Escuela Técnica Superior de Ingeniería Informática

Máster en Lógica, Computación e Inteligencia Artificial

TRABAJO FIN DE MÁSTER

Implementación de Red Neuronal Artificial basado en Scikit-learn

AUTOR:

Ariel Batista Valdez

TUTOR:

José Luís Ruiz Reina

JUNIO DE 2019

Introducción

De las redes neuronales artificiales se dicen muchas cosas, ¿pero que son en realidad? Se dice que fueron inspiradas por el funcionamiento del cerebro humano, que las neuronas (biológicas) y su interconexión motivaron a los pioneros de la inteligencia artificial a pensar en cómo implementar un algoritmo similar para dotar a las maquinas la capacidad de aprender y pensar.

Si bien es cierto que esta fue la premisa inicial, sería muy difícil de implementar de una forma artificial una réplica exacta de este sistema, debido a que aun hoy no se sabe cómo funciona el cerebro biológico.

Entonces, ¿que son las redes neuronales artificiales? Son un algoritmo computacional para aprendizaje automático supervisado, que utiliza nodos independientes e interconectados llamados perceptrones, para almacenar los valores aprendidos de los conjuntos de entrenamiento. Son utilizadas para problemas de clasificación y regresión. Su ventaja principal es que pueden incorporar múltiples pesos W para cada valor X permitiendo adaptarse bastante bien a la generalización.

En este trabajo vamos a abordar el algoritmo de las redes neuronales en detalle e implementaremos su funcionamiento en el lenguaje de programación C#, explicando cada uno de sus componentes y de cómo interactúan entre sí.

Objetivo general

El objetivo de este trabajo es estudiar el algoritmo básico de las redes neuronales y de cómo llevar a cabo su implementación, explicar por qué funciona, como funciona y por qué es importante su conocimiento y cuales problemas se pueden resolver con este sistema. Es un trabajo practico, pero se incluirá teorías relevantes para el mismo.

Objetivos específicos

Estudiar el algoritmo de las redes neuronales.

Estudiar los diferentes componentes de las redes neuronales.

Implementar el algoritmo en el lenguaje de programación C#.

Realizar ejemplos de regresión y clasificación con esta implementación.

Comparar los resultados con la implementación con la librearia scikit-learn.

Dar una breve explicación de problemas que se pueden resolver con este algoritmo y por qué.

Capítulo 1. Marco teórico

Perceptrón simple

El perceptrón simple es un modelo lineal de aprendizaje supervisado, sirve para regresión y para clasificación al añadirle la función de activación. Este está limitado en que el set de datos debe ser linealmente separable.

Perceptrón multicapa

Es una combinación de varios perceptrones, en donde existe una hilera de perceptrones en paralelo que conforman una capa y estos a su vez se conectan a otra hilera de perceptrones en paralelo que sería otra capa, por esto el nombre de multicapa, y así la salida de una capa seria la entrada de otra capa, al igual que el perceptrón simple sirve para regresión y clasificación la ventaja de este es que el conjunto de datos puede no ser linealmente separable.

Tanto el perceptrón simple como multicapa funcionan asignado pesos W a las entradas X para obtener una suma ponderada que dé como resultado el valor de salida esperado en cada caso.

La fórmula del perceptrón simple seria:

$Y' = \text{Activación} (\sum W * X)$

donde:

W= vector de pesos

X= vector de datos de entrada

Activación = función de activación

Y’ = valor predicho

Función de activación

La función de activación es una función que toma como entrada la combinación lineal (∑W * X) y devuelve el resultado de dicha función, se utiliza para modificar la salida.

Las funciones de activación usadas en este TFM son:

Ninguna: no aplica ninguna transformación. Activación(x) = x, debe ser usada para casos de regresión

Umbral: devuelve solo dos valores 1 o 0, se puede utilizar para hacer una clasificación binaria. Activación(x) = si x<0 entonces 0 de lo contrario 1.

Relu: omite los valores negativos. Activación(x)= si x<0 entonces 0 de lo contrario x

Sigmoide: función que devuelve la probabilidad de pertenecer a una clase u otra, se utiliza para clasificador binario, pero agrega la probabilidad. Activación(x) = $1/(1 + e^{x})$

Tanh: función que devuelve números comprendido entre -1 y 1. Activación(x)= $(e^{x} – e^{-x}) / (e^{x} + e^{-x})$

Softmax: a diferencia de las demás funciones esta toma como valor de entrada un vector y devuelve un vector del mismo tamaño, con números comprendidos entre 0 y 1, la suma de este vector dará siempre 1, cada uno de los elementos de este vector contendría la probabilidad de pertenecer a una clase, sirve para problemas de clasificación donde el número de clases será mayor o igual a 2.

