# Práctica 1. Implementación del perceptrón multicapa

## Introducción a los modelos computacionales

Ingeniería Informática (Rama Computación)

Escuela Politécnica Superior

Universidad de Córdoba

Curso 2016 – 2017

**Autor:** Rubén Medina Zamorano - 46068656R

**Email:** *i32mezar@uco.es*

# ÍNDICE

## 1. INTRODUCCIÓN

Se trata de la primera práctica para la asignatura Introducción a los Modelos Computacionales. En ella, procederemos a elaborar una memoria que describa el problema a tratar, analice el pseudocódigo y el funcionamiento del algoritmo utilizado y por último, obtendremos unos resultados que estudiaremos y compararemos para llegar a unas conclusiones.

## 2. DESCRIPCIÓN DE LOS MODELOS DE REDES NEURONALES

### 2.1. Arquitectura

El modelo que se desarrollará serán redes neuronales basadas en un perceptrón multicapa. Esto quiere decir, que nuestro modelo tendrá unas entradas y mediante las neuronas internas, se obtendrá una salida. Las entradas serán suministradas mediante un fichero de texto, aunque también se podría implementar para introducir los datos manualmente. Y posteriormente, se procesan unas salidas que aproximan a las salidas deseadas en entrenamiento o estiman un valor para generalización al tratarse de un problema de regresión.

Cabe aclarar, que este problema se puede abordar para clasificación con algunas pequeñas modificaciones, de forma que las salidas correspondan a la probabilidad o probabilidades de pertenencia a cierta clase.

El funcionamiento de entrenamiento consiste en propagar las entradas, mediante los pesos de los enlaces y comparar las salidas con el valor objetivo. Y retropropagar el error de la salida, para poder reajustar los pesos en los enlaces. Así, intentar minimizar la función del error. Y por último, comprobar el error para los test cuando se finaliza el entrenamiento.

### 2.2. Organización de capas

La estructura de nuestro modelo se organizará generalmente con una capa de n entradas, una capa oculta con 5 neuronas por defecto y con un único valor de salida. Para el caso de clasificación, la organización sería similar, a diferencia que la salida determinaría la probabilidad de pertenencia a la clase positiva y la probabilidad de la clase negativa sería 1-P(C1).

El número de entradas se establecerán mediante el fichero de texto, al igual que las salidas obtenidas. Y el número de capas ocultas o el número de neuronas se establecerán mediante los parámetros de entrada definidos más adelante. Además, se podrá configurar la existencia de sesgo en estas capas ocultas.

Por otro lado, los pesos de las conexiones entre nodos se establecen en la neurona que tiene la entrada de ese enlace. Es decir, cada neurona de capa oculta o las neuronas de salida (capa h), tienen los vectores de pesos que conectan directamente con las salidas de la capa anterior (capa h-1).

## 3. DESCRIPCIÓN DEL PSEUDOCÓDIGO DEL ALGORITMO DE RETROPROPAGACIÓN

En esta apartado, procederemos a analizar el pseudocódigo y el funcionamiento de las funciones para el algoritmo de retropropagación:

Esta función copia el vector de entradas en la primera capa:

```
void PerceptronMulticapa::alimentarEntradas(double* input) {
   Para i desde  0 hasta pCapas[0].nNumNeuronas
   {
         pCapas[0].pNeuronas[i].x <- input[i];
         pCapas[0].pNeuronas[i].dX <- input[i];
   }
}
```

Esta función propaga las entradas, comprobando el sesgo y los pesos de cada enlace junto con el valor de los nodos en la capa anterior.

```
void PerceptronMulticapa::propagarEntradas() {
   Para i desde 1 hasta nNumCapas //La capa 0 es la entrada
         Para j desde 0 hasta pCapas[i].nNumNeuronas
               int nEntradas <- pCapas[i-1].nNumNeuronas; //Guardo
el numero de neuronas de la capa anterior para recorrer el vector
de pesos
               double exponente <- 0.0;
               Para k desde 0 hasta nEntradas
               {
                     if(k==0 && bSesgo) //Si existe
                           k++;
                     double weight <-  pCapas[i].pNeuronas[j].w[k];
                     double x <- pCapas[i-1].pNeuronas[k].x;
                     exponente += (weight*x);
               }
               pCapas[i].pNeuronas[j].x  <-  (  1/(1+exp(-exponente))
);
}
```

Realiza la retropropagación del error desde la última capa hasta la inicial, tomando como referencia el vector objetivo de salida. Está considerando los valores para dX, pero aún no esta aplicando el cambio en los pesos.

