ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

EPSC

4º Grado en Ingeniería Informática

Especialidad de Computación

Práctica 1: Implementación del perceptrón multicapa

Introducción a los Modelos Computacionales

i72hisan@uco.es
Noelia Hinojosa Sánchez (46270988W)

UNIVERSIDAD D CORDOBA

Índice General

Capítulo 1

Introducción

En esta práctica, se va a estudiar el diseño del perceptrón multicapa. Se implementará el algoritmo de retropropagación del error haciendo uso de distintas arquitecturas que se explicarán a continuación. Por último, se analizarán los resultados obtenidos tras ejecutar el código implementado con varios datasets de datos modificando algunos parámetros necesarios para su funcionamiento que corresponderán con las distintas arquitecturas que se irán aplicando.

Este análisis que realizaremos nos servirá en un futuro ya que los modelos computacionales son la base de la Inteligencia Artificial.

Capítulo 2

Arquitecturas utilizadas

Se van a probar un total de 12 arquitecturas (Tabla 2.1):

Arquitectura	Nº capas ocultas	Nº neuronas/capa oculta
n:2:k	1	2
n:4:k	1	4
n:8:k	1	8
n:32:k	1	32
n:64:k	1	64
n:100:k	1	100
n:2:2:k	2	2
n:4:4:k	2	4
n:8:8:k	2	8
n:32:32:k	2	32
n:64:64:k	2	64
n:100:100:k	2	100

Cuadro 2.1: Descripción de las arquitecturas

Capítulo 3

Modelos utilizados

Tras realizar todas las ejecuciones indicadas en la práctica, con sus respectivas arquitecturas, he escogido, para analizar posteriormente, aquellas que daban un menor error en test.

Para cada uno de los datasets utilizados he escogido la mejor arquitectura, siendo las siguientes:

- XOR con una arquitectura {n:100:100:k}. Para este problema tenemos 2 entradas (n) y 1 salida (k), por lo que nos queda una arquitectura con 4 capas con 2 variables de entrada, 1 salida y 100 neuronas de capa oculta por cada una de las capas. Debemos añadir tanto a las capas ocultas como a la capa de entrada el sesgo.
- QUAKE con una arquitectura {n:100:100:k}. Para este problema tenemos 3 entradas (n) y 1 salida (k), por lo que nos queda una arquitectura con 4 capas con 3 variables de entrada, 1 salida y 100 neuronas de capa oculta por cada una de las capas. Debemos añadir tanto a las capas ocultas como a la capa de entrada el sesgo.

Capítulo 4

Pseudocódigo

Se ha realizado la implementación del algoritmo de retropropagación del error, como bien indica su nombre este propaga desde el final hacia adelante el error cometido en la red neuronal. Existen varias formas de implementarlo, en este caso concreto nos centraremos en el on-line, que consiste en calcular el error cometido por cada patrón que pasa, cambiando en cada iteración, si es necesario, los pesos de la red.

He seguido los siguientes pasos, basándome en el pseudocódigo de las diapositivas:

Inicio
    Topology //Definir arquitectura de la red
    Initialize() //Reservar memoria de la arquitectura
    randomWeights() //Inicializar pesos aleatorios entre -1 y +1
    Repetir
        Para cada patrón
            AccumulateChangeRestore() //Restaurar a 0 los deltaW
            feedInputs() //Alimentar las entradas
            forwardPropagate() //Activación hacia delante
            backPropagate() //Retropropagar el error
            accumulateChange() //Calcular el ajuste de los pesos
            weightAdjustment() //Aplicar el ajuste calculado a los pesos
        Fin Para
    Hasta (Condición de parada)
Fin

4.1. Función randomWeights

Función que inicializa aleatoriamente los pesos del perceptrón.

Algorithm 1 randomWeights

```
    procedure RANDOMWEIGHTS
 2:     for all layer i desde 1 hasta nOfLayers do
            for all layers[i].nOfNeurons j do
 4:             for all layers[i-1].nOfNeurons+1 do
                    ram ← randomDouble(−1, 1)
 6:                 layers[i].neurons[j].w[k] ← ram
                    layers[i].neurons[j].lastDeltaW[k] ← 0.0
 8:                 layers[i].neurons[j].deltaW[k] ← 0.0
                    layers[i].neurons[j].wCopy[k] ← 0.0
10:             end for
            end for
12:     end for
    end procedure
```

4.2. Función trainOnline

Función de entrenamiento online para un conjunto de datos de entrenamiento.

Algorithm 2 TrainOnline

```
  procedure TRAINONLINE
      for all trainDataset− >nOfPatterns i do
          performEpochOnline(trainDataset−                    >inputs[i],
  trainDataset− >outputs[i])
      end for
  end procedure
```

4.3. Función test

Función de test que prueba la red usando un dataset y devuelve el error cuadrático medio (MSE).