Activación(Z)= $e^{zj}/\sum^{k}e^{zk}$

Donde:

Z= vector de entrada

e^{zj}= Euler elevado al componente del vector Z

$^{k}e^{zk}$= Sumatoria de Euler elevado a cada componente del vector Z

Función de perdida

Para encontrar los pesos correspondientes del modelo de red neuronal se realiza mediante un proceso conocido como optimización, en este interviene la función de perdida.

Esta función es una comparación de valores que se realiza entre el valor predicho por la red y el valor real o esperado, mientras más difiera este valor mayor seria la perdida, de ahí que la optimización consista en minimizar dicha perdida. A la perdida también se le conoce como error.

Las diferentes funciones de perdidas utilizadas en este trabajo tienen la siguiente firma:

E= perdida(y, y')

Donde:

E= perdida del lote o batch

y= valor esperado

y' valor predicho por la red.

Las funciones de pérdidas son:

MSE: Mean Square Error o Error Cuadrático Medio, la perdida debe ser siempre positiva de lo contrario perdidas negativas compensarían a las perdidas positivas, en esta función se eleva al cuadrado las diferencias de (y, y') y se divide entre el número de ejemplos en el lote(n).

$MSE(y, y') = (y - y')^2/n$

MAE: Mean Absolute Error o Error Absoluto Medio, similar a la anterior, pero en vez de elevar al cuadrado se calcula el valor absoluto de las diferencias de (y, y').

$MAE(y, y') = |y - y'|/n$

MSLE: Mean Squared Logarithmic Error o Error Cuadrado Logarítmico medio similar a la MSE pero agrega el logaritmo, esta es usado cuando no se quiere penalizar cantidades bajas.

$MSLE(y, y') = (\log(1 + y) - \log(1 + y'))^2/n$

Binary CrossEntropy: utilizada para modelos de clasificación de 2 clases donde se espera que y sea un numero comprendido entre 0 y 1, como la diferencia de estos valores máximo valdrán 1 más adelante cuando explique el de retro-propagación se vera la importancia de que la perdida sea más que 1.

$BinaryCrossEntropy(y, y') = (y * \log(y') + (1 - y) * \log(1 - y'))/n$

Multiclass CrossEntropy: similar a la entropía binaria, pero en problemas de clasificación multiclase se utiliza esta variante

$MulticlassCrossEntropy(y, y') = - y * \log(y')$

Algoritmo de retro propagación

Al igual que en modelo de regresión lineal en las redes neuronales se utiliza el descenso por gradiente para encontrar las matrices de pesos W que minimicen la perdida.

Una vez obtenido el error de la capa de salida, es necesario encontrar la matriz deltaW, esta matriz se sumará a W para así obtener la matriz de pesos que se necesita.

Para calcular la matriz deltaW se necesita propagar el error y calcular por cada nodo su error correspondiente, en caso de que la función de activación sea derivable se multiplica también y por último se multiplica por la tasa de aprendizaje.

$DeltaWo = (Eo . Wo^T)* dActivacion * tasa\ de\ aprendizaje$

$Wo = Wo + deltaWo$

Donde:

Wo = matriz de la última capa

Eo= perdida del lote o batch

dActivacion = pendiente de la función de activación

Una vez calculado el error a la capa de salida, se debe proceder a calcular el error a la capa previa y así hasta la última primera capa, de ahí el nombre de retro-propagación porque la optimización comienza desde la última capa hacia la primera.

Para las calcular las capas ocultas seria todo muy parecido, simplemente se necesita calcular el error de dicha capa, así por ejemplo para calcular la matriz DeltaW de la penúltima capa seria:

$DeltaWh = (Eh . Wh^t) * dActivacion * tasa\ de\ aprendizaje$

$Wh = Wh + DeltaWh$

Donde:

$Eh = (Eo . Wo^t)$

Wh = matriz de la penúltima capa oculta

dActivacion = derivada de la función de activación, si tiene.

Para las demás capas solo sería cuestión de calcular dicho error de la misma forma, Eh sería igual al error de la capa previa multiplicado la matriz transpuesta de los pesos de la capa actual.