---

```
void PerceptronMulticapa::retropropagarError(double* objetivo) {
   Para i desde 0 hasta pCapas[nNumCapas-1].nNumNeuronas
         //Calcula los dX en primer lugar para la ultima capa de
salida
         double out = pCapas[nNumCapas-1].pNeuronas[i].x;
         pCapas[nNumCapas-1].pNeuronas[i].dX <- - (objetivo[i] -
out )*out*(1-out);

   Para i desde (nNumCapas-2) hasta 0 //La capa 0 es la entrada, y
recorre de final a principio
         Para j desde 0 hasta pCapas[i].nNumNeuronas
         {
               int nEntradas = pCapas[i+1].nNumNeuronas;
               double sum = 0.0;
               for (int k = 0; k < nEntradas; ++k)
               {
               sum                                                 +=
pCapas[i+1].pNeuronas[k].w[j]*pCapas[i+1].pNeuronas[k].dX;
               }
               double out =  pCapas[i].pNeuronas[j].x;
               pCapas[i].pNeuronas[j].dX = sum*out*(1-out);
         }
   }
}
```

---

Esta función acumula los cambios producidos en el valor deltaW para aplicarlo posteriormente al peso de los enlaces

---

```
// Acumular los cambios producidos por un patrón en deltaW
void PerceptronMulticapa::acumularCambio() {
   Para i desde 1 hasta nNumCapas
         Para j desde 0 hasta pCapas[i].nNumNeuronas
               int nEntradas <- pCapas[i-1].nNumNeuronas;
               Para k desde 0 hasta nEntradas
                     Si existe bSesgo && k==0) //Si tiene sesgo, el
primer peso se considera diferente, porque no tiene entrada x.
                           deltaW[k]+= pCapas[i].pNeuronas[j].dX;
                     sino
                           deltaW[k]+=pNeuronas[j].dX   *   pCapas[i-
1].pNeuronas[k].x;
}
```

---

Esta función ajusta los pesos, aplicando los cambios obtenidos en las funciones anteriores de retropropagar el error y acumular el cambio

```
void PerceptronMulticapa::ajustarPesos() {
   Para i desde 1 hasta nNumCapas
      Para j desde 0 hasta pCapas[i].nNumNeuronas
            int nEntradas <- pCapas[i-1].nNumNeuronas;
            Si existe bSesgo
                  nEntradas++;
            Para k desde 0 hasta nEntradas
            Si existe bSesgo y k==0
                  w[k]   +=   (-dEta)*   deltaW[k]   -   dMu   *   (dEta*
   ultimoDeltaW[k]);
            sino
                  w[k]     +=     (-dEta)*deltaW[k]     -     dMu     *
   (dEta*ultimoDeltaW[k]);
   }
```

Esta función simula el funcionamiento de una red neuronal para un patrón de entrada: en el que inicializa los valores, propaga las entradas, comprueba el error, retropropaga el error, y reajusta los pesos.

```
void PerceptronMulticapa::simularRedOnline(double* entrada, double*
objetivo) {
      Para i desde 1 hasta nNumCapas
            Para j desde 0 hasta pCapas[i].nNumNeuronas
                  int nEntradas = pCapas[i-1].nNumNeuronas;
                  Para k desde 0 hasta nEntradas
                        ultimoDeltaW[k] = deltaW[k];
                        pCapas[i].pNeuronas[j].deltaW[k] = 0.0;
      alimentarEntradas(entrada);
      propagarEntradas();
      retropropagarError(objetivo);
      acumularCambio();
      ajustarPesos();
}
```

## 4. EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

### 4.1. Bases de datos utilizadas

Las bases de datos que utilizaremos serán las suministradas en la carpeta /dat, y se dividen en 4 conjuntos, todos con ficheros de train y test:

- **Problema XOR**: esta base de datos representa el problema de clasificación no lineal del XOR. Aproximaríamos una clasificación mediante una regresión, cuyas entradas son -1 y 1 y las salidas son 1 o 0.
- **Función seno**: esta base de datos esta compuesta por 120 patrones de entrenamiento y 41 patrones de test. Ha sido obtenido añadiendo cierto ruido aleatorio a la función seno. Tiene una entrada flotante y una salida flotante.
- **Base de datos CPU**: esta base de datos esta compuesta por 109 patrones de entrenamiento y 100 patrones de test. Las entradas son las características de un conjunto de procesadores y la salida es el rendimiento relativo del procesador. Tiene 6 entradas flotantes y una salida flotante.
- **Base de datos forest**: esta base de datos esta compuesta por 387 patrones de entrenamiento y 130 patrones de test. Las entradas son una serie de datos meteorológicos y otras variables descriptoras sobre incendios en bosques al norte de Portugal y como salida es el área quemada. Tiene algunas entradas binarias y otras flotantes, con salida flotante.

### 4.2. Parámetros de entrada

A la hora de ejecutar nuestro programa, tiene una serie de parámetros de entrada para modificar el aprendizaje o las estructura de nuestra red. Además, estos parámetros cuentan con un valor por defecto:

**- t**: Indica el nombre del fichero que contiene los datos de entrenamiento a utilizar. Sin este argumento, el programa no puede funcionar.

**- T**: Indica el nombre del fichero que contiene los datos de test a utilizar. Si no se especifica este argumento, utilizar los datos de entrenamiento como test.

**- i**: Indica el numero de iteraciones del bucle externo a realizar. Si no se especifica, utilizar 1000 iteraciones.

**- l**: Indica el numero de capas ocultas del modelo de red neuronal. Si no se especifica, utilizar 1 capa oculta.

**- h:** Indica el numero de neuronas a introducir en cada una de las capas ocultas. Si no se especifica, utilizar 5 neuronas.

**- e:** Indica el valor del parámetro eta ($\eta$). Por defecto, utilizar $\eta = 0{,}1$.

**- m**: Indica el valor del parámetro mu ($\mu$). Por defecto, utilizar $\mu = 0{,}9$.

**- b**: Indica si se va a utilizar sesgo en las neuronas. Por defecto, no debemos utilizar sesgo.

### 4.3. Resultados obtenidos

Para las bases de datos dadas y los diferentes parámetros realizaremos una serie de experimentos modificando esos parámetros, y ajustando una mejor estructura para minimizar el error en test y obtener un buen modelo que estime correctamente.

**Errores en CPU**

| Arquitectura de la red | Una capa oculta | | Dos capas ocultas | |
|---|---|---|---|---|
| | Train | Test | Train | Test |
| | (media, DV) | (media, DV) | (media, DV) | (media, DV) |
| **{n:2:k}** | 0.00133, 0.00134 | 0.00447, 0.00084 | 0.00221, 0.0015 | 0.00524, 0.00124 |
| **{n:5:k}** | 0.00070, 1.425-e05 | **0.00350, 0.00052** | 0.00076, 1.2617-e05 | 0.00418, 0.00061 |
| **{n:10:k}** | 0.00065, 1.517-e05 | 0.0036, 0.00068 | 0.00071, 9.957-e06 | 0.00351, 0.00037 |
| **{n:25:k}** | 0.00061, 1.144e-05 | 0.0038, 0.00057 | 0.00066, 1.493-e05 | 0.00358, 0.00015 |
| **{n:50:k}** | 0.00059, 1.473e-05 | 0.00395, 0.00016 | 0.00061, 1.208-e05 | 0.00373, 0.00065 |
| **{n:100:k}** | 0.00061, 1.5578-e05 | 0.00376, 0.00047 | 0.00055, 1.2642e-05 | 0.00476, 0.00075 |

A partir de la anterior tabla, podemos deducir que la estructura **{n:5:k}** tiene el menor error de test, aunque también podría considerarse la estructura {n:10:10:k} y {n:25:25:k} pero esto constituiría un modelo más complejo y menos interpretable. Por tanto, procederemos a modificar los parámetros con la primera estructura.