Algorithm 3 Test

```
procedure TEST
    MSE ← 0.0
    for all testDataset− >nOfPatterns i do
        ∗currentInput ← testDataset-¿inputs[i]
        ∗currentOutput ← testDataset-¿outputs[i]
        feedInputs(currentInputs)
        forwardPropagate()
        MSE ← MSE + obtainError(currentOutput)
    end for
    MSE ← MSE/testDataset-¿nOfPatterns
end procedure
```

4.4. Función feedInputs

Función que alimenta las entradas de la red con el vector pasado como argumento.

Algorithm 4 FeedInputs

```
procedure FEEDINPUTS
    for all layers[0].nOfNeurons i do
        layers[0].neurons[i].out ← inputs[i]
    end for
end procedure
```

4.5. Función forwardPropagate

Función que calcula la salida de la red neuronal, propagando las entradas de la red hasta la capa de salida.

Algorithm 5 ForwardPropagate

```
1: procedure FORWARDPROPAGATE
2:     net ← 0.0
3:     for all nOfLayers i do
4:         for all layers[i].nOfNeurons j do
5:             net ← 0.0
6:             for all layers[i-1].nOfNeurons k do
7:                 net ← net + layers[i].neurons[j].w[k + 1] ∗ layers[i − 1].neurons[k].out
8:             end for
9:             net ← net + layers[i].neurons[j].w[0]
10:            layers[i].neurons[j].out ← 1.0/(1 + exp(−net))
11:        end for
12:    end for
13: end procedure
```

4.6. Función restoreWeights

Función que restaura una copia de todos los pesos de la red neuronal.

Algorithm 6 RestoreWeights

```
procedure RESTOREWEIGHTS
    for all layer i desde 1 hasta nOfLayers do
        for all layers[i-1].nOfNeurons j do
            for all layers[i-1].nOfNeurons+1 k do
                layers[i].neurons[j].w[k] ← layers[i].neurons[j].wCopy[k]
            end for
        end for
    end for
end procedure
```

4.7. Función printNetwork

Función que imprime todos los pesos de la red neuronal de forma ordenada.

Algorithm 7 PrintNetwork

```
1: procedure PRINTNETWORK
2:     for all layer i desde 1 hasta nOfLayers do
3:         cout ≪ "Layer" ≪ i ≪ endl
4:         for all layers[i].nOfNeurons j do
5:             for all layers[i-1].nOfNeurons+1 k do
6:                 cout ≪ layers[i].neurons[j].w[k] ≪ ""
7:             end for
8:             cout ≪ endl
9:         end for
10:        cout ≪ endl
11:    end for
12: end procedure
```

4.8. Función getOutputs

Función que obtiene las salidas de la red neuronal y las guarda en un vector pasado como argumento.

Algorithm 8 GetOutputs

```
for all layers[nOfLayers-1].nOfNeurons i do
    output[i] ← layers[nOfLayers − 1].neurons[i].out
end for
```

4.9. Función performEpochOnline

Función que realiza una época del algoritmo de retropropagación del error de forma online. Realiza todo el proceso por cada patrón del conjunto de datos, alimentando las entradas, propagandolas hacia delante, retropropagando el error y ajustando los pesos de la red.

Algorithm 9 PerformEpochOnline

```
accumulateChangeRestore()
feedInputs(input)
forwardPropagate()
backPropagateError(target)
accumulateChange()
weightAdjustment()
```

4.10. Función backPropagateError

Función que retropropaga el error desde la salida de la red hasta las entradas de la misma.

Algorithm 10 BackPropagateError

```
1: procedure BACKPROPAGATEERROR
2:     for all layers[nOfLayers-1].nOfNeurons i do
3:         layers[nOfLayers − 1].neurons[i].delta ← −(target[i] − layers[nOfLayers − 1].neurons[i].out) ∗ layers[nOfLayers − 1].neurons[i].out ∗ (1 − layers[nOfLayers − 1].neurons[i].out)
4:     end for
5:     sumDelta ← 0.0
6:     for all layer h desde (nOfLayers-2) hasta 1 do
7:         for all layers[h].nOfNeurons j do
8:             sumDelta ← 0.0
9:             for all layers[h+1].nOfNeurons i do
10:                sumDelta ← sumDelta + layers[h + 1].neurons[i].w[j + 1] ∗ layers[h + 1].neurons[i].delta
11:            end for
12:            layers[h].neurons[j].delta ← sumDelta ∗ layers[h].neurons[j].out ∗ (1 − layers[h].neurons[j].out)
13:        end for
14:    end for
15: end procedure
```

4.11. Función accumulateChange

Función que acumula los cambios producidos por cada patrón pasado y los guarda en el vector deltaW.

