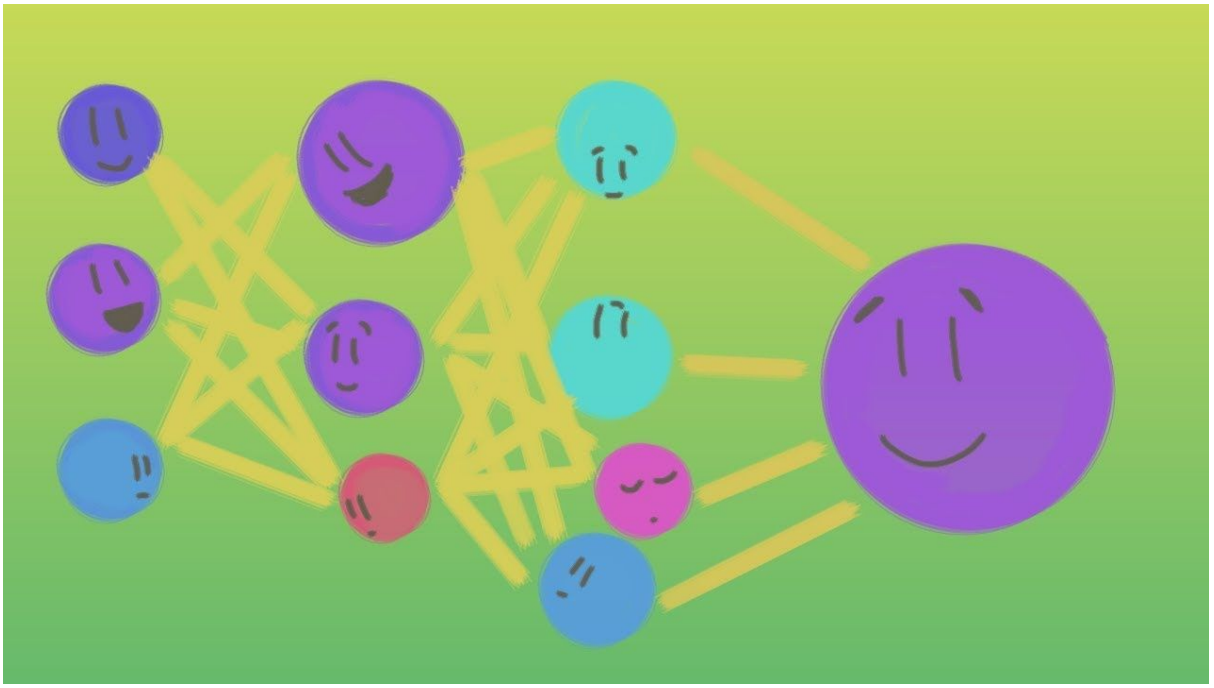
Introducción a los modelos computacionales

Práctica 1: Implementación del perceptrón multicapa



Universidad de Córdoba, Escuela Politécnica Superior de Córdoba
Titulación en ingeniería informática, 4º Curso, modularidad Computación

Rafael Hormigo Cabello - i72hocar@uco.es

1.- Descripción de los modelos de redes neuronales.

Los modelos neuronales que vamos a considerar se componen de tres a cuatro capas, siendo siempre la primera capa la capa de entrada y la última capa la capa de salida. Las capas ocultas, en caso de haber más de una, tendrán todas el mismo número de neuronas. Todas las neuronas de cada capa están directamente conectadas con todas las neuronas de la capa siguiente y anterior a la suya.

2.- Pseudocódigo del algoritmo de retropropagación.