Sin sesgo, eta=0.1 y mu=0.9

Train → Media = 0.00070 y DV = 1.425-e05

Test → Media = 0.00350 y DV = 0.00052

Con sesgo, eta=0.1 y mu=0.9

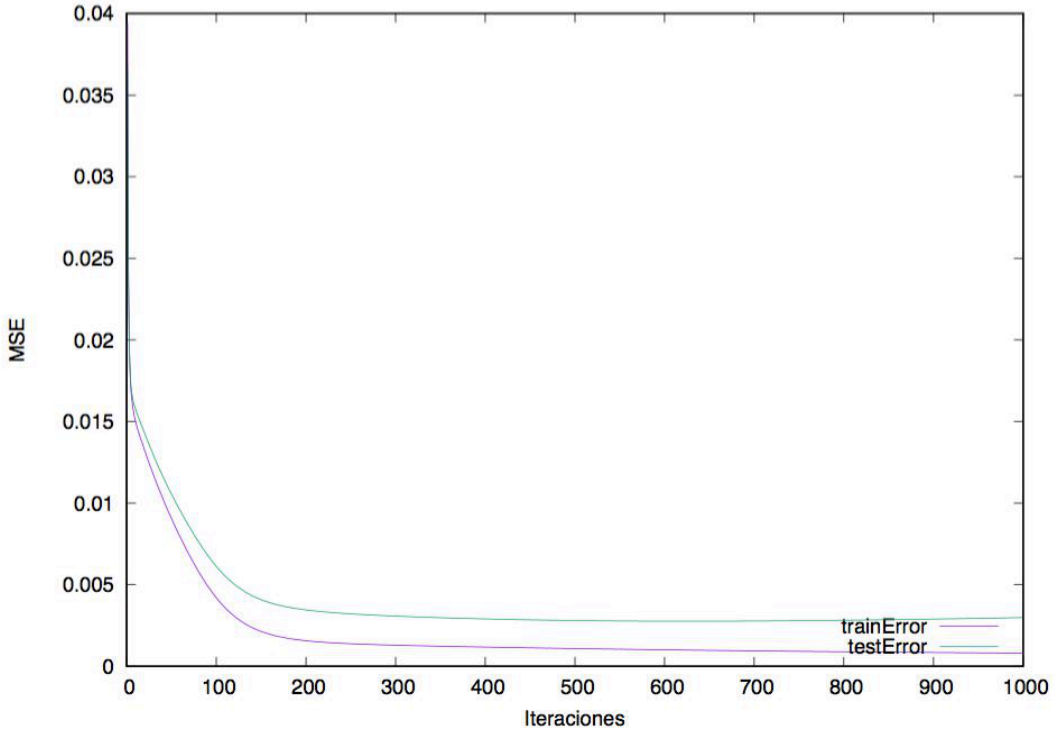
Train → Media = 0.02452 y DV = 0.00295

Test → Media = 0.02707 y DV = 0.00381

El sesgo también dificultaría el modelo y además no minimiza el error, por tanto ajustaremos los parámetros sin sesgo:

| Prueba | Iteraciones | Mu | Eta | Train/Test |
|---|---|---|---|---|
| **1** | 1000 | 0.9 | 0.1 | 0.0007/0.0035 |
| **2** | " | " | 0.5 | 0.00063/0.0036 |
| **3** | " | " | 0.9 | 0.00061/0.00361 |
| **4** | " | 0.5 | 0.1 | 0.00072/0.00346 |
| **5** | " | 0.1 | 0.1 | 0.00075/0.00333 |
| **6** | 500 | 0.1 | 0.1 | 0.00089/0.0032 |
| **7** | 200 | 0.1 | 0.1 | 0.0013/0.0032 |

La **mejor configuración** sería mu y eta a 0.1 y reducir las iteraciones a 500 aproximadamente con un error de test igual a **0.0032**. En la siguiente gráfica podemos ver el sobreentrenamiento, y la necesidad de reducir la iteraciones para minimizar el error en test.




**Errores en SIN**

| Arquitectura de la red | Una capa oculta | | Dos capas ocultas | |
|---|---|---|---|---|
| | Train | Test | Train | Test |
| | (media, DV) | (media, DV) | (media, DV) | (media, DV) |
| **{n:2:k}** | 0.02972, 1.3701e-05 | 0.03644, 0.00017 | 0.02971, 1.691e-05 | 0.03617, 7.1321e-05 |
| **{n:5:k}** | 0.02988, 1.6309e-05 | 0.03612, 2.7483e-05 | 0.02988, 7.4629-e05 | 0.03615, 7.6683e-05 |
| **{n:10:k}** | 0.02994, 0.00068 | 0.03576, 0.00135 | 0.0296, 0.00029 | 0.0363, 0.00014 |
| **{n:25:k}** | 0.02918, 0.00096 | 0.03454, 0.00201 | 0.030259, 2.2614e-05 | 0.03634, 0.00014 |
| **{n:50:k}** | 0.02774, 3.0709e-05 | 0.03379, 0.00011 | 0.02678, 0.00239 | 0.03357, 0.00303 |
| **{n:100:k}** | 0.03154, 6.4196e-05 | 0.04022, 0.00011 | 0.01367, 0.00019 | **0.02029, 0.00036** |