Algorithm 11 AccumulateChange

```
1: procedure ACCUMULATECHANGE
2:     for all layer h desde 1 hasta nOfLayers do
3:         for all layers[h].nOfNeurons j do
4:             for all neuron i desde 1 hasta layers[h-1].nOfNeurons+1 do
5:                 layers[h].neurons[j].deltaW[i] ← layers[h].neurons[j].deltaW[i] + layers[h].neurons[j].delta * layers[h − 1].neurons[i − 1].out
6:             end for
7:             layers[h].neurons[j].deltaW[0] ← layers[h].neurons[j].deltaW[0] + layers[h].neurons[j].delta
8:         end for
9:     end for
10: end procedure
```

4.12. Función accumulateChangeRestore

Función que restaura a 0 los valores del vector deltaW, guardando antes los valores del mismo en el vector lastDeltaW.

Algorithm 12 AccumulateChangeRestore

```
1: procedure ACCUMULATECHANGERESTORE
2:     for all layer h desde 1 hasta nOfLayers do
3:         for all layers[h].nOfNeurons j do
4:             for all neuron i desde 1 hasta layers[h-1].nOfNeurons+1 do
5:                 layers[h].neurons[j].lastDeltaW[i] ← layers[h].neurons[j].deltaW[i]
6:                 layers[h].neurons[j].deltaW[i] ← 0
7:             end for
8:             layers[h].neurons[j].lastDeltaW[0] ← layers[h].neurons[j].deltaW[0]
9:             layers[h].neurons[j].deltaW[0] ← 0
10:        end for
11:    end for
12: end procedure
```

4.13. Función weightAdjustment

Función que actualiza los pesos de la red neuronal teniendo en cuenta el vector deltaW donde se van acumulando los cambios que se han realizado en la red neuronal anteriormente.

Algorithm 13 WeightAdjustment

```
1: procedure WEIGHTADJUSTMENT
2:     for all layer h desde 1 hasta nOfLayers do
3:         for all layers[h].nOfNeurons j do
4:             for all neuron i desde 1 hasta layers[h-1].nOfNeurons+1 do
5:                 layers[h].neurons[j].w[i] ← layers[h].neurons[j].w[i] −
   eta ∗ layers[h].neurons[j].deltaW[i] − mu ∗ (eta ∗
   layers[h].neurons[j].lastDeltaW[i])
6:             end for
7:             layers[h].neurons[j].w[0] ← layers[h].neurons[j].w[0] −
   eta ∗ layers[h].neurons[j].deltaW[0] − mu ∗ (eta ∗
   layers[h].neurons[j].lastDeltaW[0])
8:         end for
9:     end for
10: end procedure
```

Como comentarios centradas en mi propia implementación:

- He trabajado con los sesgos en la posición 0 de todos los vectores.
- He creado la función AccumulateChangeRestore() para inicializar a 0 los cambios de los pesos acumulados para cada patrón que pasaba la red.
- En la función Initialize() he reservado memoria tanto de la red neuronal (topology) como de todos los vectores asociados a cada neurona de la red (w, deltaW, lastDeltaW, wCopy)

Capítulo 5

Experimentos y análisis

5.1. Bases de datos utilizados

Para probar el código desarrollado, se han utilizado 3 de los 4 datasets que se nos aportan con la documentación de la práctica:

- **Dataset XOR**
  El problema del XOR tiene 2 entradas y lo que indica es que si ambas entradas son falsas o ambas verdaderas, su salida será falsa. Por lo que en el fichero del dataset, nos encontramos con 2 entradas y 1 salida, además de todas las combinaciones posibles con estas dos entradas, es decir, 4 patrones, tanto para entrenamiento como para test. Es un problema de clasificación no lineal.

- **Dataset QUAKE**
  En este problema el objetivo es averiguar la fuerza de los terremotos en la escala de Richter. Para ello tenemos 3 entradas (profundidad focal, latitud y longitud) y obtenemos una sola salida.

5.2. Valores de parámetros considerados

- Argumento **t**: Indica el nombre del fichero con los datos de entrenamiento a utilizar. Obligatorio.
- Argumento **T**: Indica el nombre del fichero con los datos de test a utilizar. Opcional.
- Argumento **i**: Indica el número de iteraciones del bucle externo a realizar. Opcional.
- Argumento **l**: Indica el número de capas ocultas del modelo de red neuronal. Opcional.
- Argumento **h**: Indica el número de neuronas a introducir en cada una de las capas ocultas. Opcional.
- Argumento **e**: Indica el valor del parámetro *eta* (η). Opcional.
- Argumento **m**: Indica el valor del parámetro *mu* (μ). Opcional.
- Argumento **s**: Indica la normalización en los datos de entrenamiento y test. Opcional.