Optimizadores

La optimización de la red neuronal consiste en encontrar las matrices de pesos W que minimicen el valor de la función de perdida, el algoritmo de la retro propagación sirve a este propósito, pero dentro de este se incluyen diferentes variaciones, los optimizadores,

dependiendo del elegido y de los parámetros suministrados se encontrara el mínimo más o menos mínimo de la función de perdida en más o menos tiempo.

La tasa de aprendizaje, esta es un valor constante, multiplicado por la matriz DeltaW se minimiza los valores de esta y permite cambiar el tamaño del paso dado. Pero que sea un valor constante no es útil pues a medida que se aproxime al mínimo es necesario disminuir el avance, para no pasarse, una de las formas es reducir a la tasa de aprendizaje a ritmo constante otra seria utilizar el optimizador ADAM.

Los optimizadores implementados en este trabajo son:

SGD: Stocastic Gradient Decent o Descenso por gradiente estocástico, en este se multiplica la matriz Grandiente por la tasa de aprendizaje.

DeltaW = Gradiente * Tasa de aprendizaje.

Donde:

Gradiente = $E * W^T * dActivación$

ADAM: Adaptive Moment Estimation o Estimación de momento adaptativo, es un optimizador de que calcula una tasa de aprendizaje variable para iniciar con un valor alto al inicio y decae en cada interacción, este utiliza 2 parámetros que son beta_1 y beta_2 valores que pueden ser modificados, pero en la mayoría de los casos siempre se presentan con los mismos valores beta_1 = 0.9 y beta_2= 0.999.

Por cada Matriz de pesos W de cada capa se calcula 2 matrices de variaciones llamada momentos M y V, estas matrices se inicializan en 0, y son utilizadas a partir de la segunda interacción, adicionalmente se utiliza el numero de la interacción actual.

M = _beta1 * Manterior + (1 - _beta1) * Gradiente;

V = _beta2 * Vanterior + (1 - _beta2) * $Gradiente^2$

Tanto la matriz M como la V se almacenan para poder ser utilizada en la siguiente iteración, por eso M y V actuales serán menor que M y V de la interacción anterior. Con

los valores de M y V calculados se procede a calcular las matrices M’ V’ y la diferencia de M’ entre V’ elevado al cuadrado conformaran la nueva matriz gradiente.

$M' = M / (1 - beta1^{epoch\ +1})$

$M' = M / (1 - beta2^{epoch\ +1})$

Nuevo Gradiente = M'/ V^2

Donde:

Epoch= número de epoch actual, como este comienza en 0 se le añade 1.

Finalmente se actualizan los pesos.

DeltaW = nuevo Gradiente * tasa de aprendizaje.

Capítulo 2. Implementación en el lenguaje C#

Consideraciones

La implementación de la librería de la red neuronal se realizó usando el lenguaje de programación C# y el framework .NET 4.6.1, esta versión solo funciona en Windows, también se utilizó la librería Math Net Numerics para las operaciones con matrices.

https://numerics.mathdotnet.com/

Las variables y nombre de funciones están escritas en el idioma inglés, solo las etiquetas, botones y mensajes que vería el usuario están en idioma español.

Cuando especifique que la variable es una matriz me referiré específicamente al tipo MathNet.Numerics.LinearAlgebra.Matrix<double>

