DISEÑO E IMPLEMENTACIÓN DE UN SOFTWARE DE RECONOCIMIENTO DE HUMANOS EN MOVIMIENTO EN AMBIENTES NO CONTROLADOS

FREDY MCNISH BERNAL

JUAN CARLOS CASTELLANOS BALDOVINO

UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR

FACULTAD DE INGENIERÍA Y TECNOLOGÍA DE SISTEMAS

CARTAGENA DE INDIAS

2006

DISEÑO E IMPLEMENTACIÓN DE UN SOFTWARE DE RECONOCIMIENTO DE HUMANOS EN MOVIMIENTO EN AMBIENTES NO CONTROLADOS

FREDY MCNISH BERNAL

JUAN CARLOS CASTELLANOS BALDOVINO

TRABAJO DE GRADO PRESENTADO PARA OPTAR AL TÍTULO DE INGENIERO DE SISTEMAS

DIRECTOR

DR. GONZALO PAJARES MARTINSANZ

CO-DIRECTOR

ING. GLORIA I. BAUTISTA LASPRILLA

UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR

FACULTAD DE INGENIERÍA Y TECNOLOGÍA DE SISTEMAS

CARTAGENA

2006

Cartagena de Indias D.T. y C., Enero 30 de 2006

Señores:

Comité Evaluador

Facultad de Ingeniería y Tecnología de Sistemas

Apreciados señores:

Por medio de la presente me permito informarles que el proyecto de grado titulado "**DISEÑO E IMPLEMENTACIÓN DE UN SOFTWARE DE RECONOCIMIENTO DE HUMANOS EN MOVIMIENTO EN AMBIENTES NO CONTROLADOS**" ha sido desarrollado de acuerdo a los objetivos establecidos.

Como autores del proyecto consideramos que el trabajo es satisfactorio y amerita ser presentado para su evaluación.

Atentamente,

Fredy McNish Bernal

Juan C. Castellanos Baldovino

Cartagena de Indias D.T. y C., Enero 30 de 2006

Señores:

Comité Evaluador

Facultad de Ingeniería y Tecnología de Sistemas

Apreciados señores:

Por medio de la presente me permito informarles que el proyecto de grado titulado "**DISEÑO E IMPLEMENTACIÓN DE UN SOFTWARE DE RECONOCIMIENTO DE HUMANOS EN MOVIMIENTO EN AMBIENTES NO CONTROLADOS**" ha sido desarrollado de acuerdo a los objetivos establecidos.

Como directora del proyecto considero que el trabajo es satisfactorio y amerita ser presentado para su evaluación.

Atentamente,

Ing. Gloria I. Bautista Lasprilla

Gónzalo Pajares Martinsanz

Nota de aceptación

Firma de presidente del jurado

Firma del Jurado

Firma del jurado

Cartagena, Enero 30 de 2006

DEDICATORIAS

A Dios, a mis padres y a mis hermanos.

Por estar ahí, educándome, aconsejándome.

Por ofrecerme sus sabios consejos en el momento oportuno.

Fredy McNish Bernal

A mi mama y mi papa por

apoyarme siempre y

hacer todo esto posible

Juan C. Castellanos Baldovino

AGRADECIMIENTOS

Primero, queremos agradecer a DIOS por guiarnos y permitirnos tener esta gran oportunidad para demostrar mediante este trabajo el esfuerzo que hemos realizado por lograr nuestros sueños.

Quisiéramos también darle las gracias al Dr. Gonzalo Pajares por brindar un apoyo incondicional a la investigación y por darnos la oportunidad de trabajar a su lado, de guiarnos y ayudarnos a través de todo el proyecto.

Gracias a la profesora Gloria Isabel Bautista por apoyarnos todo el tiempo con la investigación y dedicarnos tanto tiempo para revisar, corregir y complementar todos los documentos elaborados.

Y no podemos de dejar de agradecer a todas las personas que nos dieron su cooperación justo en el momento cuando lo necesitamos, ni a todas aquellas personas que directa o indirectamente nos apoyaron, ya que también fueron parte de esto y ayudaron a que esto fuera posible.

CONTENIDO

LISTA DE FIGURAS

Comentario [FMB1]: Se ha eliminado la pagina en blanco

RESUMEN

En este trabajo se propone un sistema automático de reconocimiento de humanos en movimiento basado en técnicas de visión artificial. El sistema apunta a servir como prototipo base para un sistema de seguridad, a partir del procesamiento de imágenes de video en un sitio determinado. Las imágenes son adquiridas periódicamente mediante una cámara la cual se controla por el mismo sistema. Luego, a través de la aplicación de diversas técnicas de procesamiento de imágenes, se obtiene información acerca de que objetos se encuentran en movimiento y se extraen las características relevantes de los objetos detectados para que, a través de un clasificador previamente escogido y probado, sea capaz de reconocer e identificar cuando un objeto presenta los patrones de figuras humanas.

El proyecto involucra algoritmos de procesamiento y tratamiento de imágenes, programación en lenguajes de alto nivel, manejo de cámaras de video por computador y reconocimiento de patrones por medio de redes neuronales.

PREFACIO

Una de las áreas que actualmente ha estado en continuo progreso ha sido el área de visión por computador, más aún durante la última década. La razón principal de este progreso, ha sido el esfuerzo de modernización que se ha venido realizando por parte de las industrias, quienes son las que más demandan este tipo de aplicaciones, y a esto se le puede sumar el aumento de las prestaciones del hardware y el carácter multidisciplinar de esta área (tiene aplicaciones en la robótica, en la medicina, en la cartografía, etc.).

Con sus resultados, dentro de los campos que han salido beneficiados encontramos el campo de la seguridad, el cual es un campo comercial muy demandado en estos tiempos y que ha encontrado en las videocámaras, debido a su bajo costo y fácil acceso, un muy buen aliado para extender su campo de acción. Debido precisamente a esto, las cámaras de vigilancia predominan en diversos tipos de establecimientos, las cuales capturan imágenes que son grabadas en cintas que generalmente son regrabadas al cabo de un tiempo o son almacenadas y archivadas. Luego cuando ha ocurrido algún delito, estas imágenes son consultadas para analizar los hechos, pero para ese entonces ya es tarde. En estos casos, lo ideal sería contar con un monitoreo y análisis continuo de las imágenes de manera permanente para que se pueda alertar a tiempo cualquier suceso sospechoso cuando aun hay posibilidades de evitarlo, sin embargo esto no es una solución muy viable.

