ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

EPSC

4º Grado en Ingeniería Informática

Especialidad de Computación

Práctica 2: Perceptrón multicapa para problemas de clasificación

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 para problemas de clasificación, adaptando el código implementado para la práctica 1.

Las modificaciones que se realizarán son las siguientes:

- Función de activación softmax en la capa de salida.
- Función de error basada en la entropía cruzada.

No se cambiará completamente la implementación solamente se dará a escoger mediante los parámetros pasados para la ejecución del programa si se utilizará la función de activación sigmoide o la función softmax, y lo mismo ocurre con la función de error.

Por otro lado, se podrá seleccionar también si se realiza el algoritmo con su versión online u offline.

Capítulo 2

Arquitecturas utilizadas

Para el problema XOR se utilizará la arquitectura que resultó mejor en la práctica anterior.

Para los dataset ildp y noMNIST se probarán las siguientes arquitecturas (Tabla 2.1):

Arquitectura	Nº capas ocultas	Nº neuronas/capa oculta
n:4:k	1	4
n:8:k	1	8
n:16:k	1	16
n:64:k	1	64
n:4:4:k	2	4
n:8:8:k	2	8
n:16:16:k	2	16
n:64:64:k	2	64

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 2 salida (k), por lo que nos queda una arquitectura con 4 capas con 2 variables de entrada, 2 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

Para este apartado, se comentará las funciones que han sido modificadas respecto de la práctica 1, donde se añade la función softmax o la función de error de entropía cruzada.

4.1. Función initialize

Función que inicializa la red neuronal, reservando memoria para los vectores que se van a utilizar exceptuando los de la capa de entrada.

Algorithm 1 Initialize

```
1: procedure INITIALIZE
2:     for all nOfLayers i do
3:         layers[i].nOfNeurons ← npl[i]
4:         layers[i].neurons ← newNeuron[npl[i]]
5:     end for
6:     for all nOfLayers i do
7:         for all npl[i] j do
8:             if i == 0 then
9:                 layers[i].neurons[j].w ← NULL
10:                layers[i].neurons[j].deltaW ← NULL
11:                layers[i].neurons[j].wCopy ← NULL
12:                layers[i].neurons[j].lastDeltaW ← NULL
13:            else
14:                layers[i].neurons[j].w ← newdouble[npl[i − 1] + 1]
15:                layers[i].neurons[j].deltaW ← newdouble[npl[i − 1] + 1]
16:                layers[i].neurons[j].wCopy ← newdouble[npl[i − 1] + 1]
17:                layers[i].neurons[j].lastDeltaW ← newdouble[npl[i − 1] +
    1]
18:            end if
19:        end for
20:    end for
21: end procedure
```

4.2. Función forwardPropagate

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

Algorithm 2 ForwardPropagate

```
1: procedure FORWARDPROPAGATE
2:     net ← 0.0
3:     sumNet ← 0.0
4:     for all layer i desde 1 hasta nOfLayers do
5:         sumNet ← 0.0
6:         for all layers[i].nOfNeurons j do
7:             net ← 0.0
8:             for all neuron k desde 1 hasta layers[i-1].nOfNeurons+1 do
9:                 net ← net + layers[i].neurons[j].w[k] ∗ layers[i − 1].neurons[k − 1].out
10:            end for
11:            net ← net + layers[i].neurons[j].w[0]
12:            if (i == (nOfLayers − 1))&&(outputFunction == 1) then
13:                layers[i].neurons[j].out ← exp(net)
14:                sumNet ← sumNet + exp(net)
15:            else
16:                layers[i].neurons[j].out ← 1.0/(1 + exp(−net))
17:            end if
18:        end for
19:        if (i == (nOfLayers − 1))&&(outputFunction == 1) then
20:            for all layers[i].nOfNeurons j do
21:                layers[i].neurons[j].out ← layers[i].neurons[j].out/sumNet
22:            end for
23:        end if
24:    end for
25: end procedure
```

4.3. Función obtainError

Función que calcula el error cometido por la red neuronal tras obtener los resultados de salida de la red. Según se indicará mediante parámetros al programa se utilizará el error cuadrático medio o la entropía cruzada.

