. Se debe especificar la red de neuronas previamente entrenada nn, el número de clases que puede clasificar dicha red num_out y el salto entre punto y punto usado para calcular para las áreas step dentro de un rango ranges, siendo para el eje x el mínimo ranges[0] y el máximo ranges[1] y para el eje y ranges[2] y ranges[3].

```
void minmax_normalization(double **data, int param, int size);
```

La rutina **minmax_normalization** realiza para el atributo en la posición param de todos los casos en el conjunto data de tamaño size una normalización de tipo Min Max siguiendo la siguiente formula:

$$minmax(X) = \frac{2(X - \min(X))}{\max(X) - \min(X)} - 1$$

Así estableciendo para todos los casos del conjunto en el atributo seleccionado un valor normalizado entre -1 y 1.

```
double single_binary_acurracy_rate(neural_net nn, double **input, int data_size,
double **expected_arr, double dist, int case_num);
```

El método **single_binary_acurracy_rate** provee una forma sencilla de calcular la tasa de acierto de una red de neuronas entrenada nn. Recibe como parámetros de entrada el puntero input, que apunta a los valores de entrada del conjunto de datos deseado para calcular el acierto, el puntero expected_arr, que igualmente apunta a los resultados esperados de dicho conjunto, el entero data_size, que es el número de atributos en un caso del conjunto, case_num que es el número de casos en el conjunto y por último el valor real en punto flotante dist, que es la distancia máxima entre el resultado obtenido y el esperado para considerarse semejantes.

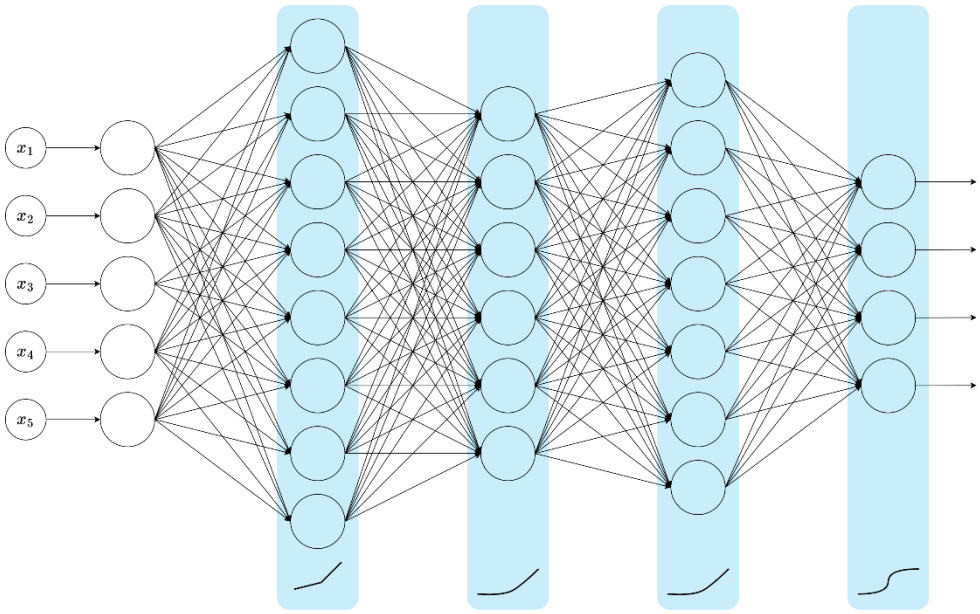
```
int choose_class(double *outputs, int num_out, int target);
```

La rutina **choose_class**, es un método selector de clase cuando se trabaja en la clasificación de mas de dos clases. Recorre los resultados en el array de números reales outputs y escoge la clase que tenga el valor más cercano a un valor objetivo target. Es necesario especificar mediante la variable de entrada num_out el número de clases existentes (número de salidas de la red) a las que puede pertenecer el caso analizado por la red de neuronas nn.

Ejemplos de código

En esta sección se mostrarán varios ejemplos de uso de la librería con el fin de mostrar al programador que vaya a hacer uso de este código de ejemplo por el cual poder guiarse.

