Capítulo 2

REDES NEURONALES PERCEPTRON Y ADALINE

Introducción

El Perceptron fue el primer modelo de RNA presentado a la comunidad científica por el psicólogo Frank Rosenblatt en 1958. Como es natural despertó un enorme interés en la década de los años sesenta, debido a su capacidad para aprender a reconocer patrones sencillos con una superficie de separación li-neal, razón por la cual fue también objeto de severas críticas que terminaron por dejar en el olvido la propuesta de Rosenblatt. La estructura del Perceptron es supremamente sencilla: en su entrada posee varias neuronas lineales que se encargan de recibir el estímulo externo de la red y a la salida presenta una única capa de neuronas, con función de activación binaria o bipolar lo cual trae como consecuencia que la salida de cada neurona esté entre dos posibles valores.

De manera casi simultánea al desarrollo del Perceptron, Bernard Widrow y su estudiante Marcian Hoff presentaron su red neuronal ADALINE, esta red es similar al Perceptron pero en vez de tener una función de activación binaria o bipolar posee una función de activación lineal. La red ADALINE, a pesar de tener las mismas limitaciones de la red Perceptron, fue un gran adelanto en el desarrollo de las redes neuronales pues el mecanismo de aprendizaje que utiliza es basado en elementos de la teoría de optimización; específicamente se usa la propuesta del gradiente descendente para la deducción de la regla de entrenamiento. El uso del gradiente descendente es la base de los algoritmos de aprendizaje que hoy se usan para entrenar redes neuronales más sofisticadas que las presentadas en este capítulo. Uno de los aspectos más interesante de la red ADALINE es su aplicación en problemas de filtrado de señales, pues su estructura se ajusta a la de un filtro adaptativo.

Red neuronal perceptron

Arquitectura y funcionamiento

El Perceptron es una red monocapa pues la capa de entrada, al no realizar procesamiento alguno sobre la señal de estímulo, no se tiene en la cuenta y por tanto, esta red posee solo una capa de procesamiento que es la misma de salida. Posee conectividad total, ya que la neurona de la capa de salida está conectada a todas las unidades o neuronas de entrada, tal como lo observamos en la figura 2.1. Además, por lo general se le implementan unidades de tendencia o umbral para que la superficie de separación no se quede anclada en el origen del espacio *n-dimensional* en donde se esté realizando la separación lineal, ya que existen algunos problemas de clasificación de patrones que sin el término de umbral no tendrían solución. La función de activación que utiliza la neurona de salida es la tipo escalón binario [0,+1] o bipolar [-1,+1].

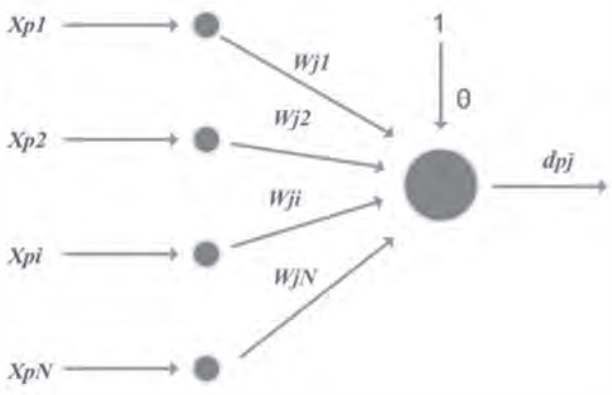


Fig. 2.1 Arquitectura de un Perceptron

Para comprender de mejor manera el funcionamiento de un Perceptron, veamos apoyados en la figura 2.2, el comportamiento de esta red cuando la dimensión del espacio de entrada es igual a dos, esto significa que la red tendrá dos entradas y por simplicidad se supondrá una sola neurona de procesamiento.

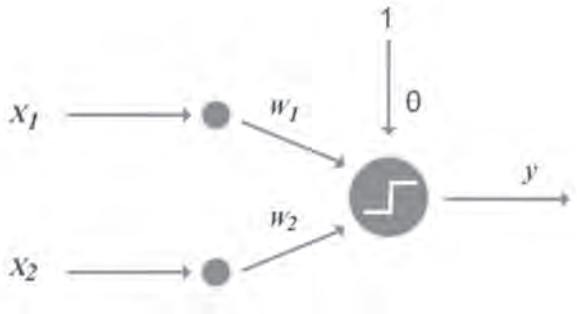


Fig. 2.2 Perceptron para trabajar puntos en dos dimensiones

La única neurona de salida del Perceptron realiza la suma ponderada de las entradas, se suma el umbral y se pasa el resultado a una función de activación binaria o bipolar. Debido a la naturaleza de la función de activación la salida de la red tiene solo dos opciones para la señal de salida *y*, 1 ó 0. Si generamos la salida de la red para todos los posibles valores del espacio de entrada y, si "pintamos" de negro el semiplando donde la salida es 1, o de blanco si la salida es 0, el plano quedará dividido como mostramos en la figura 2.3.

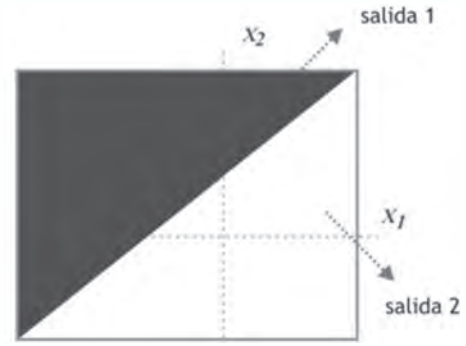


Fig. 2.3 Semiplanos generados por un Perceptron

Para analizar matemáticamente cuál es la superficie de separación que genera el Perceptron en el plano cartesiano, tomemos la expresión de la entrada neta,

$$Neta = x_1 * w_1 + x_2 * w_2 + \theta \tag{2.1}$$

La superficie de separación se tiene exactamente cuando la entrada neta es igual a cero,

$$x_1 * w_1 + x_2 * w_2 + \theta = 0 \tag{2.2}$$

Teniendo esta condición, reorganicemos la expresión de la siguiente manera,

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{\theta}{w_2} \tag{2.3}$$

La superficie de separación generada por el Perceptron corresponde a una línea recta y, este comportamiento, se puede extender a problemas donde tenemos tres entradas y el espacio de trabajo será tridimensional. En este caso la superficie de separación corresponderá a un plano con la siguiente ecuación,

$$x_3 = -\frac{w_1}{w_3} x_1 - \frac{w_2}{w_3} x_2 - \frac{\theta}{w_3} \tag{2.4}$$

Extendamos el análisis a problemas donde la dimensión del vector de patrones de entrada del Perceptron es *n,* ahora la superficie de separación lineal es de dimensión (*n-1)* que denominaremos *hiperplano.* Cuando entrenamos un Perceptron modificaremos sus pesos, lo cual trae como consecuencia la variación de la superficie de separación que está generando.

En la clasificación de patrones podemos aplicar directamente los conceptos de Perceptron, por ejemplo, tenemos un conjunto de datos con dos clases que representamos en el plano de la figura 2.4 con círculos negros y grises. El perceptron propuesto para resolver esta tarea genera una línea que separa las dos clases adecuadamente.

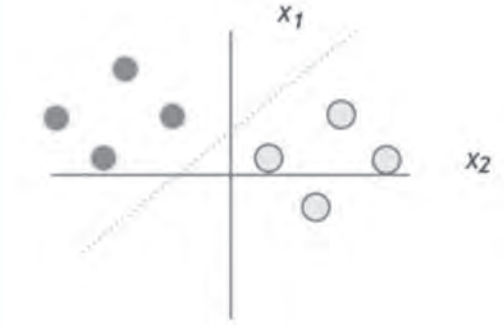


Fig. 2.4 Regiones de separación de un Perceptron

Algoritmo de aprendizaje

Antes de pasar a describir los pasos del algoritmo de aprendizaje del Perceptron, vamos a presentar la nomenclatura utilizada.

N Número de neuronas en la capa de entrada.
x_{pi} Componente *i-ésima* de la entrada correspondiente al *p-ésimo* patrón.
$Neta_{pj}$ Entrada neta de la neurona *j-ésima* de la única capa de procesamiento del Perceptron.
w_{ji} Valor del peso de la conexión entre la *j-ésima* neurona de la capa de procesamiento y la *i-ésima* neurona de la capa de entrada.
θ_j Valor del umbral o tendencia para la *j-ésima* neurona de la capa de procesamiento.
y_{pj} Valor de salida de la *j-ésima* neurona de la capa de procesamiento.
d_{pj} Valor de salida deseado para la *j-ésima* neurona de la capa de procesamiento.
e_p Valor del error para el *p-ésimo* patrón de aprendizaje.
α Tasa de aprendizaje.