La visión por computador propone métodos con los cuales se puede abordar este problema, sin embargo en nuestro país muy poco se han aprovechado. Por esto, lo que se pretende con este trabajo es hallar una posible solución a este problema, la cual puede ser un sistema basado en software, que haciendo uso de técnicas de visión por computador, sea capaz de analizar automáticamente las imágenes presentadas por las video-cámaras, reconocer figuras humanas y alertar por su aparición y movimiento.

Cabe mencionar que más que la realización del sistema, lo que se pretende es dar a conocer las diferentes técnicas y teorías que implican la realización de un sistema como este, para que a partir de estas, se pueda llegar a implementar dicho sistema. Además de esto, otro de los objetivos que se quiere es poner a disposición de futuros interesados en este tema, una base de conocimiento para el uso de técnicas de visión por computador en diversos tipos de situaciones.

Debido a que los temas que se tratarán en este documento abarcan técnicas de procesamiento de imágenes, es necesario tener previamente conocimiento sobre el procesamiento de imágenes. En este documento no abordamos estos temas debido a que, además de no ser el propósito de este trabajo, su contenido es tan amplio que implicaría la realización de otro libro. Como guía para el estudio y tratamiento de estos temas proponemos [01], [02] o [03].

CAPITULO 1. INTRODUCCIÓN

En el intento por dotar a las maquinas de un sistema de visión aparece el concepto de visión artificial. Uno de los principales intereses de la visión artificial es el procesamiento de los datos de una imagen para la percepción de estos por parte de las maquinas de manera autónoma.

La visión artificial es una ciencia que aún está en proceso de desarrollo, su verdadera potencialidad no se ha alcanzado todavía; aunque los investigadores han desarrollado potentes técnicas y procedimientos de gran valor práctico.

Muchas aplicaciones actuales emplean una combinación de redes neuronales y procesamiento digital de imágenes, para desarrollar poderosos sistemas de visión artificial, tales sistemas los encontramos hoy en día con mayor frecuencia en procesos de reconocimiento de iris y de huellas, en reconocimiento de ambientes para robots, y en muchas otras aplicaciones de última generación.

Además de los adelantos de esta ciencia, el hardware también ha venido experimentando grandes avances, hasta el punto en que hoy en día son de fácil acceso por su bajo costo. Dentro de este hardware encontramos las cámaras de video, que tienen una amplia demanda en el campo de la seguridad, ya que sirven para aumentar el campo de acción en la vigilancia. Debido precisamente a esto, las cámaras de vigilancia predominan en diversos tipos de establecimientos. Estas

a pesar de toda la ayuda que brindan, tienen sus desventajas y es el de encontrar el personal que se encargue de la supervisión y monitoreo de todas estas imágenes. Por lo tanto, muchas de estas imágenes solamente se capturan para ser grabadas y archivadas en cintas, para que, por si algo ocurre, quede la evidencia y se puedan analizar los hechos.

El caso ideal de esto sería contar con un monitoreo y análisis continuo de las imágenes de manera permanente para que se pueda alertar a tiempo cualquier suceso sospechoso cuando aun hay posibilidades de evitarlo. Por eso, a manera de solución, nuestro objetivo es realizar un sistema basado en software, que haciendo uso de técnicas de visión por computador, sea capaz de analizar automáticamente las imágenes presentadas por las video-cámaras, reconocer figuras humanas y alertar por su aparición y movimiento.

Para poder realizar este trabajo se estudiaron y se analizaron varias técnicas, tanto de procesamiento de imágenes como de reconocimiento de formas, los cuales son expuestos en este documento.

CAPITULO 2. EL PROBLEMA

Como se dijo anteriormente, lo que se desea hacer es una aplicación capaz de analizar automáticamente las imágenes capturadas desde una cámara de video, reconocer figuras humanas y alertar por su aparición y movimiento. Por eso hemos definido este proyecto como "reconocimiento de figuras humanas en movimiento". Ampliando un poco más esta idea, podemos decir que se hará un algoritmo para detectar objetos en movimiento dentro de una escena adquirida por una cámara de video estacionaria, un algoritmo de segmentación se usará para determinar si un píxel es estático o esta en movimiento, para cada píxel de la imagen y un clasificador que, en base a la figura que formen los píxeles a los que se les detectó movimiento, determine si es o no una figura humana.

Una parte importante de este sistema es la robustez de la clasificación de "lo que se mueve". Por esto, para alcanzar a realizar una aproximación óptima para esta tarea de extracción y clasificación de objetos fue necesario examinar varios algoritmos y técnicas para obtener los resultados deseados más por práctica de laboratorio que por teórica.

No sobra decir que realizaremos todo un *framework* que soporte a la aplicación y en el cual podamos realizar todas las pruebas además de que sirva para futuros usos.

CAPITULO 3. EL SISTEMA PROPUESTO

El reconocimiento de figuras humanas en movimiento tiene dos etapas importantes: La detección del movimiento y el reconocimiento de las figuras.

La detección del movimiento en secuencias de video se suele describir como un problema de investigación significativo y difícil [08]. La clasificación y el reconocimiento de objetos tampoco es una tarea sencilla. En estas etapas generalmente se incluyen procesos de aprendizaje, mediante técnicas que han sido diseñadas para este propósito (redes neuronales, redes de probabilidad, sistemas difusos,… [03]).

Una metodología muy aceptada y difundida para el abordaje de problemas de procesamiento de imágenes y reconocimiento de formas corresponde a la que se muestra en el diagrama de bloques de la Figura 3-1 (ver [01] p8).

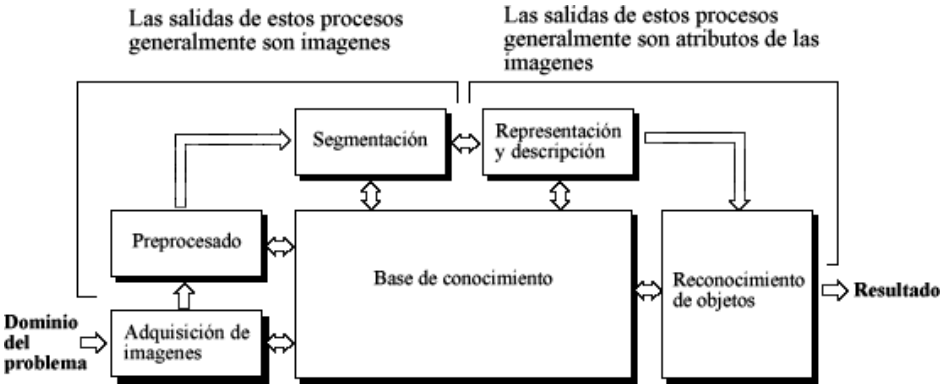


Figura 3-1. Etapas fundamentales del procesamiento de imágenes

Este diagrama muestra como a partir de un problema cualquiera se puede llegar a conseguir un resultado haciendo uso del procesamiento de imágenes. Esto no significa que todos los problemas se resuelvan siempre de esta forma, ya que para algunos será conveniente dividir o unir varias de estas etapas, haciendo que aparezcan o desaparezcan otras; pero de manera general el proceso es el mismo.