Algorithm 3 ObtainError

```
1: procedure obtainError
2:     if errorFunction == 0 then
3:         mse ← 0.0
4:         for all layers[nOfLayers-1].nOfNeurons i do
5:             mse ← mse + pow(target[i] − layers[nOfLayers − 1].neurons[j].out, 2)
6:         end forreturn mse ← mse/(double)layers[nOfLayers − 1].nOfNeurons
7:     end if
8:     entropy ← 0.0
9:     for all layers[nOfLayers-1].nOfNeurons i do
10:        entropy ← entropy + target[i] ∗ log(layers[nOfLayers − 1].neurons[i].out
11:    end forreturn entropy ← entropy/(double)layers[nOfLayers − 1].nOfNeurons
12: end procedure
```

4.4. Función backPropagateError

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

Algorithm 4 BackPropagateError

```
1: procedure BACKPROPAGATEERROR
2:     out ← 0.0, aux ← 0.0
3:     for all layers[nOfLayers-1].nOfNeurons i do
4:         out ← layers[nOfLayers − 1].neurons[i].out
5:         layers[nOfLayers − 1].neurons[i].delta ← 0.0
6:         if outputFuction == 1 then
7:             conditionSoftmax ← 0
8:             for all layers[nOfLayers-1].nOfNeurons j do
9:                 if j == i then
10:                    conditionSoftmax ← 1
11:                else
12:                    conditionSoftmax ← 0
13:                end if
14:                if ErrorFuction == 0 then
15:                    layers[nOfLayers − 1].neurons[i].delta ← layers[nOfLayers − 1].neurons[i].delta − (target[j] − layers[nOfLayers − 1].neurons[j].out) ∗ out ∗ (conditionSoftmax − layers[nOfLayers − 1].neurons[j].out
16:                else
17:                    layers[nOfLayers − 1].neurons[i].delta ← layers[nOfLayers−1].neurons[i].delta−(target[j]/layers[nOfLayers− 1].neurons[j].out) ∗ out ∗ (conditionSoftmax − layers[nOfLayers − 1].neurons[j].out
18:                end if
19:            end for
20:        else
21:            if errorFunction == 0 then
22:                layers[nOfLayers − 1].neurons[i].delta ← −(target[i] − out) ∗ out ∗ (1 − out)
23:            else
24:                layers[nOfLayers − 1].neurons[i].delta ← −(target[i]/out) ∗ out ∗ (1 − out)
25:            end if
26:        end if
27:    end for
28:    for all layer i desde nOfLayers-2 hasta 1 do
29:        for all layers[i].nOfNeurons j do
30:            out ← layers[i].neurons[j].out
31:            aux ← 0.0
32:            for all layers[i+1].nOfNeurons k do
33:                aux ← aux + layers[i + 1].neurons[k].w[j + 1] ∗ layers[i + 1].neurons[k].delta
                                                    10
34:            end for
35:            layers[i].neurons[j].delta ← aux ∗ out ∗ (1 − out)
36:        end for
37:    end for
38: end procedure
```

4.5. 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 5 WeightAdjustment

```
1: procedure WEIGHTADJUSTMENT
2:     if online then
3:         for all layer i desde 1 hasta nOfLayers do
4:             for all neuron j desde 1 hasta layers[i].nOfNeurons do
5:                 for all neuronK k desde 1 hasta layers[i-1].nOfNeurons do
6:                     layers[i].neurons[j].w[k] ← layers[i].neurons[j].w[k] + (η * layers[i].neurons[j].w[k]) − (μ * η * layers[i].neurons[j].lastDeltaW[k])
7:                 end for
8:                 layers[i].neurons[j].w[0] ← layers[i].neurons[j].w[0]+(η* layers[i].neurons[j].w[0]) − (μ * η * layers[i].neurons[j].lastDeltaW[0])
9:             end for
10:        end for
11:    else
12:        for all layer i desde 1 hasta nOfLayers do
13:            for all neuron j desde 1 hasta layers[i].nOfNeurons do
14:                for all neuronK k desde 1 hasta layers[i-1].nOfNeurons do
15:                    layers[i].neurons[j].w[k] ← layers[i].neurons[j].w[k]− (η * layers[i].neurons[j].deltaW[k]/nOfTrainingPatterns) − (μ * (η * layers[i].neurons[j].lastDeltaW[k])/nOfTrainingPatterns)
16:                end for
17:                layers[i].neurons[j].w[0] ← layers[i].neurons[j].w[0] − (η * layers[i].neurons[j].deltaW[0]/nOfTrainingPatterns) − (μ * (η * layers[i].neurons[j].lastDeltaW[0])/nOfTrainingPatterns)
18:            end for
19:        end for
20:    end if
21: end procedure
```

4.6. Función performEpoch

Función que realiza una época del algoritmo de retropropagación del error. 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. Algunos de los pasos indicados únicamente los realiza para la versión online del algoritmo, ya que se puede indicar realizar la versión offline mediante parámetros al programa.