Paso 1
Asignamos los valores iniciales de forma aleatoria a los pesos w_{ji} y al umbral θ_j. Se sugiere que estos valores estén en un rango entre -1 y +1.

Paso 2
Presentamos a la red el vector de entrada $\boldsymbol{x_p}$ y especificar el vector de salida deseada $\boldsymbol{d_{pj}}$.

$$\boldsymbol{x_p} = \{x_{p1}, x_{p2}, \ldots, x_{pN}\} \tag{2.5}$$

Paso 3
Calculamos los valores netos procedentes de las entradas para las unidades de la capa de salida.

$$Neta_{pj} = \sum_{i=1}^{N} w_{ji} x_{pi} + \theta_j \tag{2.6}$$

Paso 4
Calculamos la salida de la red.

$$y_{pj} = f_j\,(Neta_{pj}) \tag{2.7}$$

Paso 5
Actualizamos los pesos de la capa de salida.

$$w_{ji}(t+1) = w_{ji}(t) + \alpha\left[d_{pj} - y_{pj}\right] x_{pi} \tag{2.8}$$

Paso 6
Calculamos el error del *p-ésimo* patrón.

$$e_p = \frac{1}{2}\sum_{j=1}^{M}(d_{pj} - y_{pj})^2 \qquad (2.9)$$

Paso 7
Si el error es diferente de cero para cada uno de los patrones aprendidos volver al paso 2.

La velocidad de convergencia la ajustamos con el valor de parámetro de aprendizaje (α), normalmente dicho parámetro debe ser un número pequeño (del orden de 0.05 a 0.25), aunque este valor dependerá de las características del problema que estemos resolviendo.

Vamos a proponer como un ejemplo del proceso de aprendizaje, la solución de la función lógica OR usando un Perceptron. Los patrones de entrada para el entrenamiento de la red se presentan en la siguiente tabla.

Tabla No 2.1. Patrones de entrenamiento

Patrón	x_1	x_2	***d (OR)***
$\mathbf{p_1}$	0	0	0
$\mathbf{p_2}$	0	1	1
$\mathbf{p_3}$	1	0	1
$\mathbf{p_4}$	1	1	1

Cuya representación en forma matricial viene dada por,

$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$$

La salida de la red se produce de acuerdo a la siguiente condición:

$$\text{Si} \quad w_1x_1 + w_2x_2 \geq 0 \longrightarrow y = 1$$

$$\text{Si} \quad w_1x_1 + w_2x_2 < 0 \longrightarrow y = 0$$

Si seguimos los pasos del algoritmo, empezamos por elegir los valores de los pesos, *bias* y factor de aprendizaje:

θ=0.5, w_1=-0.5, w_2=-0.5, α=1

Se deben evaluar los cuatro patrones de entrada 0 0, 0 1, 1 0, 1 1; sin embargo, para este ejemplo, sólo evaluaremos el patrón 1,16, que de acuerdo a la tabla 2.1 tiene salida igual a 1.

- Con el patrón 1,16, la suma ponderada se calcula así,

$$Net = w_1 x_1 + w_2 x_2 + \theta = (-0.5)(1) + (-0.5)(1) + 0.5 = -0.5$$

la salida de la red es $y = f(Net) = 0$

Calculamos el error $e(t) = d(t) - y(t) = 1 - 0 = 1$

Se procede con la modificación de los pesos,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta w = \begin{cases} w_1(t+1) = -0.5 + (1.0)(1.0-0)(1.0) = 0.5 \\ w_2(t+1) = -0.5 + (1.0)(1.0-0)(1.0) = 0.5 \\ \theta(t+1) = 0.5 + (1.0)(1.0-0) = 1.5 \end{cases}$$

Con estos nuevos valores de pesos y *bias* se vuelven a calcular las salidas como se hizo con el patrón inicial 1 1. El procedimiento continua de manera iterativa hasta que los pesos tengan los valores adecuados para modelar la función lógica OR.

Red neuronal adaline (adaptative linear element)

Arquitectura

La red ADALINE es similar al Perceptron, pero en vez de tener una función de activación binaria o bipolar posee una función de activación lineal, esto se puede observar en la figura 2.5

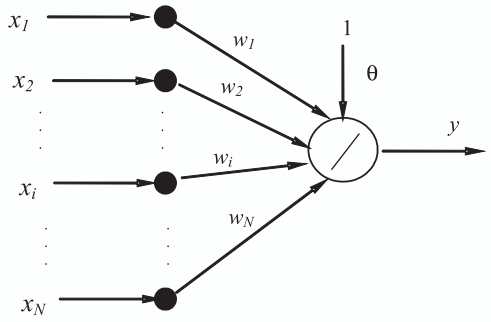


Fig. 2.5. Estructura de una red tipo ADALINE

El modelo del ADALINE fue desarrollado por Bernard Widrow y su estudiante Marcian Hoff a comienzos de la década de los sesenta. Como la neurona de esta red tiene una función de activación lineal, el valor de salida será igual al valor de la entrada neta que la neurona esté recibiendo, que lo podemos expresar con la ecuación 2.10.

$$y = \sum_{i=1}^{N} w_i x_i + \theta \tag{2.10}$$

Es importante recalcar que la red tipo ADALINE puede tener más de una neurona en su capa de procesamiento, dando origen a un sistema con N entradas y M salidas, como el de la figura 2.6.

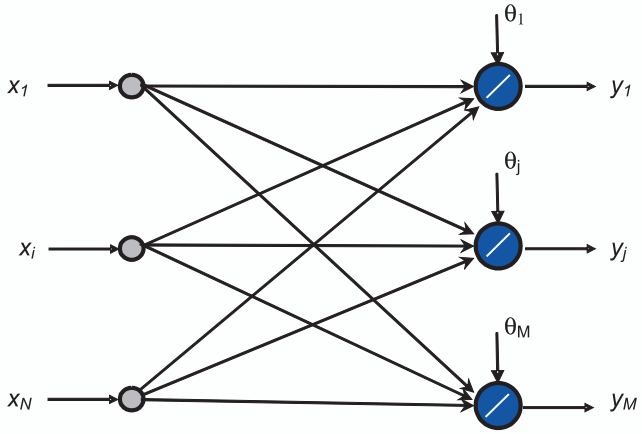


Fig. 2.6 Estructura de una red tipo ADALINE

Algoritmo de aprendizaje

Supongamos que nos encontramos en un sistema montañoso muy complejo en un lugar a gran altura y totalmente cubierto por neblina. Estamos perdidos, pero sabemos que en el valle hay una población donde podremos obtener ayuda, ¿qué estrategia utilizamos para llegar a dicha población? Muy seguramente estamos de acuerdo en que lo mejor es empezar a bajar y siempre mantener esta tendencia, por ejemplo, podríamos seguir el cause de un río, pues esta corriente de agua nos garantiza que siempre caminaremos paulatinamente hacia un lugar más bajo.

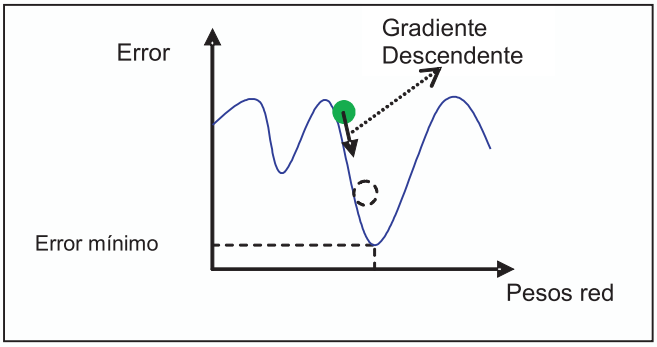


Fig. 2.7 Gradiente Descendente

Este es el principio del algoritmo utilizado en el proceso de aprendizaje de la red ADALINE, en donde, partiendo desde un punto aleatorio en la superficie de error, buscaremos un punto donde el error sea mínimo, siguiendo una trayectoria descendente, como se ilustra en la figura 2.7. Esta técnica de búsqueda se denomina Algoritmo de Gradiente Descendente.