Adaptando el modelo anterior a nuestro caso, se decidió implementar el sistema dividiéndolo en 6 etapas, las cuales se muestran en la Figura 3-2.

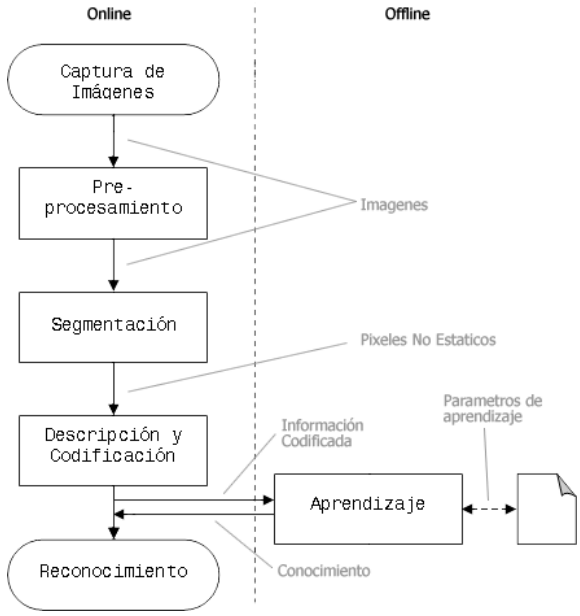


Figura 3-2. Implementación del sistema

A continuación se explican de manera general cada una de las etapas. Información mas detalla sobre las técnicas que se emplearon se presentarán en el Capitulo 4.

3.1 Captura de Imágenes

La primera etapa del proceso corresponde con la captura de imágenes, la cual consiste en adquirir y entregar al sistema una imagen digital. Para esto, se dispondrá de un componente encargado de la captura de imágenes, el cual estará compuesto de una cámara de video con interfaz USB y un conjunto de clases que la soportan (aunque lo que importa es que se pueda tener una imagen digital para ser procesada, con estos requerimientos hacemos que el sistema sea independiente del hardware de la cámara). Estas cámaras son capaces de capturar hasta 30 fps (*frames per second*, imágenes por segundo), el video es a color y soportan varios tipos de formatos (*RGB* y *CMY* son los mas comunes).

Además de la adquisición, este componente del sistema tiene como fin normalizar la entrada de video y es capaz de trabajar a un nivel intermedio entre la aplicación y el hardware. Dentro de las funciones que cumple se encuentran:

- Normalizar el flujo de entrada del video.
- Normalizar el formato de salida de las imágenes.
- Entregar la cantidad de imágenes necesarias para procesar.

De las características de este componente, tenemos:

- Es capaz de funcionar, sin necesidad de realizar ningún tipo de cambio, con varios tipos de cámaras de video que sean de interfaz USB. También será ampliable con el tiempo sin necesidad de cambios importantes.

- Soporte y recuperación de las fallas más comunes que se presentan en la adquisición de video.
- Fácil manipulación y configuración.

3.2 Pre-procesamiento

Una vez obtenida la imagen (o imágenes en este caso), se procede al *pre-procesamiento* de la imagen. La finalidad de esta etapa es mejorar la imagen de tal manera que sea más fácil de interpretar lo cual va desde hacer más evidente detalles escondidos o hacer resaltar las características de mayor interés. Generalmente, en esta etapa se aplican técnicas para mejorar el contraste, eliminar el ruido,…

Vale decir que el pre-procesamiento es una etapa que depende mucho del tipo de problema que se maneja.

3.3 Segmentación

La etapa que le sigue al pre-procesamiento es la *segmentación*. De una forma general, la segmentación consiste en "partir una imagen de entrada en sus partes constituyentes o en los objetos que la componen". En general, "la segmentación autónoma es una de las labores más difíciles del tratamiento digital de imágenes" ([01] p9, [02] p27).

En nuestro caso particular, la segmentación consiste en extraer de las secuencias de imágenes, aquellos objetos que se movieron.

Las salidas que produce esta etapa generalmente son píxeles o información sobre los objetos deseados que corresponden con el contorno o con toda una región de la imagen. La elección entre la una y la otra se deja a libre decisión y depende de las técnicas que se usen en las etapas posteriores. Sin embargo podemos decir que la representación como un contorno es apropiada cuando la información de interés se refiere a características de la forma exterior, como lo son siluetas, figuras, formas, esquinas, inflexiones. La representación como una región es apropiada cuando se está interesado en propiedades internas como la textura, el color. También ocurre el caso en que ambos tipos de representación son necesarios, como suele ocurrir.

3.4 Descripción y codificación

Una vez conseguidos los datos que se desean de la imagen, se hace conveniente convertirlos de tal forma que sea más fácil su manipulación y tratamiento. La *descripción* se refiere al hecho de extraer los detalles o rasgos que contienen alguna información que sea de interés o que sea fundamental para diferenciar los objetos que resulten involucrados. Por otra parte, la *codificación* hace referencia a la manera como se van a manejar esos datos. Si por ejemplo, se desea contar el número de figuras circulares, una buena manera de describirlos sería con su radio. Para codificar estas figuras, se podría utilizar ese mismo radio y la posición que tiene en la imagen.

3.5 Reconocimiento

Luego, como etapa final del proceso, tendríamos el *reconocimiento*. Es en esta etapa donde se realiza la clasificación de los objetos. Aquí se eligen cuales son los que se buscan y cuales se pueden descartar. Podemos decir que en esta etapa se descubre, se identifica y se comprenden los datos entregados en la etapa de descripción. Esta etapa tiene como fin dotar a la máquina de la capacidad de decisión y análisis, en nuestro caso, la interpretación de una imagen, lo cual consiste en asignar un significado a un conjunto de elementos reconocidos en dicha imagen.

Todas estas etapas comprenden el proceso del reconocimiento de imágenes. Sin embargo, en algunos tipos de aplicaciones como en el caso nuestro, el sistema consta de otra etapa que es fundamental para poder llevar a cabo el reconocimiento, aunque esta etapa no hace parte explícita del proceso.