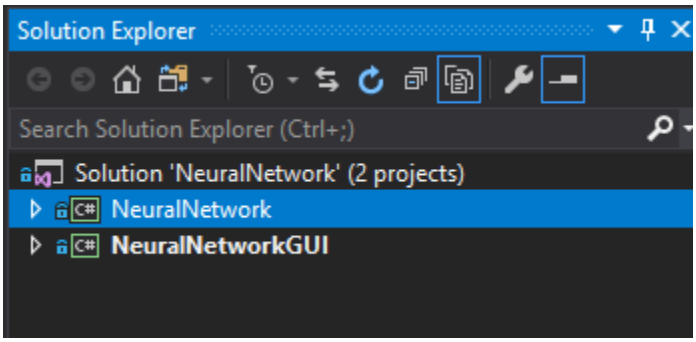
Requisitos de instalación

Sistema operativo Windows

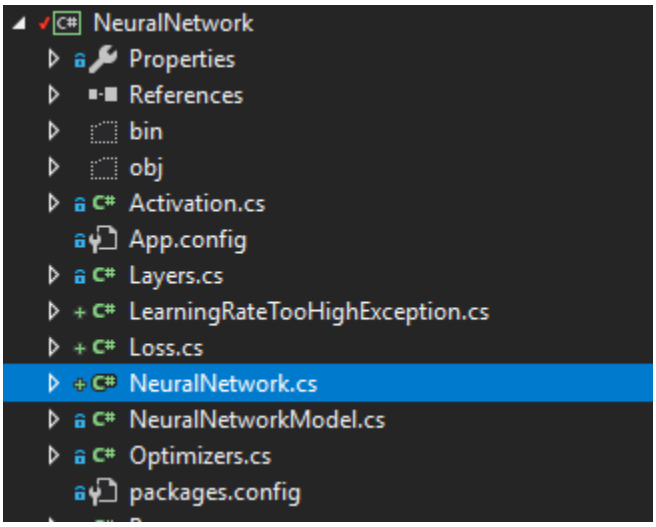
Visual Studio Comunity 2015+

La URL del proyecto en Git es: https://github.com/abatista667/RedNeuronalTFM

Descripción del proyecto



La solución tiene dos proyectos: NeuralNetwork que contiene la librería y NeuralNetworkGUI que contiene un programa de escritorio que permite utilizar la mayoría de las funciones de la librería.



Los archivos del proyecto NeuralNetwork son:

Activation.cs: contiene el código de las funciones de activación

App.config: sin utilidad en este proyecto, creado por defecto por Visual Studio.

Layers.cs: clase que define una capa, contiene los números de nodos y el nombre de la función de activación de esta.

LearningRateTooHighException.cs: define una excepción definida por el usuario, es utilizada cundo las matrices llegan a valores infinitos o extraños, si no es causado por un mal cálculo puede ser que la tasa de aprendizaje sea muy alta.

Loss.cs: contiene las implementaciones de las diferentes funciones de pérdidas y métodos para acceder a ellas.

NeuralNetwork.cs: contiene la implementación de la red neuronal es donde tota la lógica se une.

NeuralNetworkModel.cs: contiene propiedades que sirven para guardar y recrear una red neuronal, configuración y matrices de pesos.

Optimizers.cs: contiene un enum con los nombres de los optimizadores implementados en esta librería.

Package.config: archivo de configuración, contiene los nombres y versiones de las librerías que no son parte de .NET framework, es utilizado para restaurar esas librerías.

La clase NeuralNetwork

Variables privadas

Matrix<double> _inputs: almacena el lote de ejemplos actuales (X).

List<Matrix<double>> _weigths: lista de matrices, almacena una matriz de peso W por cada capa menos 1.

List<Matrix<double>> _bias: lista de matrices, almacena una matriz de bias por cada matriz de peso.

List<Layer> _layers: lista de capas.

List<Matrix<double>> _layerOutput: lista de matrices, almacena la salida de cada capa, del método FeedForward().

Dictionary<int, Matrix<double>> _momentuns: diccionario de matrices, almacena el primer momento de cada matriz de peso, utilizado por optimizador ADAM.

Dictionary<int, Matrix<double>> _2momentuns: diccionario de matrices, almacena el segundo momento de cada matriz de peso, utilizado por optimizador ADAM.

double _learningRate: tasa de aprendizaje.

int _epoch: cantidad de epoch.

Matrix<double> _totalLost: almacena la Perdida del epoch.

Func<Matrix<double>, Matrix<double>, Matrix<double>> _lostFunction: delegado que almacena la función de perdida, todas las funciones de perdidas reciben 2 matrices y retornan una matriz con la perdida.

readonly MatrixBuilder<double> M = Matrix<double>.Build: variable que almacena el instanciador de matrices.

bool _useBias: variable boleana cuando es False no se utiliza el bias en el calculo de la combinación. lineal.

int _batchSize: tamaño del lote o batch.

bool _shufle: variable boleana cuando es False no se reorganizan los ejemplos en cada epoch.

OPTIMIZER _optimizer: enum con el nombre del optimizador a usar.

double _beta1: variable usada por el optimizador ADAM para calcular el primer momento.
double _beta2: variable usada por el optimizador ADAM para calcular el segundo momento.
double epsilon: variable que almacena un numero positivo cercano a 0, usado para sustituir los 0 en el algunos cálculos y evitar multiplicaciones por 0.

private static Random rng = new Random(): variable que permite crear numeros aletorios.