Si recordamos del capítulo 1, la ecuación de actualización de pesos para una red neuronal la podemos definir en términos del peso en el instante *t* y de la variación del peso sináptico,

$$w(t+1) = w(t) + \Delta w(t) \tag{2.11}$$

donde,

$w(t+1)$	:	Valor actualizado del peso sináptico
$w(t)$	:	Valor actual del peso sináptico
$\Delta w(t)$	:	Variación del peso sináptico

Si variamos los pesos de la red en un factor proporcionar al gradiente negativo del error de la red respecto a los pesos, logramos que los pesos

así generados disminuyan el índice de desempeño (J) que para este caso lo definimos en términos del error cuadrático promedio en la ecuación 2.12 y para una red ADALINE de una neurona de salida, con P patrones de entrenamiento.

$$J = \frac{1}{2P}\sum_{p=1}^{P}(d_p - y_p)^2 \tag{2.12}$$

p : *p-ésimo* patrón del conjunto de entrenamiento
d_p : Valor deseado de salida para el *p-ésimo* patrón
y_p : Valor de salida de la red ADALINE para el *p-ésimo* patrón

El objetivo de algoritmo es encontrar el conjunto de pesos que hagan que J sea mínimo; si hacemos la variación de los pesos proporcional al gradiente negativo, el cambio en el peso de la *i-ésima* conexión la representamos por la ecuación 2.13, donde α es el factor de aprendizaje de la red.

$$\Delta w_i = -\alpha \frac{\partial J}{\partial w_i} \tag{2.13}$$

Si en la ecuación anterior aplicamos la regla de la cadena para obtener la derivada, queda en términos de la derivada del índice de desempeño respecto del error, la derivada del error respecto de la salida de la red ADALINE y la derivada de la salida de la red respecto del peso de la *i-ésima* conexión.

$$\Delta w_i = -\alpha \frac{\partial J}{\partial e}\cdot\frac{\partial e}{\partial y}\cdot\frac{\partial y}{\partial w_i} \tag{2.14}$$

Tras evaluar las derivadas parciales de la ecuación 2.14, obtenemos los siguientes valores,

$$\frac{\partial J}{\partial e} = e = (d_p - y_p)$$
$$\frac{\partial e}{\partial y} = -1$$
$$\frac{\partial y}{\partial w_i} = x_i$$

Estos valores de las derivadas parciales los reemplazamos en la ecuación 2.14 para obtener el valor final de la variación del el peso de la *i-ésima* conexión.

$$\Delta w_i = -\alpha e_p(-1)x_i$$
$$\Delta w_i = \alpha e_p x_i \tag{2.15}$$

Retomando la ecuación 2.11 y 2.15, la expresión para la actualización del peso de la *i-ésima* conexión está dada por la ecuación 2.16.

$$w_i(t+1) = w_i(t) + \Delta w_i$$
$$w_i(t+1) = w_i(t) + \alpha e_p x_i \tag{2.16}$$

Con base en las ecuaciones obtenidas, proponemos un algoritmo de aprendizaje para la red ADALINE, cuyos pasos los describimos a continuación.

Paso 1
Asignamos los valores iniciales de forma aleatoria a los pesos w_{ji} y el umbral θ_j.
Asignamos un valor al parámetro de aprendizaje α.
Definimos un valor para el error mínimo aceptado en el entrenamiento de la red.

Paso 2
Mientras la condición de parada sea falsa, ejecutamos los pasos 3 al 7.

Paso 3.
Para cada patrón de entrenamiento $\{\boldsymbol{x_p}, \boldsymbol{y_p}\}$ ejecutamos los pasos 4 al 5

Paso 4.
Calculamos la salida de la red ADALINE

$$y_{pj} = Neta_j = \sum_{i=1}^{N} w_i x_{pi} + \theta_j \tag{2.17}$$

Paso 5.
Calculamos el error e_p en la salida de la red ADALINE

$$e_p = \frac{1}{2}\sum_{j=1}^{M}(d_{pj} - y_{pj})^2 \tag{2.18}$$

Paso 6.
Actualizamos los pesos

$$w_i(t+1) = w_i(t) + \alpha e_p x_i \qquad (2.\ 19)$$

$$\theta_j(t+1) = \theta_j(t) + \alpha e_p \qquad (2.20)$$

Paso 7.
Calculamos el error global de la red y si es menor que un valor mínimo detenemos el algoritmo. En caso contrario, retornamos al paso 2.

Limitaciones del perpectrón

Como hemos mencionado a lo largo de este capítulo, uno de los principales inconvenientes que tiene el Perceptron es su incapacidad para separar regiones que no sean linealmente separables, como se observa en la figura 2.8. Sin embargo, Rosenblatt ya intuía que un Perceptron multicapa sí podía solucionar este problema pues, de esta manera, se podían obtener regiones de clasificación mucho más complejas, debido a una estructura, como la de la figura 2.9, que incluye además de las capas de entrada y salida, una o más capas internas u ocultas.

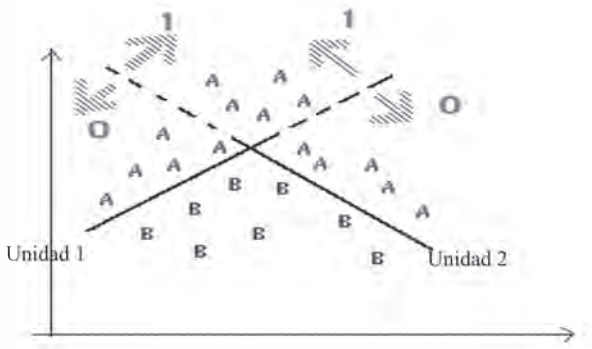


Fig. 2.8 Regiones de separación lineales que genera un Perceptron

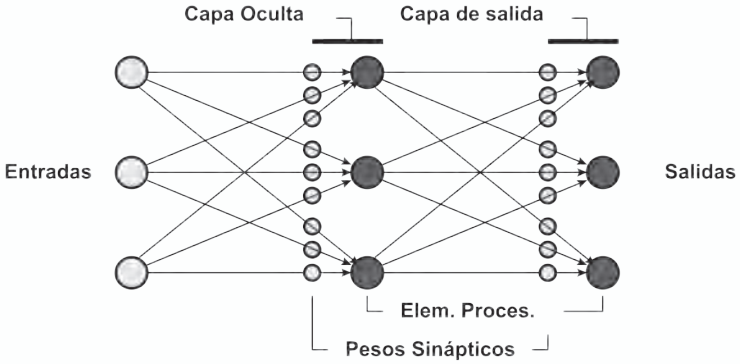
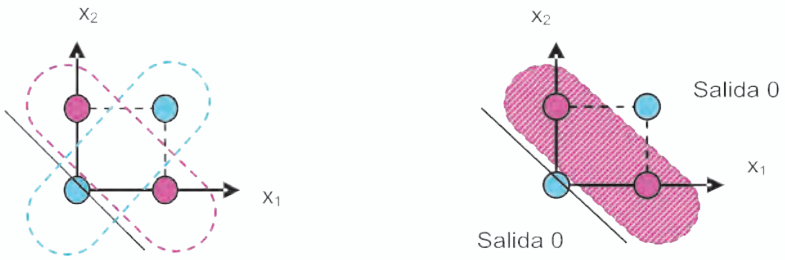


Figura 2.9 Estructura de una Red Neuronal Multicapa

Para ilustrar mejor este planteamiento observe la figura 2.10, donde se muestra que un Perceptron no está en capacidad de solucionar la función lógica XOR, descrita en la Tabla 2.1, ya que no es posible separar linealmente su salidas. Esta situación nos permite comprender de alguna manera los planteamientos de Minsky y Paper, pues se muestra con un problema relativamente sencillo las limitaciones de este tipo de red neuronal.

Tabla 2.1 Función Lógica XOR

Patrón	x_1	x_2	*X OR*
$\mathbf{x^1}$	0	0	0
$\mathbf{x^2}$	0	1	1
$\mathbf{x^3}$	1	0	1
$\mathbf{x^4}$	1	1	0



a) Representación en el espacio de entradas para la función XOR

b) Clasificación sin separación lineal

Fig. 2.10 Solución de la XOR con un Perceptron

Una arquitectura de Perceptron multinivel con dos neuronas en el nivel de entradas, una única capa oculta con dos neuronas y la respectiva neurona de la capa de salida, como se ilustra en la siguiente figura 2.11, con adecuado entrenamiento puede resolver el problema de separación no lineal planteado por la función XOR.

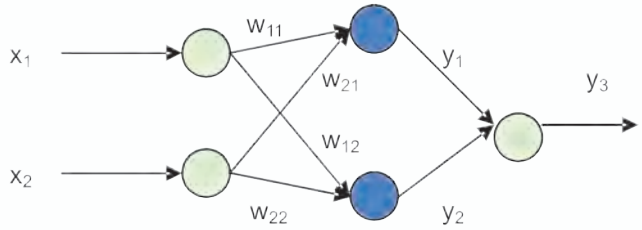


Fig. 2.11 Perceptron multicapa propuesto para la solución la función XOR

Lo anterior es cierto y ya lo vislumbraba Rosenblatt, pero en ese entonces se tenían algunos interrogantes sin respuesta ¿Cómo entrenar un Perceptron multicapa? ¿Cómo evaluar el error en las capas ocultas si no hay un valor deseado conocido para las salidas de estas capas?