Comenzaremos con el pseudocódigo de la función que ejecuta la retropropagación e iremos desglosando las funciones importantes que se usen en ella. Se explicarán por medio de comentarios en el mismo pseudocódigo, excepto comentarios adicionales que se especificarán fuera de este.

```
runOnlineBackPropagation(trainDataset, pDatosTest, maxIter, errorTrain, errorTest)
        //Iniciamos los pesos con valores aleatorios
        randomWeights();
        //Si hay conjunto de validación se separa el conjunto de entrenamiento en dos, el
        //conjunto de entrenamiento nuevo y el conjunto de validación.
        if validationRatio > 0 and validationRatio < 1 do
                splitDataset(trainDataset, newTrainData, validationTrainData);//función que
                //realiza la separación del dataset de entrenamiento
        else
                newTrainData = trainDataset;
        endif
        do
                //Entrenamos la red con el dataset de entrenamiento
                trainOnline(newTrainData);
                //Probamos la red con el dataset de entrenamiento y así obtener el error de
                //entrenamiento
                trainError = test(newTrainData);
                //si el error es el mínimo lo guardamos, además guardamos los pesos
                if countTrain == 0 or trainError < minTrainError do
                        minTrainError = trainError;
                        copyWeights();
                        iterWithoutImproving = 0;
                //Si el error es menos de 0.00001 veces más grande que el error mínimo lo
                //consideramos como igual, por lo tanto no contamos esta iteración como
                //que no mejora
                else if trainError - minTrainError < 0.00001 do
```

```
                    iterWithoutImproving = 0;
                //Finalmente si el error empeora contamos una iteración de no mejora más
                else
                    iterWithoutImproving++;
                endif
                //Si llegamos a 50 iteraciones sin mejora reponemos los pesos de la
                //iteración que menor error ha dado y salimos del bucle
                if iterWithoutImproving == 50 do
                    restoreWeigths();
                    countTrain = maxiter;
                endif
                //Si hay conjunto de validación recogeremos el error de validación para
                //parar la ejecución si hay sobreentrenamiento
                if validationRatio > 0 and validationRatio < 1 do
                    validationError = test(ValidationTrainData);
                    if countTrain == 0 or validationError < lastValidationError do
                        iterWithoutImprovingV = 0;
                    else if validationError - lastValidationError < 0.00001 do
                        iterWithoutImprovingV = 0;
                    else
                        iterWithoutImprovingV++;
                    endif
                    //la única diferencia entre validación y entrenamiento es que
                    //cogemos siempre el error anterior y no el menor, además que no
                    //guardamos los pesos
                    lastValidationError = validationError;
                endif
                countTrain++;
        while countTrain<maxiter
        … //obviamos esta parte que solo proporciona información al usuario
        //copiamos los errores en las variables para usarlas fuera de la función
        errorTest = testError;
        errorTrain = minTrainError;
end
```

La función splitDataset(trainDataset, newTrainData, validationTrainData) no la explicamos puesto que la implementación no tiene relevancia en el algoritmo de retropropagación. Las funciones randomWeigths(), copyWeigths() y restoreWeigths() se obvian puesto que son triviales (recorrer los pesos de la red y operar con ellos).

```
trainOnline(trainDataset)
        //Esta función ejecuta una época por cada patrón del dataset
        for i from o to trainDataset->nOfPatterns step 1
            performEpochOnline(trainDataset->inputs[i], traindataset->outputs);
        endfor
end
```

```
performEpochOnline(input[], target[])
        //igualamos cada deltaW a cero
        deltaW[] = 0;
        //Alimentamos las entradas
        feedinputs(input);
        //propagamos hacia delante
        forwardPropagate();
        //Retro propagamos el error
        backpropagateError(target);
        //acumulamos el cambio de los pesos
        accumulateChange();
        //aplicamos el cambio de pesos
        weightAdjustment();
end

forwardPropagate()
        //recorremos los pesos de cada neurona de cada capa excepto la de entrada
        for i from 1 to nOfLayers step 1
                for j from 0 to layers[i].nOfneurons step 1
                        for k from 0 to layers[i].neurons[j].nOfWeigths step 1
                                //calculamos el valor net
                                layers[i].neurons[j].net += layers[i].neurons[j].w[k];
                        endfor
                        //calculamos la salida como la sigmoide de net
                        layers[i].neurons[j].out = 1/(1+e^-layers[i].neurons[j].net);
                endfor
        endfor
end

backpropagateError(target)
        //para cada neurona de salida calculamos el delta
        for i from 0 to nOfOutputNeurons step 1
                //g'(net) = out(1-out) en caso de sigmoide
                outputNeuron[i].delta=-(target[i] - realOut[i])*g'(net);
        enfor
        for i from nOfLayers-1 to 1 step 1
                for j from o to layers[i].nOfneurons step 1
                        for k from 0 to layers[i+1].nOfNeurons step 1
                                //calculamos el sumatorio de delta por el peso de las
                                 //neuronas
                                //de la siguiente capa conectadas con la neurona actual
                                //w[j] porque la posición j del vector de pesos de la neurona
                        k
                                //representa el peso del enlace entre la neurona j y la
                                //neurona k
                                sum += layers[i+1].neurons[k].delta *
                                layers[i+1].neurons[k].w[j].;
                        endfor
                        //Aplicamos el delta a la neurona j
                        layers[i]neurons[j].delta = sum * g'(net);
                        sum = 0;
                endfor
        endfor
end
```

```
accumulateChange()
        //calculamos deltaW para cada peso según la fórmula
        for i from 1 to nOfLayers step 1
                for j from 0 to layers[i].nOfneurons step 1
                        for k from 1 to layers[i].neurons[j].nOfWeigths step 1
                                layers[i].neurons[j].deltaW[k] += layers[i].neurons[j].delta *
                                layers[i-1].neurons[j].out;
                        endfor
                        //considerando el sesgo
                        layers[i].neurons[j].deltaW[0] += layers[i].neurons[j].delta;
                endfor
        endfor
end

weightAdjustment()
        //aplicamos el cambio en los pesos según la fórmula
        for i from 1 to nOfLayers step 1
                for j from 0 to layers[i].nOfneurons step 1
                        for k from 1 to layers[i].neurons[j].nOfWeigths step 1
                                layers[i].neurons[j].w[k] = layers[i].neurons[j].w[k] -
                                eta*layers[i].neurons[j].deltaW[k] -
                                mu*(eta*layers[i].neurons[j].lastDeltaW[k]);
                        endfor
                        //considerando el sesgo
                        layers[i].neurons[j].w[0] = layers[i].neurons[j].w[0] -
                        eta*layers[i].neurons[j].deltaW[0] -
                        mu*(eta*layers[i].neurons[j].lastDeltaW[0]);
                endfor
        endfor
end
```