El Constructor de la clase NeuralNetwork

```
public NeuralNetwork(List<Layer> layers, double learningRate = 0.001, int epoch = 100, LOSS lost = LOSS.MSE, bool useBias = true, int batchSize = 200, OPTIMIZER optimizer = OPTIMIZER.SGD, double beta1 = 0.9, double beta2 = 0.999, double epsilon = 1e-8, bool shufle = true)
```

Este es el constructor que sirve para inicializar las variables de la red, como son:

Layers: toma una lista de N capas.

LearningRate: es la tasa de aprendizaje

Epoch: cantidad de epoch.

LOSS: enun con el nombre de la función de perdida.

useBias: valor boleano cuando sea false se omitirán las matrices de bias.

batchSize: tamaño del lote o batch

optimizar: enum con el nombre del optimizador a usar

beta1: valor usado por el optimizador ADAM

beta2: valor usado por el optimizador ADAM

epsilon: valor cercano a 0, utilizado en el optimizador ADAM para evitar multiplicaciones por 0,

shufle: valor boleano, cuando es False no se reodenan los ejemplos en cada epoch.

Adicionalmente se inicializan las matrices de pesos W, 1 por cada capa – 1, debido a que la primera capa no tiene pesos.

También existe otro constructor público, que no inicializa ningún valor, pero este debe ser usado solo para cargar desde un archivo estos valores, usando la función Load.

Métodos públicos de la clase NeuralNetwork

El método Fit

public NeuralNetworkModel Fit(double[][] input, double[][] desiredOutPut)

Este método recibe dos arrays de dobles de dos dimensiones, y los separa en lotes según la variable _batchSize el primero input corresponde a las columnas predictoras (X) mientras que el segundo desiredOutPut corresponde a las columnas a predecir (Y). Este método interacciona por cada epoch y llama al método privado Train para entrenar la red, también reordena el orden los Ejemplos (X, Y)

El método Predict

public double[] Predict(double[] x)

recibe un único ejemplo en forma de array de una dimensión con las columnas predictoras, y retorna las columnas a predecir.

El método Save

public void Save(string path)

Guarda el modelo de red neuronal en la ruta especificada.

El método Load.

public void Load(string path)

Carga el modelo de red neuronal previamente entrenado desde la ruta especificada. Primero se debe inicializar la red con el constructor sin parámetros de la siguiente forma

var nn = new NeuralNetwork();

nn.load(“ruta”);

Métodos privados de la clase NeuralNetwork

El método FeedFordward

```
private Matrix<double> FeedFordward(Matrix<double> input)
```

Recibe como parámetro una matriz con un lote o batch donde cada columna es un ejemplo diferente y cada fila es una columna predictora, realiza la suma ponderada Activacion(∑X*W) de cada capa y retorna la matriz resultado, que sería la predicción de cada ejemplo.

El método Train

```
private double Train(double[][] inputArray, double[][] desiredOutPut, int epoch)
```

Recibe dos array de dos dimensiones uno con varias columnas predictoras y varios ejemplos y el otro con las correspondientes columnas a predecir. Convierte estos arrays de dos dimensiones en matrices, llama al método FeedFordward para obtener la predicción, calcula la perdida, llama al método de optimización seleccionadoy por último retorna la pérdida total del batch.

El método StocasticGradientDecent

```
private void StocasticGradientDecent(Matrix<double> error)
```

Recibe una matriz con el error calculado del lote y efectúa el algoritmo de retropropagación para calcular las matrices deltaW por cada matriz de peso (W), suma estas matrices a los pesos.

El método Adam

```
private void Adam (Matrix<double> error)
```

Similar al método StocasticGradientDecent con la diferencia de que adicionalmente calcula las matrices Momentun1 y momentun2 y hace los cálculos propios del algoritmo ADAM Expuestos en el capítulo anterior.

El método GenerateBatchOrder

```
private List<int> GenerateBatchOrder(int batches)
```

Recibe la cantidad de lotes y genera una lista con cada uno de los índices.

El método ReorderList

```
private void ReorderList(List<int> list)
```

Recibe una lista de enteros y la reordena.

El método SplitInBatches

```
private double[][][] SplitInBatches(double[][] input)
```

Recibe un array de dos dimensiones y lo separa en un array de 3 dimensiones usando la variable _batchSize.

La clase Activation

Esta clase implementa las diferentes funciones de activación

El Método Relu

```
private static double Relu(double x)
{
    return x < 0 ? 0 : x;
}
```

Implementa la función de activación RELU, si el valor de x es menor a 0 devuelve 0 de lo contrario devuelve x.

El método Drelu

```
private static double Drelu(double x)
{
    return 1;
}
```

Devuelve 1 siempre debido a que la función RELU no tiene derivada.