Este inconveniente es inherente a la forma como opera el algoritmo de aprendizaje del Perceptron pues, para poder variar los pesos, lo hace con base en el error de salida y, por tanto, se necesita conocer la salida deseada de la capa a la cual se la va a realizar dicha operación. En el Perceptron sin capas ocultas, este problema no existe pues al ser el algoritmo de aprendizaje del Perceptron del tipo supervisado, las salidas de la red están definidas. Si el Perceptron es multicapa es imposible conocer con anterioridad el valor de la salida de una de las capas ocultas, motivo por el cual el algoritmo de aprendizaje que posee el Perceptron se hace inutilizable en este tipo de redes neuronales.

Una solución a este problema la planteó formalmente, y por primera vez,Werbos, y se denominó algoritmo de aprendizaje Backpropagation que se estudiará en el siguiente capítulo.

Aproximación práctica

Construcción de un perceptron usando MATLAB®

En este proyecto proponemos generar un Perceptron de dos entradas usando los bloques de *Simulink* de la herramienta de redes neuronales del MATLAB®. Usando los bloques de *Simulink* de esta aplicación como presentamos en la figura 2.12, podemos representar el funcionamiento básico de un Perceptron. En el diagrama de simulación, el bloque *dotprod* está en la biblioteca *weight functions* y el bloque *hardlim* está en la biblioteca

transfer functions.

Para simular los pesos y la entrada se usan bloques *constant* (biblioteca *sources*) y para visualizar la salida de la red y la neta se usan bloques *display* (biblioteca *sinks*).

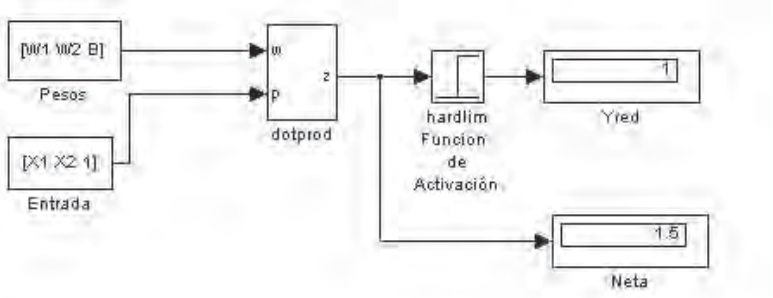


Fig. 2.12 Simulación de un Perceptron de dos entradas en Simulink

Con el diagrama construido es posible simular otros tipos de redes neuronales sencillas, por ejemplo, cambiando la función de activación a lineal, sigmoidal o tangente sigmoidal.

5.2 Solución de la función lógica and con un perceptron

Como vimos anteriormente, el Perceptron es una red neuronal artificial que está en capacidad de realizar separaciones lineales; veamos entonces, como se puede solucionar un problema de estos con la ayuda de la herramienta de redes neuronales de MATLAB®.

La función lógica AND se define como:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Sigamos cuidadosamente los siguientes pasos y podremos resolver este problema usando MATLAB®:

Definición del problema

Definir el problema que una red neuronal va a resolver es proporcionarle a la misma un conjunto de parejas ordenadas de entradas y salidas para que la red "*aprenda"* los llamados patrones de entrenamiento o aprendizaje de la red. En MATLAB® esto se hace definiendo dos matrices una para las entradas y otra para las salidas donde cada patrón de aprendizaje se define por columnas, los comandos necesarios para lo anterior son:

```
>> % Definición de la función lógica AND
>> X=[0 0 1 1 ;
      0 1 0 1 ] ;
>> D=[0 0 0 1] ;

>>%Para ver la gráfica de estos patrones se usa el comando
plotpv

>> plotpv(X,D)
```

Al ejecutar esta serie de comandos usted podrá obtener como resultado algo similar a lo presentado en la figura 2.13.

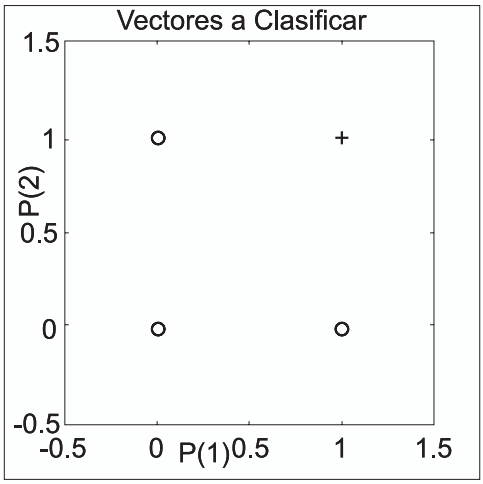


Fig. 2.13 Patrones a clasificar

Como se puede observar en la figura 2.13, MATLAB® grafica los puntos dados en el vector *X* y le asigna un símbolo para la clasificación dependiendo de la salida deseada, en este caso:

- Para salida deseada cero (0): **o**
- Para salida deseada uno (1): **+**

Inicialización de la red neuronal

Con base en los patrones de entrenamiento que definen el problema a resolver, procedemos a inicializar la red neuronal. Para el caso del Perceptron utilizamos la función de MATLAB® *initp.* En el comando de ayuda de MATLAB® *(help)* podemos encontrar una descripción completa de cada una de las funciones utilizadas.

```
% Generación de un perceptron

>> red = newp([0 1;0 1],1)
```

donde,

red	:	Objeto donde se va almacenar la red creada por el MATLAB®
[0 1;0 1]	:	Rango del valor de la entrada de la red neuronal, el número de filas de esta matriz lo utilizará MATLAB® para definir el número de entradas que tiene la red neuronal.
1	:	Número de neuronas que tiene la red neuronal en la capa de salida

Ahora procedemos a generar una matriz de pesos iniciales para la red. Este paso no es indispensable, pero permite generar un Perceptron con una superficie de separación conocida.

```
>> red.iw{1,1}=[1 1];
>> red.b{1}=0.5;
>> Pesos=red.iw{1,1};
>> Bias=red.b{1};

>> % Comando para graficar la línea que el Perceptron define
para separar las regiones
>> plotpc(Pesos,Bias)
```

Con el último comando ejecutado, adicionamos la recta de separación de las regiones de clasificación que se tiene en un instante determinado de acuerdo con los pesos de la red en ese momento. Tal como lo mostramos en la figura 2.14, la clasificación no es correcta, pues los parámetros que definen la recta fueron asignados de manera arbitraria.

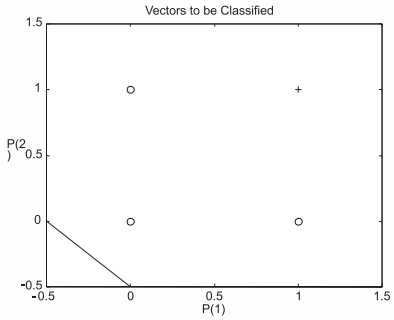


Fig. 2.14 Patrones a clasificar y la recta clasificadora inicial

Entrenamiento de la red

El entrenamiento de la red lo realizamos con el comando *train* el cual se implementa con la regla de aprendizaje tipo Perceptron. En MATLAB® el entrenamiento se realiza ejecutando el siguiente comando:

```
>> red = train(red,X,D)
```

donde,

red	:	Red a ser entrenada por el MATLAB®.
X	:	Vector con los patrones de entrada para el aprendizaje.
D	:	Vector con los patrones de salida deseada para el aprendizaje.

Cuando ejecutamos este comando en MATLAB®, iniciamos el entrenamiento y éste grafica la evolución del error al progresar las iteraciones, tal como se muestra en la figura 2.15.