Los parámetros considerados para el aprendizaje de los modelos de redes neuronales son el factor de aprendizaje (η) y el factor de momento (μ). El primero determina la velocidad de aprendizaje. El segundo mejora la convergencia del algoritmo teniendo en cuenta los cambios realizados anteriormente. Se han tomado para estos parámetros los valores de η=0.1 y μ=0.9 para todos los problemas y ejecuciones.

5.3. Resultados obtenidos

Se van a mostrar los resultados obtenidos para todos los problemas que se están tratando, únicamente de las arquitecturas elegidas por haber obtenido el mejor error en test.

	Iteraciones			
	100	500	1000	Arquitectura
XOR	0,0142972	0,00176564	**0,000749349**	{n:100:100:k}
QUAKE	**0,0270426**	0,027045	0,0271922	{n:100:100:k}

Cuadro 5.1: Tabla resumen de los errores de test de la mejor arquitectura encontrada

5.3.1. Problema XOR

Aquí se muestra la tabla de los resultados obtenidos para el problema del XOR tanto para entrenamiento como para test con 100 iteraciones (Tabla 5.2). En la tabla (Tabla 5.3), se pueden observar los resultados obtenidos tanto para entrenamiento como para test después de haber seleccionado la mejor arquitectura de la red.

Arquitectura	MSE Train	MSE Test
n:2:k	0,239484	0,239484
n:4:k	0,238866	0,238866
n:8:k	0,234115	0,234115
n:32:k	0,181913	0,181913
n:64:k	0,123102	0,123102
n:100:k	0,0980929	0,0980929
n:2:2:k	0,249174	0,249174
n:4:4:k	0,24913	0,24913
n:8:8:k	0,246041	0,246041
n:32:32:k	0,123773	0,123773
n:64:64:k	0,037083	0,037083
n:100:100:k	**0,0142972**	**0,0142972**

Cuadro 5.2: Tabla resultados obtenidos con 100 iteraciones probando todas las arquitecturas

Iteraciones	MSE Train	MSE Test
100	0,0142972	0,0142972
500	0,00176564	0,00176564
1000	**0,000749349**	**0,000749349**

Cuadro 5.3: Tabla resultados obtenidos con la mejor arquitectura variando el número de iteraciones

5.3.2. Problema QUAKE

Aquí se muestra la tabla de los resultados obtenidos para el problema del QUAKE tanto para entrenamiento como para test con 100 iteraciones (Tabla 5.4). En la tabla (Tabla 5.5), se pueden observar los resultados obtenidos tanto para entrenamiento como para test después de haber seleccionado la mejor arquitectura de la red.

Arquitectura	MSE Train	MSE Test
n:2:k	0,239484	0,239484
n:4:k	0,238866	0,238866
n:8:k	0,234115	0,234115
n:32:k	0,181913	0,181913
n:64:k	0,123102	0,123102
n:100:k	0,0980929	0,0980929
n:2:2:k	0,249174	0,249174
n:4:4:k	0,24913	0,24913
n:8:8:k	0,246041	0,246041
n:32:32:k	0,123773	0,123773
n:64:64:k	0,037083	0,037083
n:100:100:k	**0,0142972**	**0,0142972**

Cuadro 5.4: Tabla resultados obtenidos con 100 iteraciones probando todas las arquitecturas

Iteraciones	MSE Train	MSE Test
100	0,0299316	**0,0270426**
500	0,0294832	0,027045
1000	**0,0291627**	0,0271922

Cuadro 5.5: Tabla resultados obtenidos con la mejor arquitectura variando el número de iteraciones

5.4. Análisis de resultados

Como podemos observar en los resultados anteriores, para los 2 problemas se han usados redes neuronales con 2 capas ocultas con 100 neuronas en cada capa oculta. Esto no indica que los resultados obtenidos sean los mejores para cada problema pero si para todas las pruebas realizadas en esta práctica.

En el problema del QUAKE se ha detectado sobre-aprendizaje ya que si se observa la Tabla 5.5, se puede identificar fácilmente que con los datos de entrenamiento se obtiene un mejor error conforme más iteraciones se realicen pero en cambio, con los datos del conjunto de test el mejor error se obtiene con 100 iteraciones. Esto indica ese leve sobre-aprendizaje que está sufriendo la red neuronal.

Si comparamos el error obtenido en la práctica con los valores que se nos dan como referencia de los errores de entrenamiento y test utilizando Weka mediante una regresión lineal, en el guión de la práctica, podemos observar lo siguiente para cada problema:

- **XOR:** Con cualquier número de iteraciones (de las probadas) se mejora el error de este problema considerable. Obteniendo una diferencia del error de 0,249251.
- **QUAKE:** En este caso, la mejoría es caso imperceptible tanto si escogemos la arquitectura con 100 iteraciones (sin sobre-aprendizaje) como con 1000 iteraciones.