Después de las ejecuciones, vemos que la mejor estructura es **{n:100:**100**:k}** y procederemos a modificar los parámetros de entrada para obtener un mejor resultado.

Sin sesgo, eta=0.1 y mu=0.9

Train → Media = 0.01367 y DV = 0.00019

Test → Media = 0.02029 y DV = 0.00036

Con sesgo, eta=0.1 y mu=0.9

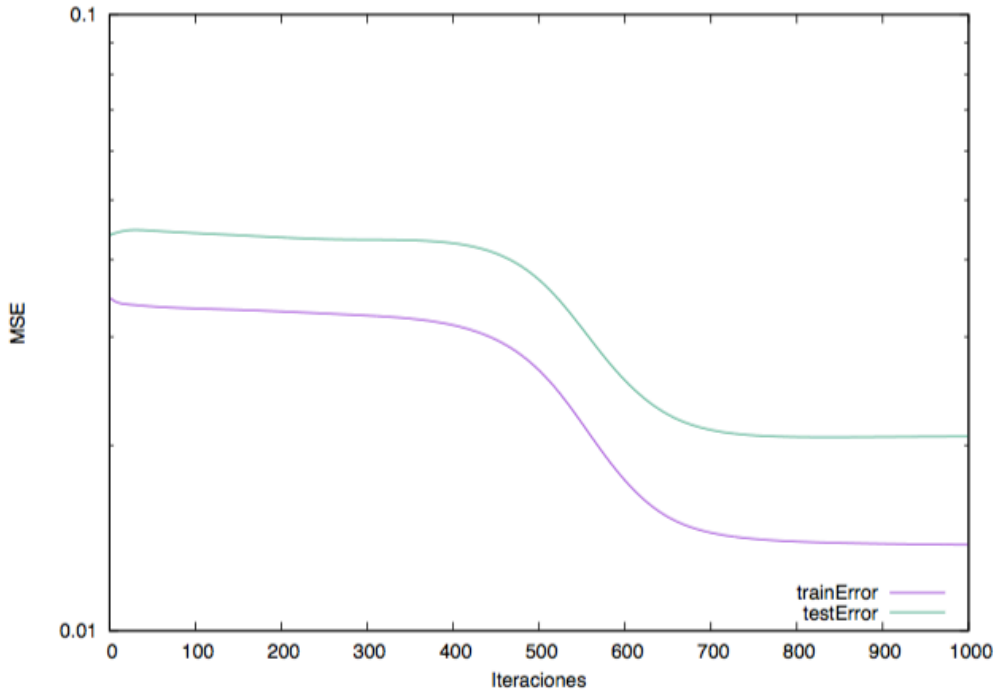
Train → Media = 0.03038 y DV = 0.00069

Test → Media = 0.0383 y DV = 0.0013

Al igual que la base de datos anterior, el sesgo no mejora el modelo, y por tanto consideraríamos una regresión sin sesgo

| Prueba | Iteraciones | Mu | Eta | Train/Test |
|---|---|---|---|---|
| **1** | 1000 | 0.9 | 0.1 | **0.01367 /0.02029** |
| **2** | " | " | 0.5 | 0.01355/0.02015 |
| **3** | " | " | 0.9 | 0.01879/0.02302 |
| **4** | " | 0.5 | 0.1 | 0.0138/0.0206 |
| **5** | " | 0.1 | 0.1 | 0.01783/0.0254 |
| **6** | 500 | 0.9 | 0.1 | 0.02647/0.036 |
| **7** | 200 | 0.9 | 0.1 | 0.03271 / 0.04326 |

Tomaremos el valor señalado como mínimo error con parámetros mu=0.9 y eta=0.1. Representaremos el entrenamiento mediante una gráfica para ver si sobreentrena.