El método None

```
private static double None(double x)
{
    return x;
}
```

Siempre devuelve el valor de x, es decir sin función de activación, esto es debido a que siempre se espera una función de activación.

El método Step

```
private static double Step(double arg)
{
    return arg <= 0 ? 0 : 1;
}
```

Es la función umbral, si x es menor o igual a 0 devuelve 0 de lo contrario devuelve 1.

El método Dstep

```
private static double Dstep(double arg)
{
    return 1;
```

```
}
```

Siempre retorna 1 debido a que la función step o umbral no tiene derivada.

El método Sigmoid

```
private static double Sigmoid(double x)
{
    return 1 / (1 + Math.Pow(Math.E, -x));
}
```

Aplica la función sigmoide a x.

El método Dsigmoid

```
private static double Dsigmoid(double dx)
{
    var val = dx * (1 - dx);

    if (val == 0) return 1;

    return val;
}
```

Aplica la derivada de la función sigmoide, cabe destacar que dx será el valor luego de aplicar la función sigmoide. dx = sigmoid(x).

El método Tanh

```
private static double TanH(double x)
{
    var e = Math.E;
    var numerator = Math.Pow(e, x) - Math.Pow(e, -x);
    var denominator = Math.Pow(e, x) + Math.Pow(e, -x);
    var tanh = numerator / denominator;

    return Math.Tanh(x);
}
```

Aplica la función Tanh a x.

El método Dtanh

```
private static double dTanH(double dx)
{
    return 1 - Math.Pow(dx, 2);
}
```

Aplica la derivada de la función Tanh, cabe destacar que dx será el valor luego de aplicar la función tanh. Dx = tanh(x).

El método Softmax.

```
internal static Matrix<double> Softmax(Matrix<double> output)
{
    var zout = output.Map(x =>
    {
        return Math.Exp(x - output.Enumerate(Zeros.AllowSkip).Maximum());
    });

    var summary = zout.ColumnSums().Sum();

    var softmax = zout.Map(e =>
    {
        return e / summary;
    });

    return softmax;
}
```

Aplica la función Softmax a la matriz de n filas 1 columna.

El método DSoftmax

```
internal static Matrix<double> DSoftmax(Matrix<double> activated)
{
    var builder = Matrix<double>.Build;
    int rc = activated.RowCount;
    var output = builder.Dense(rc, rc);
    for (int i = 0; i < rc; i++)
```

```
        {
            for (int j = 0; j < rc; j++)
            {
                var Si = activated[i, 0];
                var Sj = activated[j, 0];
                if (i == j)
                    output[i, j] = Si * (1 - Sj);
                else
                    output[i, j] = -Sj * Si;
            }
        }

        var o = builder.Dense(output.ColumnCount, 1, output.RowSums().AsArray());


        return o;
    }
```

Recibe una matriz de n filas y 1 columna, con los valores de la función softmax y retorna la derivada de la función softmax.

El método GetActivationByName

internal static Func<double, double> GetActivationByName(ACTIVATION name = ACTIVATION.NONE)

Recibe un enum con el nombre de la función de activación y retorna la implementación de dicha función.

Nota: no funciona para la función softmax, debido a que los tipos no concuerdan.

El método GetActivationDerivativeByName

internal static Func<double, double> GetActivationDerivativeByName(ACTIVATION name = ACTIVATION.NONE)

Recibe un enum con el nombre de la función de activación y retorna la implementación de la derivada de dicha función.

Nota: no funciona para la función softmax, debido a que los tipos no concuerdan.

El diccionario ByName

```
public static Dictionary<string, ACTIVATION> ByName { get; set; } = new
Dictionary<string, ACTIVATION>
        {
            {"", ACTIVATION.NONE},
            {"Ninguno", ACTIVATION.NONE},
            {"Relu", ACTIVATION.RELU},
            {"Sigmoide", ACTIVATION.SIGMOID},
            {"Softmax", ACTIVATION.SOFTMAX},
            {"Tanh", ACTIVATION.TANH},
        };
```

Enlaza el nombre de la función de tipo en string y en español con su correspondiente valor enum.

El enum ACTIVATION

```
public enum ACTIVATION
    {
        SIGMOID,
        RELU,
        TANH,
        SOFTMAX,
        NONE
    }
```

Enum con los diferentes nombres de las funciones de activación.