Algorithm 6 PerformEpoch

```
1: procedure PERFORMEPOCH
2:     if online then
3:         for all layer i desde 1 hasta nOfLayers do
4:             for all layers[i].nOfNeurons j do
5:                 for all layers[i-1].nOfNeurons+1 k do
6:                     layers[i].neurons[j].deltaW[k] ← 0.0
7:                 end for
8:             end for
9:         end for
10:    end if
11:    feedInputs(input)
12:    forwardPropagate()
13:    backpropagateError(target,errorFuction)
14:    accumulateChange()
15:    if online then
16:        weightAdjustement()
17:    end if
18: end procedure
```

4.7. Función train

Función que entrena la red neuronal, puede realizar su versión online u offline según los parámetros indicados al programa.

Algorithm 7 Train

```
1: procedure TRAIN
2:     if !online then
3:         for all layer i desde 1 hasta nOfLayers do
4:             for all layers[i].nOfNeurons j do
5:                 for all layers[i-1].nOfNeurons+1 k do
6:                     layers[i].neurons[j].deltaW[k] ← 0.0
7:                 end for
8:             end for
9:         end for
10:    end if
11:    for all trainDataset- nOfPatterns do
12:        performEpoch(trainDataset- inputs[i], trainDataset- outputs[i],
   errorFunction)
13:    end for
14:    if !online then
15:        WeightAdjustement()
16:    end if
17: end procedure
```

4.8. Función test

Función que calcula las salidas de la red neuronal, calculando el error para cada uno de los patrones pasados por la red. En esta práctica utilizamos el dataset de entrenamiento.

Algorithm 8 Test

```
1: procedure TEST
2:     sum ← 0.0
3:     for all testDataset− >nOfPatterns i do
4:         feedInputs(testDataset− >inputs[i])
5:         forwardPropagate()
6:         sum ← sum + obtainError(testDataset− > outputs[i], errorFunction)
7:     end for
8:     if errorFunction == 0 then
9:       return sum/testDataset− > nOfPatterns
10:    end if
11:      return −1 ∗ (sum/testDataset− > nOfPatterns)
12: end procedure
```

4.9. Función testClassification

Función que clasifica los patrones del conjunto de test y genera una matriz de confusión con los resultados obtenidos.

Algorithm 9 TestClassification

```
1: procedure TESTCLASSIFICATION
2:     ccr ← 0, expectedClass ← 0, obtainedClass ← 0
3:     maximo ← 0.0, maximo2 ← 0.0
4:     *outArray ← newdouble[layers[nOfLayers − 1].nOfNeurons]
5:     for all dataset- nOfPatterns i do
6:         feedInputs(dataset-¿inputs[i])
7:         forwardPropagate()
8:         getOutputs(outArray)
9:         maximo ← outArray[0]
10:        maximo2 ← dataset− > outputs[i][0]
11:        for all output j desde 1 hasta dataset- nOfOutputs do
12:            if maximo < outArray[j] then
13:                maximo ← outArray[j]
14:                obtainedClass ← j
15:            end if
16:            if maximo2 < dataset− > outputs[i][j] then
17:                maximo2 ← dataset− > outputs[i][j]
18:                expectedClass ← j
19:            end if
20:        end for
21:        if expectedClass == obtainedClass then
22:            ccr ← ccr + 1
23:        end if
24:        confusionMatrix[expectedClass][obtainedClass] ←
   confusionMatrix[expectedClass][obtainedClass] + 1
25:    end for
26:    delete[]outArray
27:     return ((double) ccr / dataset- nOfPatterns) * 100
28: end procedure
```

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 se utilizará el mismo fichero. Es un problema de clasificación no lineal. Para esta práctica se ha modificado el fichero a la codificación 1-de-k, encontrándonos con dos salidas en lugar de una.

Los dataset de ildp y noMNIST no han podido finalmente utilizarse debido a que se obtiene un Segmentation Fault que no se he conseguido solucionar.

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 **o**: Indica si usar la versión online u offline del algoritmo. Opcional.
- Argumento **f**: Indica la función de error a utilizar durante el aprendizaje. Opcional.
- Argumento **s**: Indica si usar la función softmax o no en la capa de salida. Opcional.
- Argumento **n**: Indica si se van a normalizar las entradas de los datos de entrenamiento y test en el intervalo [-1, 1]

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=0.7 y μ=1 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	75	75	75	{n:100:100:k}

Cuadro 5.1: Tabla errores de test CCR

5.4. Análisis de resultados

Como se ha mencionado anteriormente, finalmente solo se ha podido probar con el problema del XOR. En la tabla anterior (5.1) se puede observar que se obtiene el mismo valor del CCR independientemente del número de iteraciones aplicados.

Si comparamos el CCR 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, podemos observar lo siguiente:

- **XOR:** Con cualquier número de iteraciones (de las probadas) se mejora el error de este problema ligeramente, pasando de un CCR tanto en entrenamiento como en test del 50 % a un 75 %.