**Errores en XOR**

| Arquitectura de la red | Una capa oculta | | Dos capas ocultas | |
|---|---|---|---|---|
| | Train | Test | Train | Test |
| | (media, DV) | (media, DV) | (media, DV) | (media, DV) |
| **{n:2:k}** | 0.25004, 4.0129e-05 | 0.25004, 4.0129e-05 | 0.24998, 6.9847e-05 | 0.24998, 6.9847e-05 |
| **{n:5:k}** | 0.25002, 7.423e-06 | 0.25002, 7.423e-06 | 0.2182, 0.02416 | 0.2182, 0.02416 |
| **{n:10:k}** | 0.25001, 5.1055e-06 | 0.25001, 5.1055e-06 | 0.10046, 0.08589 | **0.10046, 0.08589** |
| **{n:25:k}** | 0.25001, 4.577e-06 | 0.25001, 4.577e-06 | 0.00376, 0.00118 | 0.00376, 0.00118 |
| **{n:50:k}** | 0.25005, 3.742e-06 | 0.25005, 3.742e-06 | 0.00153, 7.2077e-05 | 0.00153, 7.2077e-05 |
| **{n:100:k}** | 0.25032, 0.00012 | 0.25032, 0.00012 | 0.00086, 0.00024 | 0.00086, 0.00024 |

La arquitectura que obtiene un menor error sería {n:100:100:k} pero en este caso no la escogeremos ya que se trata de un problema simple XOR, y esa estructura es muy compleja y poco interpretable. Por tanto escogeremos {n:10:k} y modificaremos los parámetros para llegar a minimizar igualmente el error en test.

Sin sesgo, eta=0.1 y mu=0.9

Train → Media = 0.25001 y DV = 5.1055e-06

Test → Media = 0.25001 y DV = 5.1055e-06

Con sesgo, eta=0.1 y mu=0.9

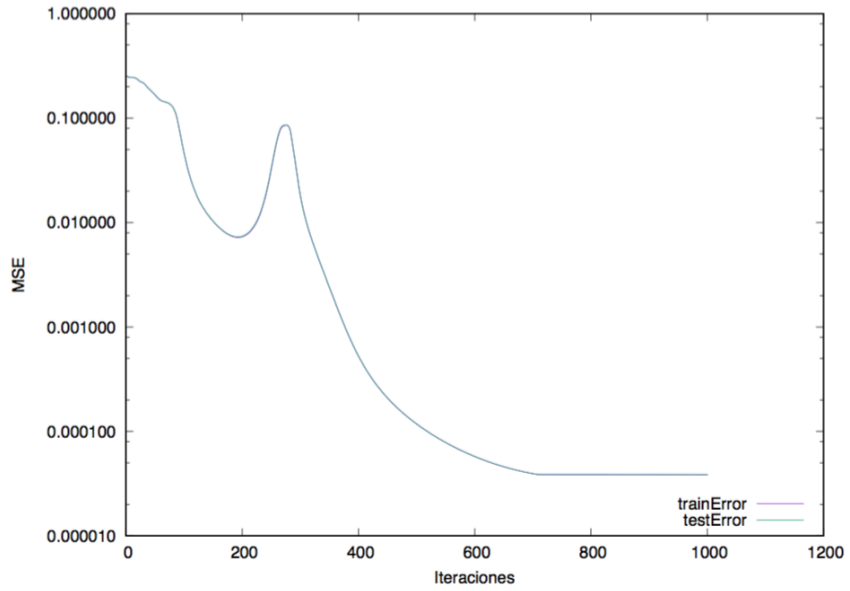
Train → Media = 0.046602 y DV = 0.0544

Test → Media = 0.046602 y DV = 0.0544

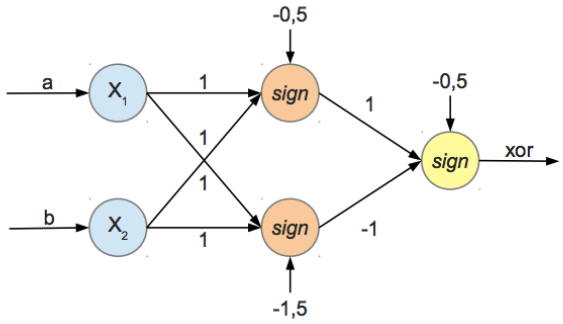
En este caso, es necesario aplicar el sesgo para mejorar considerablemente el modelo y así minimizar el error de test.