3.6 Aprendizaje

Para que se pueda hacer un buen proceso de clasificación es necesario contar con el conocimiento que nos permita tomar las decisiones correctas. Esta es la razón por la cual se incluye esta etapa. Para que el reconocimiento se lleve a cabo en una máquina, es necesario incluir un proceso que permita adquirir el conocimiento, así como también representarlo y mantenerlo. Generalmente las técnicas que se usan para este fin son: redes neuronales, funciones de

correspondencia, redes de probabilidad, clasificadores estadísticos, sistemas difusos…, en todo caso, la idea principal que se busca en todas estas diversas metodologías es tener organizadas de manera efectiva el conocimiento y realizar un buen empleo de este, que siempre es acerca del dominio específico del problema.

Otro detalle importante que se puede apreciar en la Figura 3-2 es que esta fase del proceso se hace en modo *off-line*, esto es, sin interferir o sin llevarse a cabo dentro del proceso de reconocimiento, que es el que se realiza de modo *online*. Esta etapa se ha decidido de esta manera para que el sistema cuente con el conocimiento mucho antes que se necesite. A esta fase también se le conoce con el nombre de fase de *entrenamiento*.

Tanto para el modo online como para el modo off-line, es necesario capturar la información, extraer la información relevante, codificarla y clasificarla o interpretarla para aprenderla, respectivamente de acuerdo a lo que se quiera (reconocer o entrenar).

CAPITULO 4. TÉCNICAS ESTUDIADAS

En los capítulos anteriores se ha dado una explicación que, si bien ha tenido mucho detalle, aún no es información específica que nos diga como realizar el proceso de identificación y reconocimiento de objetos en imágenes. Ese es el objetivo de este capítulo. Aquí se expondrán las técnicas de procesamiento de imágenes que implican la realización de este sistema.

Aunque cada día se siguen desarrollando nuevos métodos, expondremos aquellos que son los más comunes y aquellos que más se destacan por su eficiencia y/o rendimiento.

Vale aclarar que, todas las técnicas usadas para la implementación del sistema corresponden a métodos del *dominio espacial* y para imágenes en *escala de grises*. El dominio espacial se refiere al propio plano de la imagen, esto es el conjunto de píxeles, por lo que las técnicas de esta categoría se basan en la manipulación directa de estos.

4.1 Preliminares

El procesamiento de la imagen en el dominio espacial generalmente lo encontramos expresado de la forma:

$$g(x,y) = T[f(x,y)] \qquad \textbf{4.1}$$

Donde $f(x,y)$ es la imagen de entrada, $g(x,y)$ es la imagen procesada y T es un operador que actúa sobre la imagen de entrada o como en algunos casos, sobre un conjunto de imágenes de entrada, en algún entorno de (x,y).

La manera general (y es la utilizada aquí) como se define ese entorno es empleando un área de la imagen o subimagen de forma cuadrada o rectangular (esto por facilidad en la implementación), la cual esta centrada en (x,y), como se muestra en la Figura 4-1.

El centro de la subimagen se va desplazando por toda la imagen píxel a píxel, comenzado en la esquina superior izquierda y aplicando el operador en cada posición para obtener g.

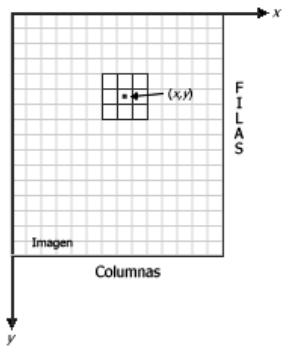


Figura 4-1. Entorno 3x3 alrededor de un punto (x,y) de una imagen

La forma más simple de T es cuando corresponde a un entorno de 1x1. En estos casos, el valor de g depende solo del valor de f en el punto (x,y). Debido precisamente a esto, las técnicas que realizan este tipo de procesamiento se les

conoce como *procesamiento de punto*. Este procesamiento generalmente se usa en etapas de mejoramiento de imágenes como la del pre-procesamiento.

Los entornos que son mayores nos permiten una amplia diversidad de funciones las cuales se suelen emplear en muchas otras etapas además del mejoramiento de imágenes. La manera como se utilizan este tipo de entornos es con el fin de determinar g en un punto (x, y) a partir de los valores de f en ese entorno. Esto suele implementarse empleando lo que se conoce con el nombre de *máscaras* (conocidas también como *ventanas* o *filtros*). En el caso de la Figura 4-1 la máscara es de tamaño 3x3. Las técnicas de mejora basadas en este tipo de procesamiento se les conocen como *procesamiento por máscaras* o *filtrado*.

4.2 Mejoramiento de la imagen

Para el mejoramiento de la imagen (objetivo que se desea lograr en la etapa de pre-procesamiento o en etapas anteriores a la presentación de la imagen), las técnicas que se conocen han permanecido sin muchas novedades.

A las técnicas de este tipo se le conocen como técnicas de *procesado de bajo nivel*: “El *procesado de bajo nivel* trata de un tipo de funciones a las que se puede considerar como reacciones automáticas, y que no requieren inteligencia por parte del sistema de análisis de imágenes.” (Véase [01] p 616). La finalidad de estas técnicas es poder compensar aspectos sobrantes o faltantes en la imagen de forma que resulte más adecuada que la original para poderla utilizar.

De las técnicas que hacen parte de esta categoría solo haremos referencia a aquellas que corresponden a las de filtrado, las otras solo las mencionaremos a manera informativa. Si se quiere tener más información de esto, véase [01] Cáp. 4.

4.2.1 Mejoras por procesamiento de punto

Como se dijo anteriormente, este tipo de procesamiento se basa solo en la intensidad de los píxeles individuales de la imagen y son las técnicas más simples para la mejora de imágenes. Dentro de esta categoría se encuentran:

Métodos de procesamiento simples: donde podemos encontrar:

- Negativo de imágenes: se emplea básicamente para la representación de imágenes médicas o donde se desee emplear negativos.
- Aumento del contraste: se emplea cuando se desea ampliar el rango de niveles de gris de una imagen. Generalmente se usa para aclarar imágenes que sean oscuras.
- Compresión del rango de niveles de gris: se emplea para visualizar imágenes que cuentan con un rango de niveles de gris demasiado amplio, reduciéndolo de tal manera que sea visible toda la información.
- Fraccionamiento de niveles de gris: se emplea para destacar un rango específico del nivel de gris de una imagen.
- Fraccionamiento de los planos de bits: sirve para destacar la contribución que realizan a la imagen determinados bits específicos.