La explicación de feedInputs(inputs) se obvia pues consiste solo en introducir los valores del vector inputs en las neuronas de entrada.
La función test(trainDataset) consiste simplemente en propagar hacia delante y obtener el error medio.

3.- Pruebas y análisis de resultados

Contamos con 4 bases de datos para nuestras pruebas: xor, sin, quake y parkinsons.
Xor: esta base de datos cuenta con 4 patrones tanto en test como en train, los patrones están duplicados ya que al ser el problema lógico xor solo contamos con 4 casos.Estos patrones tienen 2 entradas y una salida.
Sin: la base de datos consiste en la función del seno con ruido, por lo cual se quiere predecir lo mejor posible la salida que daría la función sin ruido. Contamos con 120 patrones de entrenamiento y 41 de test, cada uno con 1 entrada y una salida.
Quake: tenemos 1633 patrones de entrenamiento y 546 de test, cada uno con 3 entradas y 1 de salida.
Parkinsons: los patrones tienen 19 entradas y 2 salidas, contamos con 4406 patrones de entrenamiento y 1469 de test.

Se ha ejecutado un script que prueba las bases de datos con combinaciones de capas, neuronas y mu para determinar cuál es la mejor arquitectura para cada base de datos. Probamos 1 o 2 capas, 2, 4, 8, 16, 32, 64 o 100 neuronas por capa oculta, y mu puede tomar los valores 0.9, 0.8, 0.7 o 0.5, con esto podemos determinar que mu no tiene efecto alguno en los resultados. Vamos a repasar los resultados para cada arquitectura.

Dataset	Nº iterations	Layers	Nº neurons	eta	mu	Validation Ratio	Decrement Factor	Average test error	Average train error
xor	100	1	2	0.1	0.9	0.0	1	0.214128	0.214128
xor	100	1	4	0.1	0.9	0.0	1	0.117541	0.117541
xor	100	1	8	0.1	0.9	0.0	1	0.0651483	0.0651483
xor	100	1	16	0.1	0.9	0.0	1	0.0228261	0.0228261
xor	100	1	32	0.1	0.9	0.0	1	0.0130085	0.0130085
xor	100	1	64	0.1	0.9	0.0	1	0.00923422	0.00923422
xor	100	1	100	0.1	0.9	0.0	1	0.153623	0.153623
xor	100	2	2	0.1	0.9	0.0	1	0.249411	0.249411
xor	100	2	4	0.1	0.9	0.0	1	0.24331	0.24331
xor	100	2	8	0.1	0.9	0.0	1	0.182162	0.182162
xor	100	2	16	0.1	0.9	0.0	1	0.0348729	0.0348729
xor	100	2	32	0.1	0.9	0.0	1	0.00774114	0.00774114
xor	100	2	64	0.1	0.9	0.0	1	0.00290688	0.00290688
xor	100	2	100	0.1	0.9	0.0	1	0.00170543	0.00170543

Podemos ver que para la xor la mejor arquitectura es es con dos capas y cien neuronas por capa, aunque dado que solo queremos obtener un error menor a 0.25 podemos elegir cualquier arquitectura.