| Prueba | Iteraciones | Mu | Eta | Train/Test |
|---|---|---|---|---|
| **1** | 1000 | 0.9 | 0.1 | 0.046602 |
| **2** | " | " | 0.5 | 0.028302 |
| **3** | " | " | 0.9 | 0.00206 |
| **4** | " | 0.5 | 0.9 | 0.04348 |
| **5** | " | 0.1 | 0.9 | 0.13331 |
| **6** | 500 | 0.9 | 0.9 | 0.00729 |
| **7** | 200 | 0.9 | 0.1 | 0.02871 |

La mejor configuración la obtenemos con 1000 iteraciones y los valores 0.9 para los parámetros mu y eta. Quedando un error mínimo final de **0.00206**, lo podemos representar de la siguiente forma.

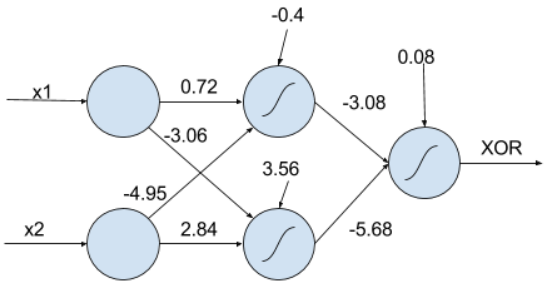



Por otra parte, utilizando la estructura {n:2:k}, vamos a representar el modelo gráficamente para compararlo con el modelo estudiado en clase, ambos con sesgo.




Este modelo estudiado aporta los valores correctos al problema del Xor, sin embargo, nuestro modelo entrenado no llega a suministrar salidas correctas con los mismos patrones y el mismo número de neuronas. Cabe destacar, que el modelo anterior está utilizando neuronas con función signo, al contrario que el inferior con la sigmoide. Por tanto, el mejor modelo después de varias iteraciones es el siguiente, que podría mejorar llegando a trabajar correctamente minimizando el error si se modifica la capa oculta introduciendo más neuronas.

```
Salida Esperada Vs Salida Obtenida (test)
=========================================
1 -- 0.333717
0 -- 0.352674
1 -- 0.88204
0 -- 0.342283
Finalizamos => Error de test final: 0.174846
```



**Errores en Forest**

| Arquitectura de la red | Una capa oculta | | Dos capas ocultas | |
|---|---|---|---|---|
| | Train | Test | Train | Test |
| | (media, DV) | (media, DV) | (media, DV) | (media, DV) |
| **{n:2:k}** | 0.00161, 6.1854-e06 | **0.00881, 5.3838e-05** | 0.0016, 1.685e-06 | 0.00877, 4.032e-05 |
| **{n:5:k}** | 0.00161, 4.5241e-06 | 0.00883, 5.066e-05 | 0.0016, 3.9399e-06 | 0.00878, 2.572e-05 |
| **{n:10:k}** | 0.00161, 1.507e-05 | 0.00884, 6.5901e-05 | 0.0016, 5.865e-06 | 0.00877, 6.2422e-05 |
| **{n:25:k}** | 0.00161, 1.1194e-05 | 0.00889, 0.00010 | 0.0016, 9.9805e-06 | 0.0087, 6.339e-05 |
| **{n:50:k}** | 0.00162, 1.623e-05 | 0.00892, 0.00011 | 0.0016, 1.181e-05 | 0.00885, 9.405e-05 |
| **{n:100:k}** | 0.00163, 5.4466e-06 | 0.00893, 7.604e-05 | 0.00155, 0.00014 | 0.00897, 8.907e-05 |

Como podemos ver en la tabla anterior, el error varía minimamente con la modificación de la arquitectura. Por tanto, para simplificar el modelo, utilizaremos la primera {n:2:k} y {n:5:k} para probar los parámetros y escoger una de las dos.