Programa 1. Creación de la siguiente red de neuronas con sus respectivas funciones de activación para cada capa:




```c
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    // Creacion de la red
    int widths[] = {8,6,7,4};
    neural_net nn =nn_create(ACT_SOFTPLUS,4,widths,5);

    // Inicializacion de los pesos de forma aleatoria
    nn_weight_randf(&nn);

    // Seleccion de funciones de activacion para las capas 1 y 4
    layer_set_act_func(nn,1,ACT_LRELU);
    layer_set_act_func(nn,4,ACT_SIGMOID);

    return 0;
}
```

Programa 2. Creación de una red de neuronas con métodos como funciones de activación y error.

```c
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

double funcion_de_prueba_act(double a){
    return 7.0f/(1.0f+ exp(-a));
}
```

```c
double funcion_de_prueba_err(double a, double b){
    return pow(a-b,4);
}

int main(int argc, char const *argv[])
{
    // Creacion de la red
    int widths[] = {4,2};
    neural_net nn =nn_create(ACT_TANH,2,widths,3);

    // Establece funcion de activacion personalizada
    layer_custom_act_func(nn,1,&funcion_de_prueba_act);
    layer_custom_act_func(nn,2,&funcion_de_prueba_act);

    // Establece funcion de error personalizada
    nn_custom_err_func(&nn,&funcion_de_prueba_err);

    return 0;
}
```

Programa 3. Creación de una red de neuronas con 2 neuronas ocultas y 1 de salida que se entrena para resolver la tabla de verdad de XOR.

```c
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/xor.csv";

    int num_output = 1;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Normalizamos los datos usando Minmax
    for (int i = 0; i < num_input; i++)
    {
        minmax_normalization(data.data_input,i,data.num_case);
    }

    // Para mejor clasificacion cambiamos los 0 de los outputs por -1
    change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);

    // Estalecemos los tamanos de la red
    int lay_count[] = {2,1};

    // Creamos la Red Neuronal
    neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,2);

    // Configuramos la tasa de aprendizaje
    nn_set_learning_rate(&nn,0.3);

    // Dar valor a la semilla del generador de pesos iniciales e inicializar estos pesos
    nn_set_rand_seed(&nn,23241);
    nn_weight_randf(&nn);

    // Agregamos los datos de entrenamiento y prueba a la red
    nn_set_training_data(nn,4,data.data_input,data.data_output);
    // Entrenamos la red
    int epoch = 50;
```

```
    int print_each = 1;
    train_network(nn,epoch,print_each,COST_TRAIN);

    //Podemos calcular la tasa de acierto discriminando si >0 o si <0
    double acurracy = single_binary_acurracy_rate(nn,
        data.data_input,2,data.data_output,.5,4);

    printf("Acurracy: %f\n",acurracy);
    return 0;
}
```

Programa 4. Creación de una red de neuronas con 2 capas ocultas de (tamaño 16 y 8) y 1 neurona en la capa de salida. Tras ello se le entregan datos de entrenamiento y de prueba y se realiza el entrenamiento de 1000 épocas. Después muestra la tasa de acierto

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/Diabetes.csv";

    int num_output = 1;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Normalizamos los datos usando Minmax
    for (int i = 0; i < num_input; i++)
    {
        minmax_normalization(data.data_input,i,data.num_case);
    }

    // Para mejor clasificacion cambiamos los 0 de los outputs por -1
    change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);

    // Dividimos los datos en set de entrenamiento y pruebas
    double div_num = 75.0/100.0;
    parser_result * div_data = data_div(data,(int) data.num_case*div_num);
    parser_result train_data = div_data[0];
    parser_result test_data  = div_data[1];

    // Estalecemos los tamanos de la red
    int lay_count[] = {16,8,1};

    // Creamos la Red Neuronal
    neural_net nn = nn_create(ACT_SOFTPLUS,3,lay_count,num_input);

    // Configuramos tamano del minilote
    nn_set_batch_size(&nn,40);

    // Configuramos la tasa de aprendizaje
    nn_set_learning_rate(&nn,0.01);

    // Dar valor a la semilla del generador de pesos iniciales e inicializar estos pesos
    nn_set_rand_seed(&nn,20201);
    nn_weight_randf(&nn);