16	sin	100	1	2	0.1	0.9	0.0	1	8.31831e-05	0.00826437
17	sin	100	1	4	0.1	0.9	0.0	1	7.72549e-05	0.00826425
18	sin	100	1	8	0.1	0.9	0.0	1	7.19973e-05	0.00826398
19	sin	100	1	16	0.1	0.9	0.0	1	6.86035e-05	0.00826452
20	sin	100	1	32	0.1	0.9	0.0	1	6.92819e-05	0.00826427
21	sin	100	1	64	0.1	0.9	0.0	1	6.72571e-05	0.00826425
22	sin	100	1	100	0.1	0.9	0.0	1	7.00144e-05	0.00826363
23	sin	100	2	2	0.1	0.9	0.0	1	8.13601e-05	0.00826438
24	sin	100	2	4	0.1	0.9	0.0	1	7.44729e-05	0.00826394
25	sin	100	2	8	0.1	0.9	0.0	1	7.05429e-05	0.00826389
26	sin	100	2	16	0.1	0.9	0.0	1	6.96355e-05	0.00826387
27	sin	100	2	32	0.1	0.9	0.0	1	6.81877e-05	0.00826402
28	sin	100	2	64	0.1	0.9	0.0	1	6.70056e-05	0.00826431
29	sin	100	2	100	0.1	0.9	0.0	1	9.08836e-05	0.00826458

Para el seno la mejor arquitectura es con una capa oculta y cien neuronas en capa oculta. Pese a esto los errores difieren tan poco que bien podríamos utilizar la arquitectura más simple para ahorrar tiempo de cómputo.

quake	100	1	2	0.1	0.9	0.0	1	3.41134e-06	0.00122354
quake	100	1	4	0.1	0.9	0.0	1	2.77097e-06	0.00122313
quake	100	1	8	0.1	0.9	0.0	1	2.52394e-06	0.00122259
quake	100	1	16	0.1	0.9	0.0	1	1.94962e-06	0.00122299
quake	100	1	32	0.1	0.9	0.0	1	1.91928e-06	0.00122274
quake	100	1	64	0.1	0.9	0.0	1	2.58117e-06	0.00122183
quake	100	1	100	0.1	0.9	0.0	1	1.69448e-06	0.00122288
quake	100	2	2	0.1	0.9	0.0	1	2.87913e-06	0.00122342
quake	100	2	4	0.1	0.9	0.0	1	2.41811e-06	0.00122327
quake	100	2	8	0.1	0.9	0.0	1	2.10457e-06	0.00122307
quake	100	2	16	0.1	0.9	0.0	1	1.66856e-06	0.0012232
quake	100	2	32	0.1	0.9	0.0	1	2.05041e-06	0.00122256
quake	100	2	64	0.1	0.9	0.0	1	1.6859e-06	0.00122299
quake	100	2	100	0.1	0.9	0.0	1	3.2138e-06	0.0012208

Lo mismo ocurre para quake, los errores son tan parecidos que podemos usar la arquitectura más sencilla.

parkinsons	100	1	2	0.1	0.9	0.0	1	0.000341449	0.00124653
parkinsons	100	1	4	0.1	0.9	0.0	1	0.000341405	0.0012465
parkinsons	100	1	8	0.1	0.9	0.0	1	0.000341384	0.00124647
parkinsons	100	1	16	0.1	0.9	0.0	1	0.000341404	0.0012464
parkinsons	100	1	32	0.1	0.9	0.0	1	0.000341476	0.00124625
parkinsons	100	1	64	0.1	0.9	0.0	1	0.000341516	0.00124615
parkinsons	100	1	100	0.1	0.9	0.0	1	0.000341323	0.00124629
parkinsons	100	2	2	0.1	0.9	0.0	1	0.00034131	0.00124673
parkinsons	100	2	4	0.1	0.9	0.0	1	0.000341174	0.00124677
parkinsons	100	2	8	0.1	0.9	0.0	1	0.000341176	0.00124674
parkinsons	100	2	16	0.1	0.9	0.0	1	0.000341257	0.00124667
parkinsons	100	2	32	0.1	0.9	0.0	1	0.000341399	0.00124652
parkinsons	100	2	64	0.1	0.9	0.0	1	0.000341462	0.0012462
parkinsons	100	2	100	0.1	0.9	0.0	1	0.000341505	0.0012461

Y parkinsons nos da resultados parecidos, la arquitectura no nos proporciona mucho margen en el error.