La clase Loss

Esta clase contiene la implementación de las diferentes funciones de perdidas. Estas reciben una matriz predicción y una matriz con los valores esperados donde las filas son corresponden a las columnas a predecir y las columnas corresponden a los diferentes ejemplos.

El método MSE

private static Matrix<double> MSE(Matrix<double> yhat, Matrix<double> y)

Recibe la predicción y los valores esperados y retorna el error cuadrático medio.

El método MSLE

private static Matrix<double> MSLE(Matrix<double> yhat, Matrix<double> y)

Recibe la predicción y los valores esperados y retorna el error logaritmico cuadrático medio.

El método MAE

private static Matrix<double> MAE(Matrix<double> yhat, Matrix<double> y)

Recibe la predicción y los valores esperados y retorna el error absoluto medio.

El metodo BinaryCrossEntropy

private static Matrix<double> BinaryCrossEntropy (Matrix<double> yhat, Matrix<double> y)

Recibe la predicción y los valores esperados y retorna la entropía cruzada binaria.

El metodo MultiClassCrossEntropy

private static Matrix<double> MultiClassCrossEntropy (Matrix<double> yhat, Matrix<double> y)

Recibe la predicción y los valores esperados y retorna la entropía cruzada multiclase.

El método dLoss

```
private static Matrix<double> dLoss (Matrix<double> yhat, Matrix<double> y)
```

Recibe la predicción y los valores esperados y retorna una matriz de n filas y 1 columna con valores de 1 y -1. Un valor por cada nodo de (y). 1 positivo en caso de que se deba sumar el gradiente de dicho nodo. 1 negativo en caso de que se deba restar el gradiente en dicho nodo. Se debe multiplicar esta matriz por la matriz de perdida, por ejemplo: Perdida = MSE(yhat, y) . dLoss(yhat, y)

El método GetLostFunction

```
public static Func<Matrix<double>, Matrix<double>, Matrix<double>>
GetLostFunction(LOSS name = LOSS.MSE)
```

Recibe un enum con el nombre de la función de perdida y retora su implementación

El enum LOSS

```
public enum LOSS
{
    MSE,
    MSLE,
    MAE,
    BINARY_CROSS_ENTROPY,
    CATEGORICAL_CROSS_ENTROPY
}
```

La clase Layer

```
[Serializable]
public class Layer
{
    public Layer(int nodes, ACTIVATION activation = ACTIVATION.NONE)
    {
```

```
            Nodes = nodes;
            Activation = activation;
        }
        public int Nodes { get; set; }
        public ACTIVATION Activation { get; set; }
    }
```

clase que almacena la configuración de cada capa, cantidad de nodos y el nombre de la función de activación.

La clase LearningRateTooHighException

Es sub-clase de Exception, y es usada para disparar el error de que se ha introducido una tasa de aprendizaje muy alta.

La clase NeuralNetworkModel

```
    [Serializable]
    public class NeuralNetworkModel
    {
        public List<Matrix<double>> Weigths { get; internal set; }

        public List<Layer> Layers { get; internal set; }

        public List<Matrix<double>> Bias { get; internal set; }

        public bool UseBias { get; set; }

        public List<double> Errors { get; internal set; }
    }
```

Contiene las propiedades necesarias para reconstruir una red neuronal previamente guardada.

El enum OPTIMIZER

Contiene los nombres de los diferentes optimizadores implementados en la red.

La clase Optimizers

Esta clase tiene una unica propiedad estatica, el diccionario byName:

```
public static Dictionary<string, OPTIMIZER> ByName { get; set; }
```