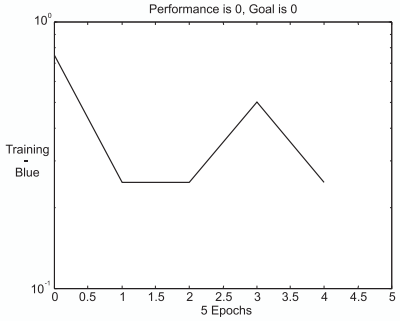


Fig. 2.15 Evolución del error durante el entrenamiento de la red

Una vez entrenada la red, ejecutamos los siguientes comandos para visualizar en la figura 2.16, la recta que separa las dos clases de salidas, comprobando con ello que la red ha realizado correctamente la tarea.

```
>> figure;
>> Pesos=red.iw{1,1};
>> Bias=red.b{1};
>> plotpv (X,D)
>> plotpc (Pesos, Bias)
```

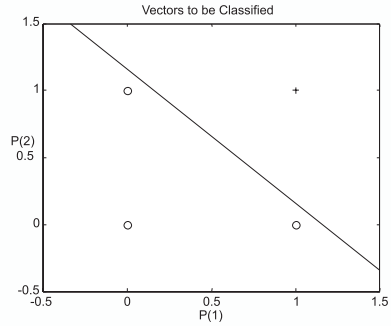


Fig. 2.16 Patrones a clasificar y la recta clasificadora final

Validación de la red

Luego de tener una red entrenada, procedemos a validar si el comportamiento de la misma es correcto o no, usando el comando *sim*. Podemos corroborar que el comportamiento de la red es el adecuado y dar por finalizado el entrenamiento.

```
>> in_prueba=[0;0];          % Patrón de prueba
>> % Prueba de la red ante el patrón de prueba, W,b son los pesos
>>% y el bias de la red entrenada

>> a = sim(red, in_prueba)
>>a =
>> 0

>> % Otro patón de prueba
>> in_prueba=[1;1];
>> a = sim(red, in_prueba)
>>a =
>>1
```

Exportando la red neuronal a simulink

En la sección anterior entrenamos y validamos nuestra primera red neuronal artificial, en particular un Perceptron. MATLAB® almacena esta red como un bloque funcional que puede ser exportado al ambiente de trabajo de ***simulink*** para verificar el comportamiento de la misma de una manera completamente gráfica. Para llevar a cabo esta tarea usamos el comando *gensim* y la herramienta nos responde en su ambiente gráfico con el diagrama de bloques de la figura 2.17.

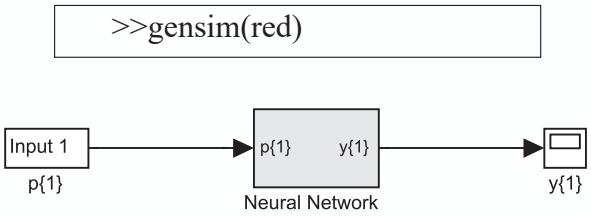


Fig. 2.17 Red neuronal generada en el Simulink

MATLAB® nos entrega el diagrama de la figura 2.17, pero lo podemos modificar como en la figura 2.18, separando las dos entradas y, con este nuevo esquema, procedemos a simular el comportamiento de la red neuronal diseñada. El valor de los bloques de entrada se puede modificar para verificar el comportamiento de la red ante las diferentes combinaciones de la función lógica AND.

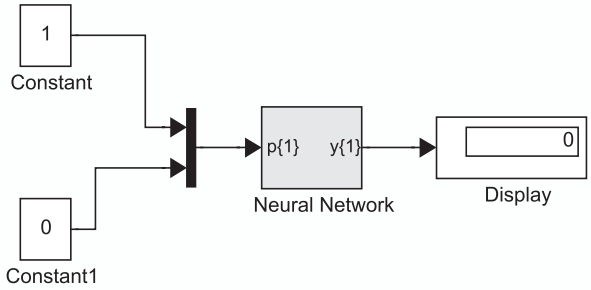


Fig. 2.18 Diagrama para verificar el comportamiento de la red neuronal

Solución de la función lógica and con uv-srna

En el Grupo de Percepción y Sistemas Inteligentes de la Escuela de Ingeniería Eléctrica y Electrónica, se desarrolló la herramienta UV-SRNA, para el diseño y simulación de Redes Neuronales Artificiales. Esta aplicación junto con su herramienta se ayuda se entrega en el CD disponible con este libro.

Para iniciar el proceso de aprendizaje con UV-SRNA, debemos crear un

archivo texto con los patrones que se desean usar para el proceso de entrenamiento que se almacenan con la extensión **.pat*, que para el caso de la función lógica **AND,** el archivo puede tener la estructura que definimos en la Tabla 2.1.

Tabla No. 2.2 Codificación de los patrones de entrenamiento para la función lógica AND

Datos en el archivo	Significado
4	Número de patrones de entrenamiento
2	Número de entradas de cada patrón
1	Número de salidas de cada patrón
0 0 0	Patrón No. 1
0 1 0	Patrón No. 2
1 0 0	Patrón No. 3
1 1 1	Patrón No. 4

Una vez creado el archivo, lo leemos desde UV-SRNA usando en la interfaz gráfica la opción *leer patrones* del menú de *archivo.*

Con los patrones en el ambiente de UV-SRNA procedemos a inicializar la red, para esto solo es necesario oprimir el botón de *inicializar*, de la interfaz gráfica.

Una vez inicializada la red procedemos a llevar a cabo el entrenamiento. Esto se logra oprimiendo el botón *entrenar*, e inmediatamente observamos la evolución del error con una clara tendencia decreciente, hasta que converge a cero.

Para finalizar, verifiquemos si el entrenamiento de la red fue adecuado, utilizando en la interfaz de usuario la opción de validación, que se puede hacer directamente por teclado o por archivo previamente construido.

Clasificador lineal con uv-srna

En el ejercicio anterior construimos un Perceptron completamente y lo aplicamos en un problema de clasificación de patrones, ahora, ejecutemos una demostración que nos permite resolver problemas de clasificación lineal en un conjunto de puntos en un plano. Como ejemplo típico, nuevamente usaremos el problema de la función lógica AND. Para ello debemos ejecutar la demostración que está dentro de las "demos" del Perceptron de UV-SRNA y cargar el conjunto de patrones disponible para la función AND.

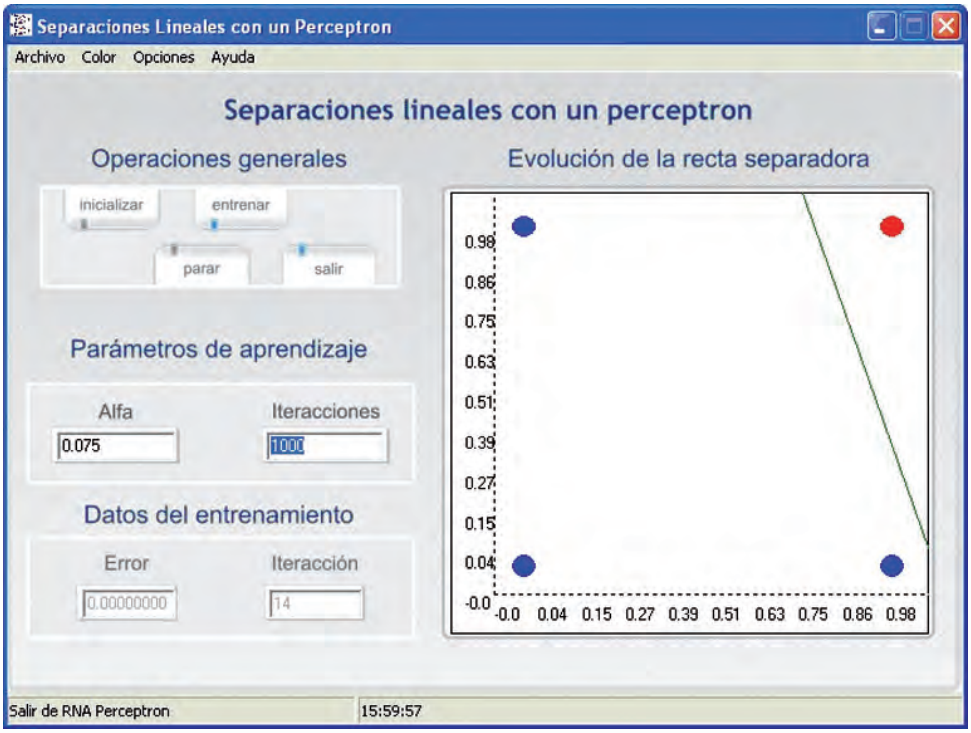


Fig. 2.19 Interfaz gráfica con el usuario del módulo de separaciones lineales de UVSRNA

Al oprimir el botón "inicializar" observemos la recta generada por el Perceptron no clasifica correctamente los datos, ver figura 2.19, pues la asignación de pesos de la red en esta fase es aleatoria. Cuando oprimimos el botón "entrenar", la recta va cambiando a medida que transcurre el proceso de aprendizaje hasta que se consiga separar adecuadamente los datos.

Un ejercicio interesante, es verificar el efecto que tiene el valor de α en la velocidad de convergencia de la red. Evaluemos qué efectos tiene el realizar este experimento con valores de α igual a 0.07 y 0.007. Notaremos que en el segundo caso el proceso de aprendizaje es mucho más lento.

Reconocimiento de caracteres usando el perceptron

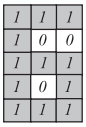
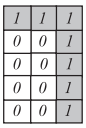
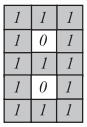
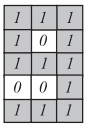
El objetivo de este proyecto es el de entrenar un Perceptron en MATLAB® que esté en capacidad de identificar los números del 0 al 9 donde cada número se define con una matriz de dimensión 5x3. Por ejemplo el número 2 sería representado así,

1	1	1
0	0	1
1	1	1
1	0	0
1	1	1

Para resolver el problema, verifiquemos todos los patrones correspondientes a cada número y asignémosle el respectivo valor de salida. En nuestro caso, definiremos por patrón un vector de entrada de 15 elementos y un vector de salida de 4 elementos, correspondientes al número binario equivalente. El vector de patrones de entrada queda definido como se muestra en la tabla 2.3, donde cada número decimal se codifica con un vector de 15 bits.