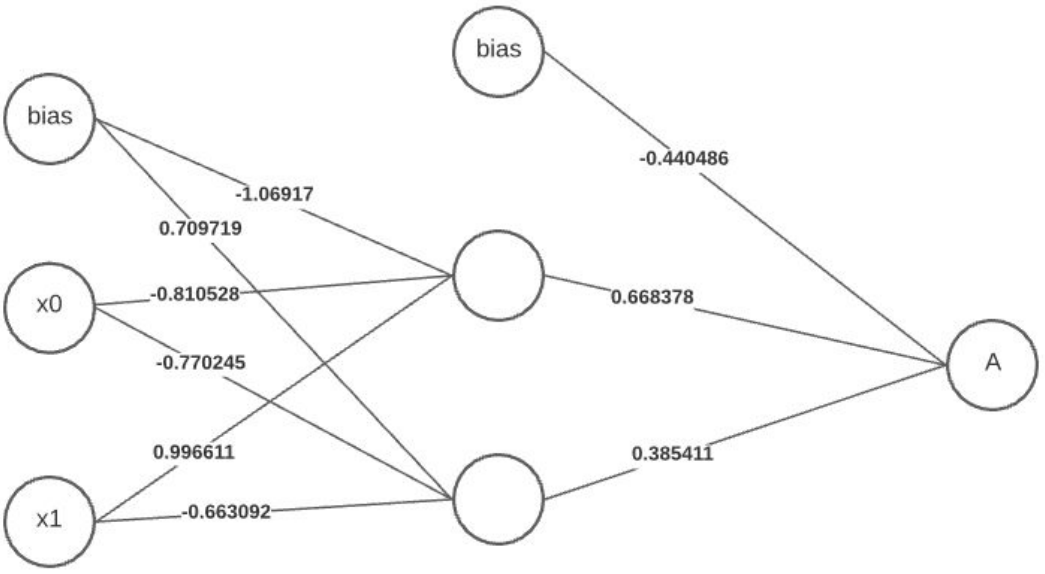
Una vez encontradas las mejores arquitecturas las usamos para determinar qué valores debemos darles al factor de decremento y al ratio de patrones de validación. Para ello usaremos otro script que pruebe las combinaciones de v {0.0, 0.15, 0.25} y F {1,2}.

Dataset	Nº iterations	Layers	Nº neurons	eta	mu	Validation Ratio	Decrement Factor	Average test error	Average train error
xor	100	1	8	0.1	0.9	0.0	1	0.0651483	0.0651483
xor	100	1	8	0.1	0.9	0.0	2	0.0651483	0.0651483
sin	100	1	2	0.1	0.9	0.0	1	8.31831e-05	0.00826437
sin	100	1	2	0.1	0.9	0.0	2	8.31831e-05	0.00826437
sin	100	1	2	0.1	0.9	0.15	1	0.000263349	0.00602093
sin	100	1	2	0.1	0.9	0.15	2	0.000263349	0.00602093
sin	100	1	2	0.1	0.9	0.25	1	9.41955e-05	0.00660489
sin	100	1	2	0.1	0.9	0.25	2	9.41955e-05	0.00660489
quake	100	1	2	0.1	0.9	0.0	1	3.41134e-06	0.00122354
quake	100	1	2	0.1	0.9	0.0	2	3.41134e-06	0.00122354
quake	100	1	2	0.1	0.9	0.15	1	3.97413e-06	0.0012946
quake	100	1	2	0.1	0.9	0.15	2	3.97413e-06	0.0012946
quake	100	1	2	0.1	0.9	0.25	1	3.98226e-06	0.000979047
quake	100	1	2	0.1	0.9	0.25	2	3.98226e-06	0.000979047
parkinsons	100	1	2	0.1	0.9	0.0	1	0.000341449	0.00124653
parkinsons	100	1	2	0.1	0.9	0.0	2	0.000341449	0.00124653
parkinsons	100	1	2	0.1	0.9	0.15	1	0.0003416	0.00127948
parkinsons	100	1	2	0.1	0.9	0.15	2	0.0003416	0.00127948
parkinsons	100	1	2	0.1	0.9	0.25	1	0.000341416	0.00108785
parkinsons	100	1	2	0.1	0.9	0.25	2	0.000341416	0.00108785

Con esta tabla podemos observar como el valor del decremento no influye, esto es normal ya que estamos utilizando arquitecturas de 1 sola capa. También vemos como en algunas bases de datos como el seno o quake el uso de un conjunto de validación hace que el error aumente, pero en otras como parkinsons puede hacer que el error baje un poco.

Por lo tanto, podemos determinar que para estos problemas lo mejor es usar arquitecturas simples para ahorrar tiempo de computación, ya que el valor del error para arquitecturas más complejas no decrementa tanto como para considerarlas.

Para terminar realizamos una ejecución del algoritmo para el problema xor y realizamos una gráfica del modelo con los pesos dados por el algoritmo.


bias
bias
x0
x1
A
-1.06917
0.709719
-0.810528
-0.770245
0.996611
-0.663092
-0.440486
0.668378
0.385411