**{n:2:k}**

Sin sesgo, eta=0.1 y mu=0.9

Train → Media = 0.00161 y DV = 6.1854-e06

Test → Media = 0.00881 y DV = 5.3838e-05

Con sesgo, eta=0.1 y mu=0.9

Train → Media = 0.00164 y DV = 2.092e-07

Test → Media = 0.00915 y DV = 6.3413e-07

**{n:5:k}**

Sin sesgo, eta=0.1 y mu=0.9

Train → Media = 0.0016 y DV = 4.5241e-06

Test → Media = 0.00883 y DV = 5.066e-05

Con sesgo, eta=0.1 y mu=0.9

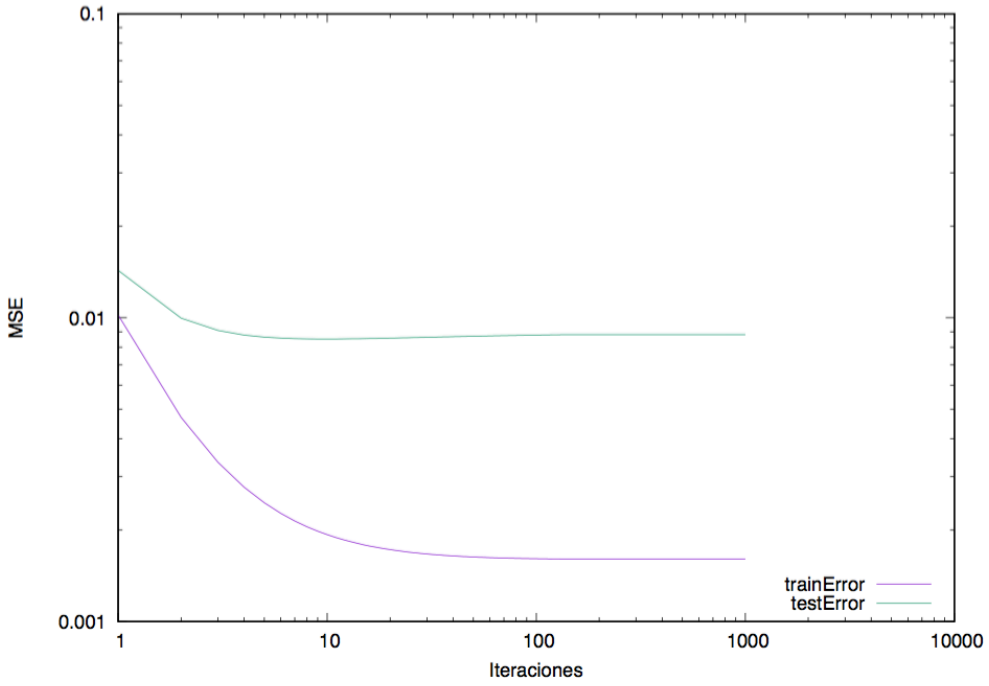
Train → Media = 0.00164 y DV = 1.3634e-06

Test → Media = 0.009152 y DV = 6.4366e-06

En ambos caso, el hecho de aplicar sesgo no mejora el modelo, incluso lo empeora. Por tanto, escogeremos la arquitectura {n:2:k} que es más simple y estudiaremos el efecto de los parámetros.

| Prueba | Iteraciones | Mu | Eta | Train/Test |
|---|---|---|---|---|
| **1** | 1000 | 0.9 | 0.1 | 0.00161/0.00881 |
| **2** | “ | “ | 0.5 | 0.0016/0.00886 |
| **3** | “ | “ | 0.9 | 0.00139/0.00897 |
| **4** | “ | 0.5 | 0.1 | 0.00161/0.00881 |
| **5** | “ | 0.1 | 0.1 | 0.00161/0.00881 |
| **6** | 500 | 0.9 | 0.1 | “ |
| **7** | 200 | 0.9 | 0.1 | “ |

Analizando los resultados, podemos ver que no hay cambio aparente en los errores, por tanto el valor de mu o eta no influyen en el entrenamiento. Esto podría deberse a que tenemos muchos patrones en nuestra base de datos, y con 1000 iteraciones ya no puede mejorar más el modelo. En la siguiente gráfica veremos, que para las 10 primeras iteraciones, minimiza el error y ya se mantiene constante.



## 5. REFERENCIAS BIBLIOGRÁFICAS

- **Presentación del algoritmo de retropropagación del error:**

**http://moodle.uco.es/m1617/pluginfile.php/168808/mod_resource/content/1/Tema 4.- Clasificación No lineal y SVM 06-07 revisado.pdf**

- **Presentación del pseudocódigo del programa:**

**http://moodle.uco.es/m1617/mod/resource/view.php?id=66915**