Tabla 2.3 Vector de Patrones correspondiente a los caracteres decimales

Caracter	**Vector de Patrones**														
	B_1	**B_2**	**B_3**	**B_4**	**B_5**	**B_6**	**B_7**	**B_8**	**B_9**	**B_{10}**	**B_{11}**	**B_{12}**	**B_{13}**	**B_{14}**	**B_{15}**
1 1 1 1 0 1 1 0 1 1 0 1 1 1 1	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
1 1 1 0 0 1 1 1 1 1 0 0 1 1 1	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1
1 1 1 0 0 1 0 1 1 0 0 1 1 1 1	1	1	1	0	0	1	0	1	1	0	0	1	1	1	1
1 0 1 1 0 1 1 1 1 0 0 1 0 0 1	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1
1 1 1 1 0 0 1 1 1 0 0 1 1 1 1	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1

	1	1	1	1	0	0	1	1	1	1	0	1	1	1	1
	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1
	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1

La salida deseada de entrenamiento de la red la representaremos con el código binario del número que le presentamos a la entrada. Es decir, si a la red le llegan como entrada los 15 bits correspondientes al número 0, esperamos que la red genere el código binario del cero expresado en 4 bits (0 0 0 0); o en otro caso, si le llegan como entrada los 15 bits correspondientes al número 1, ella generará el código binario del uno expresado en 4 bits (0 0 0 1) y así con el resto de patrones.

Teniendo en cuenta lo anterior, presentamos en la tabla 24, los patrones de entrenamiento para los números decimales y, en seguida, el código en MATLAB® para entrenar la red neuronal.

Tabla 2.4 Codificación de los patrones de entrenamiento para el reconocimiento de caracteres decimales en MATLAB®

	Patrón de Entradas X															Patrón Salidas D			
	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	$\mathbf{B_{10}}$	$\mathbf{B_{11}}$	$\mathbf{B_{12}}$	$\mathbf{B_{13}}$	$\mathbf{B_{14}}$	$\mathbf{B_{15}}$	$\mathbf{2^3}$	$\mathbf{2^2}$	$\mathbf{2^1}$	$\mathbf{2^0}$
0	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1
2	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1	0	0	1	0
3	1	1	1	0	0	1	0	1	1	0	0	1	1	1	1	0	0	1	1
4	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1	0	1	0	0
5	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1	0	1	0	1
6	1	1	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1	1	0
7	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	1
8	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	0	0
9	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1	1	0	0	1

```
% Rec_Carac.m
% Programa que reconoce los caracteres decimales

% Patrones de entrada con los valores que debe tener X
Xaux=[1 1 1 1 0 1 1 0 1 1 0 1 1 1 1;
      0 1 0 0 1 0 0 1 0 0 1 0 0 1 0;
      1 1 1 0 0 1 1 1 1 1 0 0 1 1 1;
      1 1 1 0 0 1 1 1 1 0 0 1 1 1 1;
      1 0 1 1 0 1 1 1 1 0 0 1 0 0 1;
      1 1 1 1 0 0 1 1 1 0 0 1 1 1 1;
      1 0 0 1 0 0 1 1 1 1 0 1 1 1 1;
      1 1 1 0 0 1 0 0 1 0 0 1 0 0 1;
      1 1 1 1 0 1 1 1 1 1 0 1 1 1 1;
      1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 ];
X=Xaux';
% Salida deseada de la red neuronal
Daux=[0 0 0 0;
      0 0 0 1;
      0 0 1 0;
      0 0 1 1;
      0 1 0 0;
      0 1 0 1;
      0 1 1 0;
      0 1 1 1;
      1 0 0 0;
      1 0 0 1 ];
 D=Daux';
red=newp(minmax(X),4);                % Definición de la red
red.iw{1,1}=rand(4,15);               % Definición aleatoria de los
                                      pesos
red.b{1}=rand(4,1);
red.trainParam.show=1;                % Evolución de la red en
                                      cada iteración
disp('los pesos inciales son:')       % Pesos asignados por la
                                      función random
Pesos=red.iw{1,1}
Bias=red.b{1}
pause(2);
red = train(red,X,D)                  % Entrenamiento de la red
disp ('para validar la red, digite el % Validación de la red, se pide
 vector de patrones de entrada')      digitar un patrón
                                      % para verificar el funcionamiento
                                      de la red
```

```
disp('Número de 10 binarios entre [ ]')
X1=input('X1=')              % Patrón de entrada
Y = sim(red, X1');           % Simulación de la red

disp ('el número resultante, en binario, leído de arriba para abajo es:')
Y
```

Como ejemplo presentemos dos casos particulares, en donde en el primero introducimos como valor de validación el número 7 y, en el segundo, el número 9. Para cada caso presentamos la gráfica de evolución en el entrenamiento y el patrón introducido para el proceso de validación.

Primera ejecución

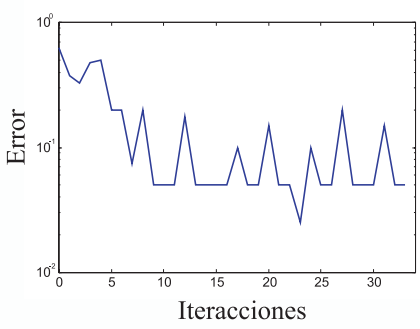
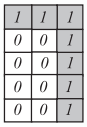


Fig. 2.20 Evolución del error de entrenamiento

Patrón de Entrada Para Validación	Vector de Entrada X1	Valor de Salida Y
	**[1 1 1 0 0 1 0 0 1 0 0 1 0 0 1]**	**[0 1 1 1] → 7**

Segunda ejecución:

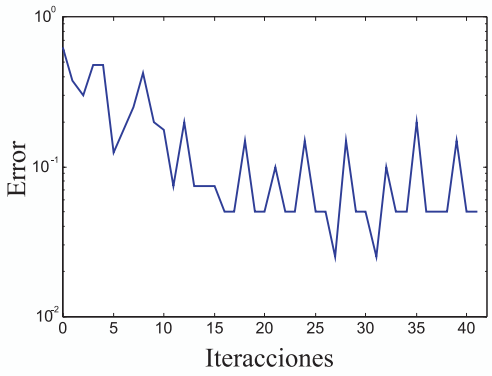
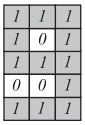


Fig. 2.21 Evolución del error de entrenamiento

Patrón de Entrada Para Validación	Vector de Entrada X1	Valor de Salida Y
	**[1 1 1 1 0 1 1 1 1 0 0 1 1 1 1]**	**[1 0 0 1]→ 9**

Teniendo la red entrenada, se puede exportar a la herramienta de simulación Simulink® asociado a MATLAB®, para verificar su comportamiento de una manera completamente gráfica como se observa en la figura 2.22.

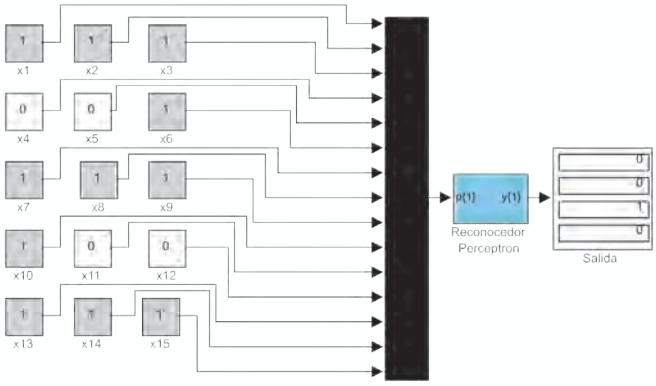


Fig. 2.22 Reconocedor de Caracteres Implementado en Simulink®

Reconocimiento de caracteres con uvsrna

En UV-SRNA se ha diseñado un módulo especial para verificar la capacidad de un Perceptron en el reconocimiento de caracteres, que se encuentra dentro de las demostraciones del Perceptron, cuya interfaz con el usuario se muestra en la figura 2.23

Fig. 2.23 Interfaz de Usuario del Módulo de Reconocimiento de Caracteres de UVSRNA

Reconocimiento de los números del 0-9 con uvsrna

Con esta aplicación podemos realizar el reconocimiento de los números del 0 al 9. En primera instancia creamos un archivo texto con los patrones que se desean usar para el proceso de entrenamiento, que debe almacenarse con la extensión *.pat., siguiendo el formato que se presenta en la tabla 2.5.