Procesamiento de histogramas: el histograma de una imagen digital es una función discreta que muestra el número de píxeles de la imagen en cada nivel de color. Esto da una idea del valor de probabilidad de que aparezca cada nivel de gris de la imagen. La representación gráfica de esta función proporciona una descripción global de la apariencia de una imagen. De las técnicas que hacen parte de este tipo de procesamiento se encuentran:

- Ecualización del histograma: es una técnica para obtener un histograma con los niveles de gris con densidad uniforme, lo que en términos de mejora se traduce en un aumento del rango de niveles de gris de una imagen.
- Especificación del histograma: tiene la misma finalidad que la ecualización de un histograma, pero solo en determinados rangos de gris de la imagen.
- Mejora local: consiste en planear funciones de transformación basadas en la distribución de los niveles de grises en la vecindad de cada píxel de la imagen.

Sustracción de imágenes: simplemente consiste en la diferencia entre todos los pares de píxel de dos imágenes. El procesamiento de este tipo tiene sus aplicaciones más importantes en la segmentación y en la mejora. En esta última suele aplicarse para la eliminación de información en una imagen con el fin de destacar solo los detalles que son diferentes entre las dos imágenes, lo que serían los detalles mejorados.

Promediado de la imagen: se emplea para limpiar imágenes, generando una imagen de salida promediando los valores de los píxeles de un conjunto de imágenes de entrada. Se suele utilizar cuando se tiene un conjunto de una misma imagen las cuales han sido corrompidas con ruido.

4.2.2 Mejoramiento por Filtrado espacial

Se le suele dominar así al procesamiento de las imágenes que emplean máscaras espaciales para el procesamiento de imágenes (la distinción de espacial se debe a como se explico en los preliminares del capítulo y por diferenciarlo del *filtrado en el dominio de la frecuencia* el cual emplea la transformada de Fourier), y a las máscaras se les denomina *filtros espaciales*. De estos filtros vale decir que se les suele clasificar como filtros lineales y no lineales. Esta clasificación se deriva de los conceptos de la transformada de Fourier, que aquí no tratamos. Las demostraciones y toda la explicación sobre las máscaras se pueden encontrar en muchos libros de tratamiento de imágenes, de los cuales por su buena explicación recomendamos [01]. Vale decir que los filtros espaciales que utilizamos en la aplicación son lineales.

Antes de empezar hay que aclarar como se realiza el procesamiento con una máscara. La Figura 4-2 muestra una máscara 3x3 general.

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Figura 4-2. Una máscara 3x3 con coeficientes arbitrarios

Si nos referimos a los niveles de gris de los píxeles que quedan debajo de la máscara en un punto determinado por $z_1,z_2,\ldots,z_9$, la respuesta de una máscara es:

$$R = w_1 z_1 + w_2 z_2 + \ldots + w_9 z_9 \qquad \textbf{4.2}$$

Si tomamos como ejemplo de esto a la máscara que se aplica a una imagen como se muestra en la Figura 4-1, este valor R reemplazaría al valor que tenga la imagen en el punto (x, y). Este procedimiento se haría en cada uno de los píxeles en los cuales encaje la máscara. Para los píxeles en los que no (que serían los que se encuentran en los bordes), el valor de R se calcularía o bien con los valores parciales de los que cubre la máscara. Una de las consideraciones que normalmente se tienen con este tipo de procesamiento es ir colocando los valores calculados dentro de otra imagen para que estos nuevos valores no influyan en los cálculos de los otros píxeles.

Dentro de los filtros de este tipo tenemos:

Filtros suavizantes: este tipo de filtros causa en la imagen un efecto de difuminado, haciendo que la imagen aparezca borrosa. Lo se quiere lograr

aplicando este tipo de filtros es eliminar el ruido, eliminar detalles pequeños de la imagen o rellenar espacios entre líneas. Entre los más comunes encontramos:

- Filtro de paso bajo: Es el filtro suavizante más básico. La forma en que se implementa este tipo de filtro es estableciendo todos los pesos con valores positivos, de esta forma y para el caso de filtros 3x3, la manera más simple consiste en una máscara en la que todos los coeficientes tienen el valor de 1. Nótese que si aplicamos la ecuación 4.2 la respuesta sería la suma de los niveles de gris de los nueve píxeles, lo que podría causar que el valor de R sobrepase el rango válido de valores de grises. La manera de solucionar este problema es cambiando la escala de la suma haciendo una división de R por 9, quedando el filtro como se muestra en la Figura 4-3. Filtro de paso bajo básico

$\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

Figura 4-3. Filtro de paso bajo básico

De esta forma, la respuesta R será el valor medio de todos los píxeles tenidos evaluados por la máscara. Debido a esto, a este tipo de filtros se les denomina *filtros de promediado* o *máscaras de promediado en el entorno*. La figura muestra un ejemplo de difuminado al empelar filtros suavizantes.

- Filtro de mediana: es una alternativa mejor que el filtro de promediado. Se usa cuando no se quiere causar un difuminado en la imagen, ni la eliminación de bordes ni detalles de realce que el promediado conlleva, pero se desea eliminar el ruido. La manera en que funciona es reemplazando el píxel en cuestión por la *mediana* de los niveles de gris en el entorno. Hace parte de los filtros no lineales, debido a que no se implementa como una máscara lineal. La mediana de un conjunto de valores es aquel valor que divide al conjunto en dos partes iguales. Para esto el conjunto debe estar previamente ordenado. Ej.: si se tiene (2, 2, 5, 8, 9), la mediana es el 5.

Filtros realzantes: se usan cuando lo que se desea es destacar los detalles finos de una imagen o los detalles que han sido difuminados. Los principales filtros realzantes son:

- Filtro de paso alto: es el filtro de realce más básico. La manera como se implementa este filtro es usando valores positivos en el centro y negativos en la periferia. En el caso de una máscara de 3x3 esto se hace estableciendo un valor positivo en el centro y valores negativos en el resto de valores, quedando la máscara como se muestra en la Figura 4-4.

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 4-4. Filtro de paso alto básico

Nótese que la suma de los valores es cero. Así cuando se evalúa un píxel que se encuentra en un entorno de valores constantes (o poco variables) la respuesta será un valor de cercano a 0, así como lo hace el operador diferencial. Una mejora a este filtro corresponde al *filtro de highboost* (Ver [01] p 213). A esta máscara se le suele hacer el cambio de escala debido a que suelen presentarse resultados negativos o resultados que sobrepasan el rango.