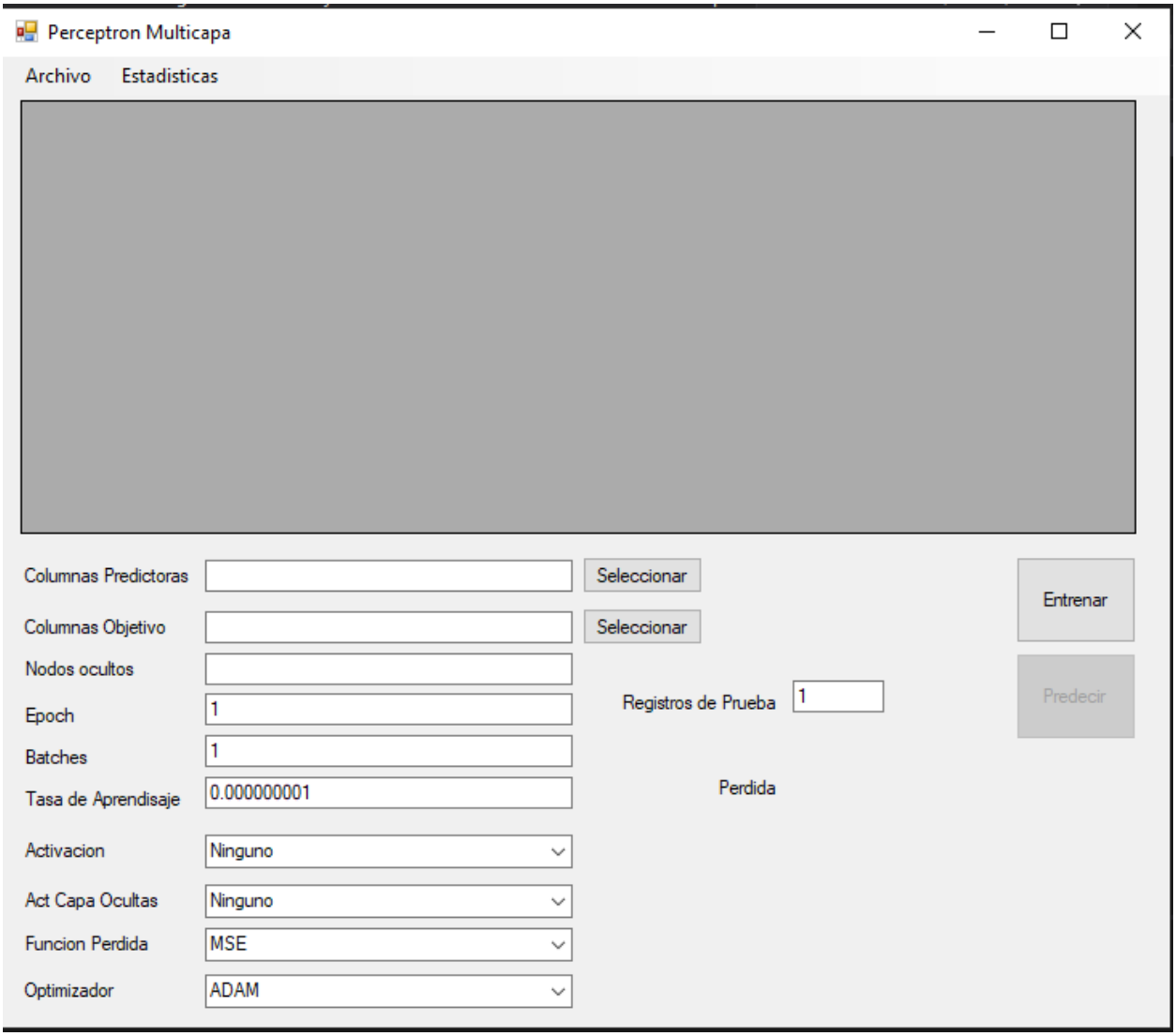
enlaza el valor enum de un optimizador con su correspondiente valor string.

El proyecto NeuralNetworkGUI

Contiene la aplicación de escritorio de Windows que permite utilizar las diferentes funciones del proyecto NeuralNetwork en un entorno grafico amigable.

La ventana principal

Esta es la primera ventana que vemos al compilar y ejecutar la aplicación



Las opciones que nos permite cambiar esta ventana son: seleccionar columnas predictoras, y objetivos, cantidad de capas, y nodos ocultos, cantidad de epoch, tamaño de lotes, tasa de aprendizaje, función de activación de la capa de salida, función de activación de todas las capas ocultas, función de perdida, optimizador, cantidad de registros de prueba.

Botón Entrenar: Entrena la red neuronal con las opciones seleccionadas.

Botón Predecir: despliega una subventana para introducir uno o varios registros con los diferentes valores de las columnas predictoras.

Menú Archivo: Despliega las opciones de cargar un dataset, reiniciar modelo, reiniciar todo, guardar modelo en el disco duro, y salir del programa.

Nota:

La opción reiniciar modelo solo reinicia el modelo de red neuronal.
La opción reiniciar todo borra el dataset cargado en memoria, permitiendo cargar uno nuevo.

Menú Estadísticas: provee de las opciones de verificar la efectividad del modelo, como son: grafico de la función de perdida, grafico de Y vs Ypred, Y vs Ypred detalle.

Capítulo 3. Pruebas y comparación de resultados con scikit-learn

Conclusiones