Tabla No.2.5 Ejemplo de codificación de los patrones de entrenamiento para el reconocimiento de números

Datos en el archivo	Significado
10	Número de patrones de entrenamiento
35	Número de entradas de cada patrón
4	Número de salidas de cada patrón
1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0	Patrón No. 1. Que corresponde al número cero
0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1	Patrón No. 2 Que corresponde al número uno
1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 0 0 1 0	Patrón No. 3 Que corresponde al número dos

Una vez creado el archivo, lo cargamos desde UV-SRNA usando la opción "*leer patrones*", para luego inicializar la red y proceder a su entrenamiento. Emerge la ventana de la figura 2.24 que nos muestra como evoluciona el error y, muy seguramente, alcanzaremos un valor igual a cero, considerando así que el entrenamiento se ha finalizado.

Fig. 2.24 Evolución del error de entrenamiento en UVSRNA

Para validar el entrenamiento de la red, el módulo de reconocimiento de caracteres de UVSRNA dispone de una matriz 7x5 para dibujar los caracteres que vamos a reconocer. Una vez dibujado el caracter, presionamos el botón *"validar"* y la red toma esta entrada y esperamos que la salida, en binario, corresponda al dígito seleccionado.

Filtro adaptativo usando una red adaline

En la figura 2.25 vemos un problema clásico en procesamiento digital de señales, donde una fuente de ruido interfiere con la señal que contiene la información válida para la aplicación. La solución implica el diseño de un filtro para eliminar la señal no deseada. La red ADALINE puede usarse como filtro y por su capacidad de aprendizaje, puede adaptarse progresivamente a medida que va recibiendo la señal y de esa manera elimina el ruido indeseado y nos entrega una señal limpia, sin interferencias.

Fig. 2.25 Figura Esquema de Filtrado adaptativo con una red adaline

La arquitectura general de la red *ADALINE* utilizada para filtrar esta señal, figura 2.26, toma la *k-ésima* muestra de la señal contaminada y *N* muestras anteriores.

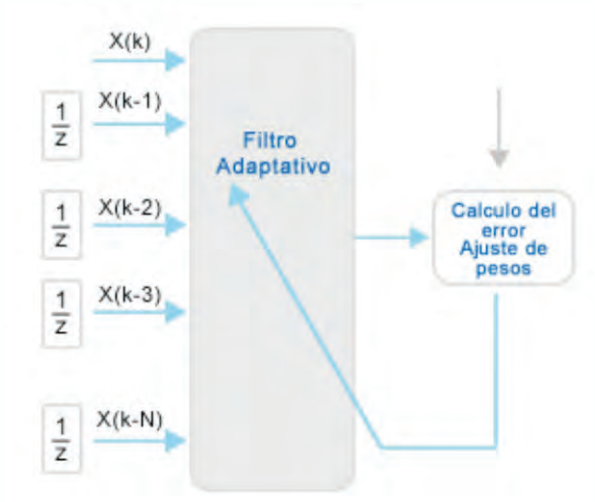


Fig. 2.26 Esquema de un Filtro Adaptativo

En la figura 2.27, tenemos una señal senoidal contaminada con una señal de alta frecuencia, pretendemos con la red *ADALINE*, eliminar la señal de alta frecuencia. Para ello programamos en *MATLAB*® una red que tiene como entradas, la muestra *k-ésima* y cinco retardos más de la señal. Como el filtro es de naturaleza adaptativa, a medida que la señal se va presentando al mismo tiempo, éste va modificando sus pesos y la convergencia va mejorando paulatinamente, en las figura 2.28 se puede apreciar esta convergencia visualizando el desempeño del filtro durante la sintonización.

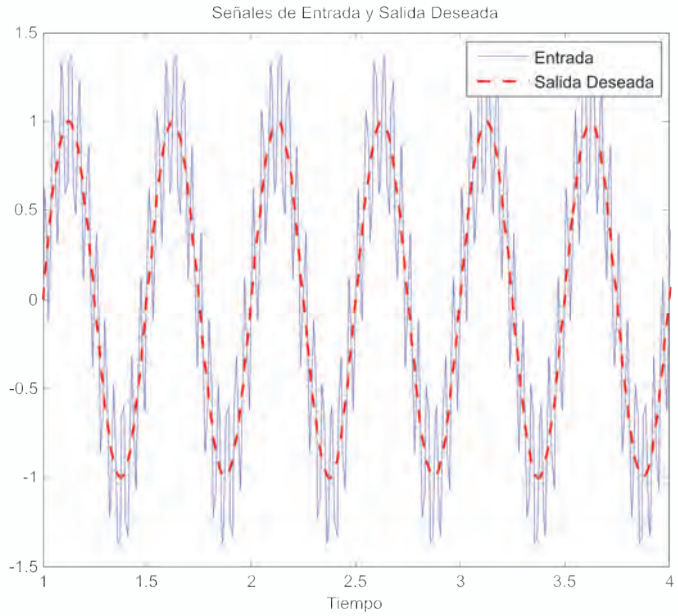


Fig. 2.27 Señal Contaminada con Ruido

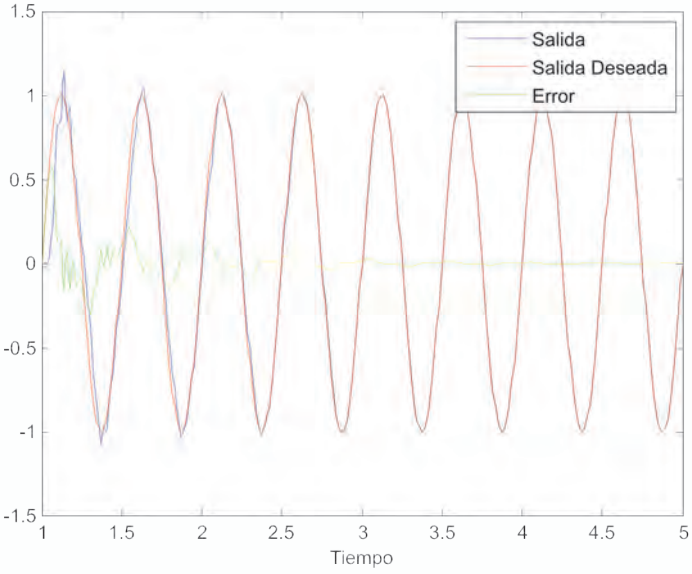


Fig. 2.28 Señal Filtrada y Evolución del Error

El código en MATLAB® que permite diseñar el filtro para la anterior aplicación es el siguiente:

```
time = 1:0.01:5;
X = sin(2*pi*2*time)+0.5*sin(2*pi*24*time);
P = con2seq(X);
Taux = sin(2*pi*2*time);
T = con2seq(Taux);
Figure
plot(time, cat(2,P{:}),time,cat(2,T{:}),'--')
title('Señales de Entrada y Salida Deseada')
xlabel('Tiempo')
legend({'Entrada','Salida Deseada'})
% Creamos una red ADALINE con 5 retardos
net = newlin([-2 2],1,[0 1 2 3 4 5],0.1);
net.biasConnect=0; % El peso de tendencia o umbral no se usa para
esta aplicación.

[net,Y,E,Pf]=adapt(net,P,T);
figure
plot(time,cat(2,Y{:}),'b', ...
   time,cat(2,T{:}),'r', ...
   time,cat(2,E{:}),'g')
legend ({'Salida','Salida Deseada','Error'})
```

Filtrado de señales biomédicas

Normalmente las señales biomédicas que provienen del cuerpo humano se encuentran contaminadas por una señal de ruido de 60Hz, producto de la inducción que hace el mismo cuerpo ante la presencia de los campos magnéticos producidos por las redes de distribución eléctrica. Por lo que es necesario un procesamiento automático que nos permita filtrar la señal de 60 Hz dejando la señal biomédica sin contaminación alguna, tal como vemos en la figura 2.29.