- Filtros diferenciales: la operación que es análoga al promediado es la integración, la cual tiende a eliminar los detalles de realce. Teniendo esto en cuenta, es de esperar que la diferenciación cause el efecto contrario, aumentar la nitidez de la imagen. La forma más común de implementar este tipo de filtros es a través del gradiente. Dentro de estas implementaciones encontramos los *operadores de gradiente de*:

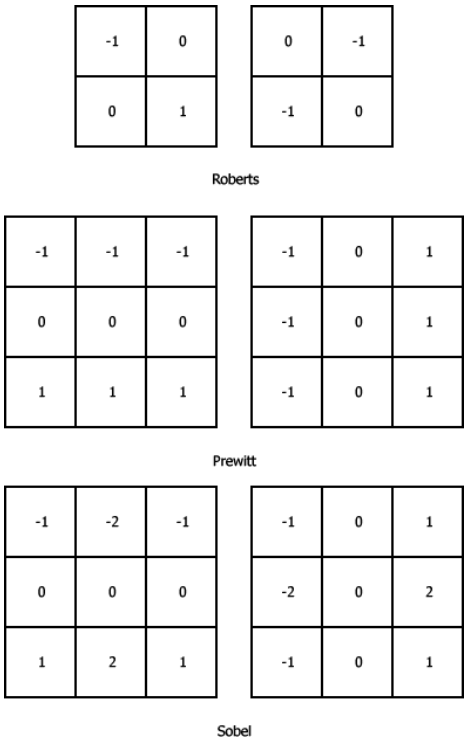


Figura 4-5. Operadores de gradiente

De estos, es el de Sobel con el que se han conseguido mejores resultados. La manera en que se suelen usar estos filtros es aplicando cada máscara a la imagen original por separado y luego se realiza una suma píxel a píxel de los dos resultados (esto es así debido a que una máscara es para las componentes horizontales y la otra para las verticales). Un ejemplo del resultado de la aplicación de un filtro de este tipo se puede apreciar en la Figura 4-6.

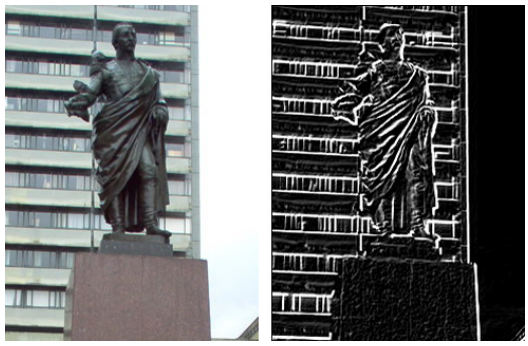


Figura 4-6. Aplicación del filtro de Sobel a una imagen

Existen otros filtros diferenciales, entre los que podemos mencionar el *Laplaciano*, pero que no explicaremos aquí debido que no es muy común en la practica.

Para nuestro sistema, se realizó la implementación de la mayoría de los filtros aquí mencionados, todo esto con el fin de que se pudiera probar cuales tienen mejores resultados en nuestros sistema y para detectar posibles formas de uso en etapas o aplicaciones posteriores.

4.3 Segmentación de Imágenes

Las técnicas de segmentación de imágenes son aquellas cuyo fin es la extracción de información de una imagen. Como su nombre lo indica, su objetivo es segmentar la imagen, esto es, dividir una imagen en sus partes constituyentes. El nivel al que se hace esta división depende del tipo de aplicación, pero que se lleva a cabo hasta que se hayan aislado los objetos deseados.

Para nuestra aplicación hemos elegido al movimiento como técnica de segmentación, lo cual se explica en el apartado siguiente. Vale decir que de esta etapa hacen parte muchas técnicas las cuales se usan para una gran variedad de propósitos, dentro de las que podemos destacar las técnicas de detección de bordes, las de umbralización, las de detección de regiones, pero que no se explicarán por su poco valor práctico que tienen para nuestra aplicación.

4.4 Detección de objetos en movimiento

El movimiento suele ser una técnica muy utilizada para la extracción de objetos de un fondo estático, tanto de manera práctica para aplicaciones como en la vida real por animales y seres humanos.

De las técnicas que se basan en el movimiento, se destacan tres metodologías muy aceptadas: *diferencias temporales*, *estimación de fondo* y *flujo óptico*.

4.4.1 Detección por diferencias temporales

Es el método más sencillo utilizado para detectar cambios entre dos imágenes tomados en tiempos diferentes. Básicamente lo que se hace es una comparación píxel a píxel de las dos imágenes. Esto en la práctica se logra generando una imagen diferencia. El procedimiento es el siguiente: se toma una imagen como la imagen de referencia, la cual solo va a contener los elementos estáticos. Luego se compara esta imagen con una imagen posterior en el mismo entorno pero que incluirá objetos en movimiento, lo cual causará que se eliminen los elementos que

no han sufrido ningún cambio en su posición dejando solamente aquellos que elementos no estacionarios.

Una imagen diferencia de dos imágenes $f(x,y,t_i)$ y $f(x,y,t_j)$, tomadas en los tiempos t_i y t_j respectivamente, se puede definir como:

$$d_{ij}(x,y) = \begin{cases} 1 & si \left| f(x,y,t_i) - f(x,y,t_j) \right| > \theta \\ 0 & en\ caso\ contrario \end{cases} \qquad \textbf{4.3}$$

Donde θ es un valor umbral. Nótese como la imagen estará formada solo por unos y ceros, estableciendo en 1 aquellos píxeles (x,y) donde la diferencia del nivel de gris de dos imágenes es apreciablemente diferente de esas coordenadas, según el valor determinado por el umbral θ. De esta forma, todos los píxeles con valor de 1 se consideran como el resultado del movimiento de algún objeto.

Este tipo de técnica es aplicable solamente cuando entre las dos imágenes la iluminación permanece constante. En la práctica, el valor de 1 suele aparecer por causa del ruido. Sin embargo, estos valores son solo puntos aislados que pueden ser tratados con las técnicas de filtrado anteriormente mencionadas.

Aunque esta metodología tiene la ventaja de ser muy adaptable a entornos dinámicos, el trabajo que realiza para extraer los píxeles relevantes es deficiente.

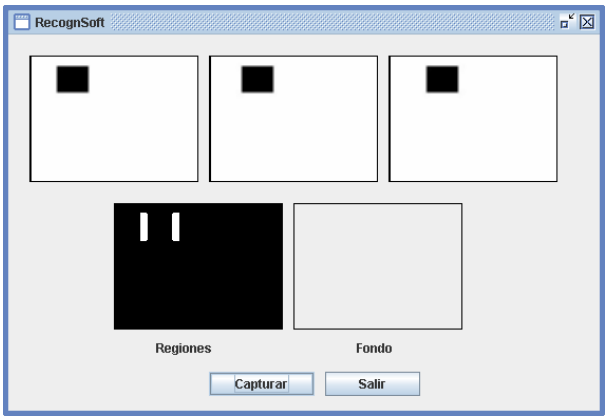


Figura 4-7. Resultados de las diferencias temporales

En la Figura 4-7 se puede observar esto. Las imágenes que se encuentran en la parte superior, corresponden a una secuencia de imágenes en movimiento. La primera de ellas, la imagen de la izquierda, es la imagen de referencia. La imagen que aparece con el rótulo de Región muestra la imagen diferencia calculada utilizando la ecuación 4.3. Observe como el proceso de diferenciación genera dos regiones separadas, o dicho de otra forma, observe como la diferenciación no detecta al objeto en su totalidad.