    // Agregamos los datos de entrenamiento y prueba a la red
    nn_set_training_data(nn,train_data.num_case,train_data.data_input,train_data.data_outpu
t);
```

```
    nn_set_testing_data(nn,test_data.num_case,test_data.data_input,test_data.data_output);

    // Entrenamos la red
    int epoch = 1000;
    int print_each = 10;
    nn_set_cost_output(&nn,COUT_GNUPLOT);
    train_network(nn,epoch,print_each,COST_BOTH);

    //Podemos calcular la tasa de acierto discriminando si >0 o si <0
    double acurracy = single_binary_acurracy_rate(nn,
        test_data.data_input,num_input,test_data.data_output,1,test_data.num_case);

    printf("Acurracy: %f\n",acurracy);
    return 0;
}
```

Programa 5. Creación de red que clasifica en 4 clases distintas puntos en un espacio de 2 dimensiones

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/4class.csv";

    int num_output = 4;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Para mejor clasificacion cambiamos los 0 de los outputs por -1
    change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);

    // Creamos la Red Neuronal
    int lay_count[] = {12,num_output};
    neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,num_input);

    // Configuramos diversos parametros
    nn_set_decay_rate(&nn,0.0);
    nn_set_learning_rate(&nn,0.01);

    // Inicializacion de los pesos
    nn_weight_randf(&nn);

    // Podemos configurar tambien distintas Funciones de activacion para cada capa
    layer_set_act_func(nn,2,ACT_OPSIGMOID);

    // Entrenamos la red
    int epoch = 2e4;
    int print_each = 1000;
    for (int i = 0; i < epoch; i++)
    {
        train_network_epoch(nn,data.num_case,data.data_input,data.data_output);
        if(i%print_each == 0){
            matrix * act_cost = cost(nn,data.num_case,data.data_input,data.data_output);
            printf("EPOCA %i Coste de entrenamiento: ",i);
            mat_print(*act_cost);
            mat_free(act_cost);
        }
    }
```

```
    //Podemos calcular la tasa de acierto

    plot2DDataForBinary(data.data_input,data.data_output,data.num_case,data.num_out);
    showAreas2DPlot(nn,data.num_out);

    return 0;
}
```

Programa 6 y 7. Creación de una red de neuronas con 2 neuronas ocultas y 1 de salida que se entrena para resolver la tabla de verdad de XOR. Una vez entrenado se guarda en "xor_model.nn" y el segundo programa lo carga y comprueba el resultado obtenido de la entrada [1,1].

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/xor.csv";

    int num_output = 1;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Estalecemos los tamanos de la red
    int lay_count[] = {2,1};

    // Creamos la Red Neuronal
    neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,2);

    // Configuramos la tasa de aprendizaje
    nn_set_learning_rate(&nn,0.3);

    // Dar valor a la semilla del generador de pesos iniciales e inicializar estos pesos
    nn_set_rand_seed(&nn,23241);
    nn_weight_randf(&nn);

    // Agregamos los datos de entrenamiento y prueba a la red
    nn_set_training_data(nn,4,data.data_input,data.data_output);
    // Entrenamos la red
    int epoch = 50;
    int print_each = 1;
    train_network(nn,epoch,print_each,COST_TRAIN);

     //Podemos calcular la tasa de acierto discriminando si >0 o si <0
    double acurracy = single_binary_acurracy_rate(nn,
        data.data_input,2,data.data_output,.5,4);

     nn_save(nn,"xor_model");


    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"
```

```c
int main(void)
{
    // Cargamos la red guardada como xor_model
    neural_net nn = nn_load("xor_model");

    // Cargamos el caso a probar
    double xor_case[] = {1,1};

    // Ejecutamos el feedforward sobre la red con el caso
    matrix *result = feed_forward(nn,xor_case,2);

    // Mostramos por consola el resultado de la alimentacion
    mat_print(*result);

    // Se comprueba que el resultado contenido en la neurona es el esperado
    if(*mat_seek(*result,0,0) < 0.1){
        printf("El feedforward ha sido correcto\n");
    }
    else{
        printf("El feedforward ha sido erroneo\n");
    }
    return 0;
}
```