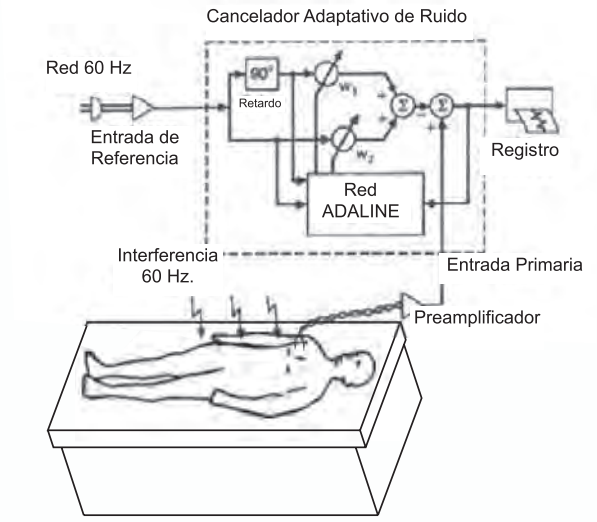


Fig. 2.29 Filtrado de una Señal biomédica

La aplicación que diseñamos y cuyo programa MATLAB®, *Filtrado_ Biomedico.m*, se encuentra en el CD que se adjunta a este libro, se utiliza para filtrar la señal proveniente de un electro cardiógrafo (ECG). En la figura 2.30, presentamos el esquema del filtro diseñado que elimina la componente inducida de 60 Hz., y en la figura 2.31 observamos las señales de entrada, salida deseada y error.

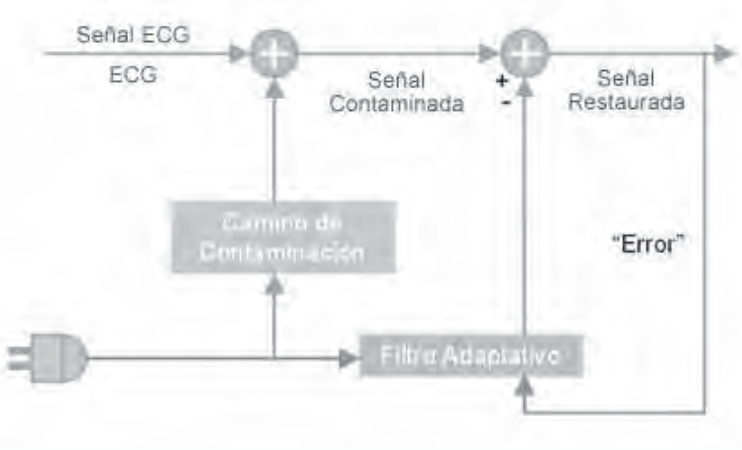


Fig. 2.30 Esquema del Filtrado planteado

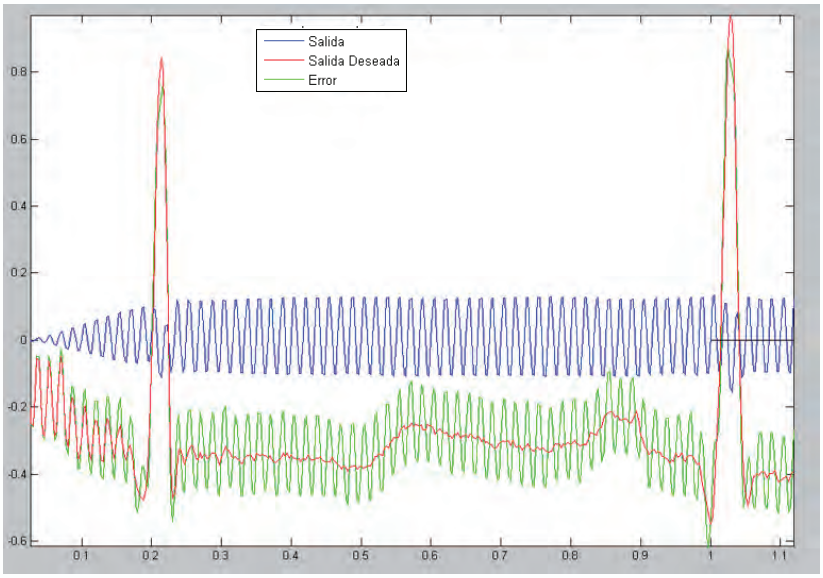


Fig. 2.31 Señales en el proceso de filtrado de una señal ECG

Filtrado de señales de voz

Cuando grabamos un concierto es imposible separar la música del ruido que hacen los espectadores, como resultado tenemos una señal de la voz del cantante contaminada con las voces y los aplausos de los asistentes. Esta fuente de ruido la podemos conocer, pero no sabemos como se ve afectada la señal de la voz del intérprete que queremos procesar. En la figura 2.32, presentamos un esquema con el que podemos eliminar el efecto del ruido en nuestra grabación utilizando un filtro adaptativo dise-ñado con una red ADALINE. Vale la pena destacar que la señal de error será la señal de la voz filtrada pues en este caso, la red ADALINE se encargará de aprender como la fuente de ruido afecto la voz original; cuando la red aprenda lo anterior, al restar de la señal de voz contaminada la salida de la red, tendremos la voz filtrada tal como lo observamos en la figura 2.33. La aplicación la diseña-mos para ser ejecutada en MATLAB®, *Filtrado_voz.m*, y se encuentra en el CD que se adjunta a este libro.

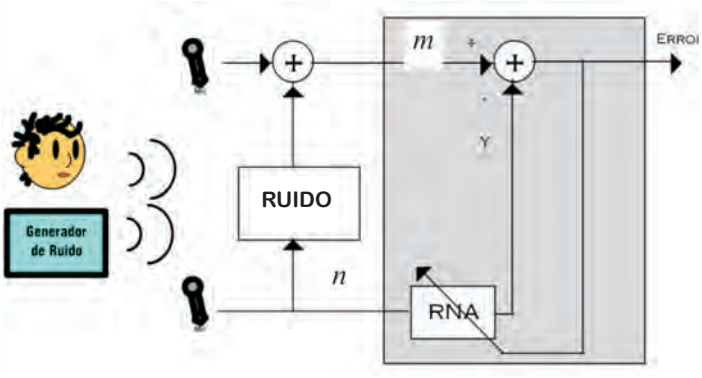


Fig. 2.32 Esquema para filtrar una señal de voz con una red ADALINE

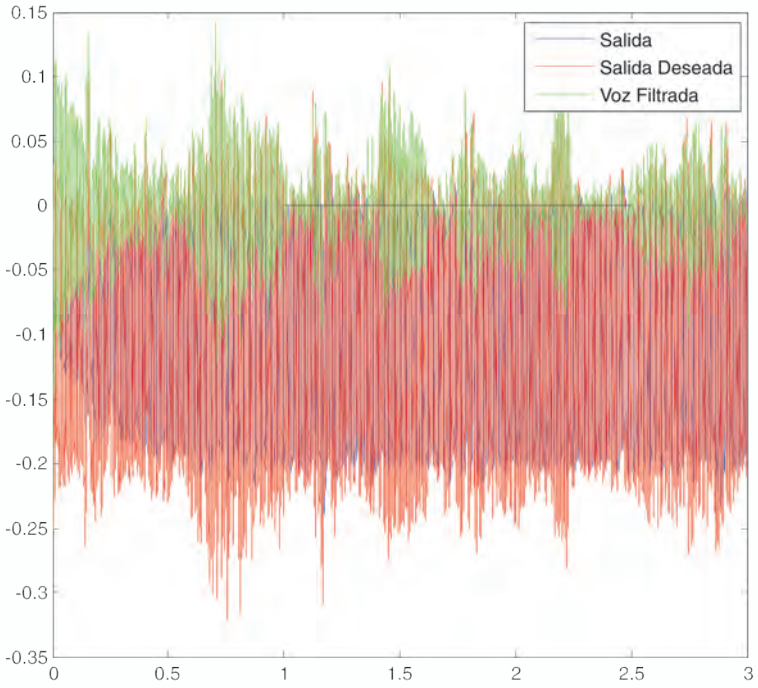


Fig. 2.33 Señales en el proceso de filtrado de voz

Proyectos propuestos

1. Realice la siguiente clasificación usando un Perceptron y con ayuda del *toolbox de redes neuronales* del MATLAB®.

X1	X2	D
-0.5	-1.0	0
1.0	1.0	1
1.0	0.5	1
-1.0	-0.5	0
-1.0	-1.0	0
0.5	1.0	1

2. Repita el ejercicio anterior con UV-SRNA.
3. Teniendo los siguientes puntos en el plano realice el procedimiento necesario tanto en MATLAB® como en UVSRNA para resolver la clasificación indicada. La clase A la puede codificar como 0 y la clase B la puede codificar como 1.

X	Y	Familia
-1	-1	A
-0.5	-0.5	A
-1	0	A
0	-1	A
1	1	B
0.5	0.5	B
1	0	B
0	1	B

4. Cree un programa en MATLAB® que implemente el algoritmo de entrenamiento tipo Perceptron para resolver el problema de la función lógica OR.
5. Intente en MATLAB® y en UVSRNA solucionar el problema de la función lógica XOR. ¿Qué conclusiones obtiene de los resultados alcanzados?.
6. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer las vocales.
7. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer el código hexadecimal.
8. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer las letras del nombre de algún integrante de su familia.
9. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer cuatro figuras geométricas sencillas (un cuadrado, un triángulo, un rectángulo y un rombo).