4.4.2 Detección por estimación de fondo

Las técnicas de estimación del fondo son las que mejor respuesta a la detección de objetos en movimiento tienen. Esta aproximación consiste en estimar el fondo estático de la escena a través de las imágenes de entrada para que luego se pueda comparar con las imágenes a evaluar. Se puede encontrar una explicación

de cada una de las técnicas aquí mencionadas en **¡Error! No se encuentra el origen de la referencia.** Pág. 2.

El método clásico de esta técnica se basa en promediar el nivel de gris de un número determinado de imágenes, a lo que se le suele denominar enventanado temporal, y cuyo cálculo se basa en la expresión:

$$B(x,y,t)=\frac{1}{N}\sum_{t'=t-N}^{t} I(x,y,t') \qquad \textbf{4.4}$$

Donde $B(x,y,t)$ es el fondo estimado en la posición (x,y) en el instante t, $I(x,y,t')$ representa el valor de la imagen en la posición (x,y) en el instante t' y N el número de imágenes de la ventana temporal. El promediado lo que hace es que los objetos que se mueven tengan menor peso en la contribución global al fondo que el resto. Sin embargo, uno de los problemas que suelen presentarse con este enfoque es que si un objeto es muy lento de tal forma que permanece en la misma región de la imagen por un periodo tiempo mayor o igual a lo que dura la ventana temporal, entonces este es considerado parte del fondo, lo que causará más tarde errores en los cálculos posteriores. La manera de solucionar este problema es haciendo que N se ajuste dinámicamente a la velocidad de los objetos no estáticos presentes en la escena, lo cual trae consigo otro problema, y es que la obtención de un N óptimo resulta complicado en la práctica. Esto también tiene una solución y es utilizar un valor de N elevado. El costo de hacer esto es el aumento del volumen de datos a almacenar y procesar lo que se traduce en un aumento de los requerimientos del sistema.

Un enfoque distinto pero con el mismo fundamento, consiste en enmascarar dinámicamente la región de la imagen donde se ha detectado movimiento para evitar que entre en los cálculos de promediado del fondo, lo cual causa buenos resultados. De los problemas que presenta este enfoque es que si un objeto no estático permanece dentro de la escena en toda la ventana temporal, podría causar que el fondo se degenere, debido a que permanecerían esas áreas ocultas por mucho tiempo y no se tendrían en cuenta para la estimación.

Otra de las aproximaciones que tiene está técnica se le denomina olvido exponencial, el cual ya no requiere el almacenamiento de las N imágenes porque la estimación que se realiza del fondo se basa en la estimación anterior. El cálculo de este se hace de la siguiente forma:

$$B(x,y,t) = (1-\alpha)\cdot B(x,y,t-1) + \alpha \cdot I(x,y,t) \qquad \textbf{4.5}$$

Donde $B(x,y,t)$ es el fondo estimado en el instante t, $B(x,y,t-1)$ el fondo estimado previamente, $I(x,y,t)$ la última imagen de entrada y α es el parámetro de olvido, el cual especifica que tan rápido el fondo se olvida para ser reemplazado por la nueva imagen de entrada. Aún en esta aproximación, los objetos lentos distorsionan la estimación.

4.4.3 Detección por medio del flujo óptico

El flujo óptico es una de las técnicas más robustas para la detección de objetos en movimiento. La finalidad de esta técnica es determinar la velocidad de los objetos basándose en el movimiento descrito dentro de una secuencia de imágenes.

Una definición del flujo óptico se puede decir que es el movimiento relativo de los píxeles con respecto a una cámara. En el caso ideal, en el que la iluminación se mantiene, el flujo óptico corresponde con el campo de velocidades. Esto debido a que con la iluminación constante, un punto de la imagen conserva su intensidad al moverse. Extendiendo este hecho a todos los puntos de la imagen, se puede estudiar la evolución temporal de los objetos y tener una noción diferencial de la posición que tendrá el objeto en el siguiente instante, calculando con esto la componente de la velocidad en dirección normal a los contornos. Para hallar la segunda componente de la velocidad deben hacerse suposiciones adicionales.

Para el cálculo de estas componentes este método se basa en varios modelos matemáticos, que aquí no explicaremos. También vale decir que la realización de estos cálculos supone un gran gasto de recursos computacionales lo cual no es conveniente para nuestra aplicación. Por lo cual se descartó esta idea.

4.4.4 El algoritmo de detección de objetos en movimiento

Ya se sabe que la detección de movimiento es una etapa importante de nuestra aplicación, lo que nos ha hecho prestarle una atención considerable. Ya hemos dicho que el flujo óptico se ha descartado por su complejidad y por ser computacionalmente costoso. También ya hemos mencionado los problemas que conllevan cada una de las posibles técnicas a usar.

Por todo esto y por los resultados vistos en otros estudios y en otras aplicaciones, el método elegido para la detección de objetos en movimiento terminó siendo una

combinación de dos de los métodos antes mencionados: se utilizó una modificación del método de diferencias temporales junto con el método de estimación de fondo por medio de olvido exponencial también con una modificación que le hicimos.

El procedimiento que sigue el algoritmo es: primero se realiza la operación de diferencia de tres imágenes para determinar las regiones de movimiento con mayor seguridad y a esto lo sigue la operación de estimación de fondo para extraer la región completa de movimiento.

La técnica es la siguiente:

Consideremos una secuencia de imágenes generadas por una cámara estática. Representemos además el valor de la intensidad de un píxel en la posición (x, y) en un instante t por $I(x, y, t)$. La regla de la diferencia de las tres imágenes sugiere que un píxel esta legítimamente en movimiento si su intensidad ha cambiado significantemente entre ambas la imagen actual y la última imagen y la imagen actual y la imagen inmediatamente siguiente. Esto es, un píxel (x, y) se esta moviendo si

$$\left(\left|I(x,y,t) - I(x,y,t-1)\right| > \theta\right) y \left(\left|I(x,y,t) - I(x,y,t-2)\right| > \theta\right) \qquad \textbf{4.6}$$

Donde θ es un valor de umbral (una aproximación más robusta corresponde con hacer este umbral variable con el tiempo con respecto a referencias estadísticas basadas en los cambios significantes de cada píxel). Ya es sabido que el problema de las diferencias temporales es que los píxeles del interior de un objeto

con intensidad uniforme no son incluidos en el conjunto de píxeles en movimiento. Sin embargo, teniendo localizados los píxeles que han indicado movimiento, los píxeles del interior pueden ser "llenados" aplicando la técnica de estimación de fondo para extraer todos los píxeles en movimiento.

El fondo se puede considerar como un modelo estadístico de las intensidades de los píxeles observados en las secuencias de imágenes. Al inicio, el fondo es inicializado con la primera imagen de entrada y luego en el tiempo es actualizado siguiendo la función:

$$B(x,y,t+1)=\begin{Bmatrix}\alpha B(x,y,t)+(1-\alpha)I(x,y,t), & si\ es\ un\ píxel\ estático\\ B(x,y,t), & si\ es\ un\ píxel\ en\ movimiento\end{Bmatrix} \quad \textbf{4.7}$$

Donde α es el parámetro de olvido.

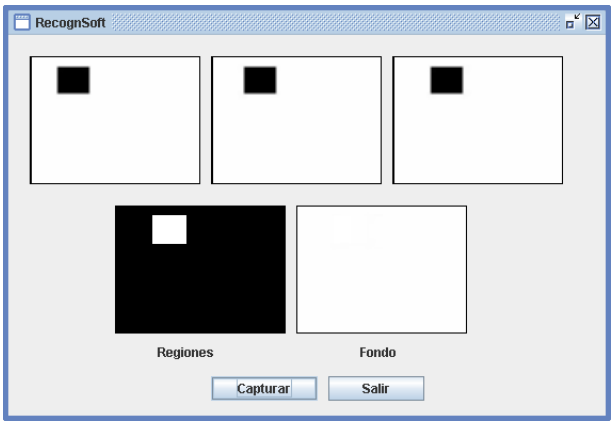


Figura 4-8. Prueba del algoritmo de detección

En la Figura 4-8 se puede ver el resultado de aplicar este método a una secuencia de imágenes en movimiento. Nótese que el fondo es una imagen en blanco, lo que

quiere decir que se ha estimado el fondo sin tener en cuenta la parte de la imagen que se mueve. La región detectada es el resultado de la comparación de la última imagen de entrada y el fondo.

4.5 Descripción de los objetos

Como se dijo antes, la descripción de un objeto se refiere al hecho de extraer los detalles o rasgos que contienen alguna información de interés o información fundamental para diferenciar los objetos que resulten involucrados en la escena. Para esto, existen varios métodos que pueden facilitar la manipulación de las características de dichos objetos entre los mas importantes encontramos los códigos de cadena y los momentos invariantes.

4.5.1 Códigos de cadena

Esta es una forma de representación basada en el trabajo de Freeman. Consiste en seguir el contorno del objeto de la imagen según las direcciones de los píxeles vecinos, así, dependiendo de la exactitud con que se quiera trabajar se pueden generar cuatro (*4-connected neighbors*) u ocho (*8-connected neighbors*) posibles direcciones de los píxeles vecinos [02] por ejemplo, para las 8 conexiones se manejan 8 direcciones Este, Noreste, Norte, Noroeste, Oeste, Suroeste y Sureste. Cada una de las direcciones puede ser representada por un string de 3 bits.

Esta representación presenta la desventaja de que los resultados de las codificaciones son variantes al escalado y al rotado del objeto, por consiguiente

esta muy limitado para situaciones donde el objeto cambie de forma en el tiempo o presente rotación alguna.

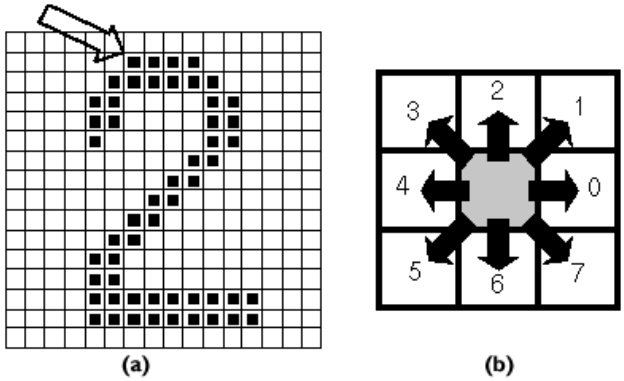


Figura 4-9. (a) Códigos de cadena del número 2 (b) Mapa de píxeles vecinos (8-connected)

4.5.2 Momentos Invariantes

Los momentos invariantes son funciones de valores que permanecen estáticos ante ciertas transformaciones del contorno, a pesar de esto, los momentos están definidos en una función de intensidad continua de una imagen, con esto, es posible una simple aproximación para una imagen binaria discreta usando la operación de sumatoria.

El principio de este método se basa en hallar los momentos invariantes mediante el cálculo de los momentos centrales de una imagen binaria representada por una matriz de tamaño MxN (véase [12] Pág. 149).

Partimos de tomando a $f(x,y)$ como la intensidad del punto (x,y) en una región. Entonces el momento de orden $(p+q)$ para la región se define como,

$$m_{pq} = \sum_x \sum_y x^p y^q f(x,y) \qquad \textbf{4.8}$$

donde p y q representan el orden de la cantidad de momentos según el teorema de representación de los momentos que nos dice que el conjunto infinito de momentos p,q determinan unívocamente $f(x,y)$ y viceversa. Si nosotros conocemos los momentos de $f(x,y)$ hasta un determinado orden N, es posible encontrar la función continua $g(x,y)$ cuyos momentos de orden hasta $(p+q)=N$ adapten la función $f(x,y)$.

La sumatoria se toma sobre todas las coordenadas espaciales (x,y) de puntos de la región. El momento central de orden $(p+q)$ viene dado por,

$$\mu_{pq} = \sum_x \sum_y (x-\bar{x})^p (y-\bar{y})^q f(x,y) \quad \text{donde, } \bar{x} = \frac{m_{10}}{m_{00}}; \bar{y} = \frac{m_{01}}{m_{00}}; \qquad \textbf{4.9}$$

Los momentos centrales normalizados de orden $(p+q)$ se definen como:

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}} \text{ donde, } \gamma = \frac{p+q}{2} + 1 \text{ para } p+q = 2,3,\ldots \ldots \qquad \textbf{4.10}$$

El siguiente conjunto de momentos invariantes se puede obtener como sigue,

$$\phi_1 = \eta_{20} + \eta_{02} \qquad \textbf{4.11}$$

$$\phi_2 = (\eta_{20} + \eta_{02})^2 + 4\eta_{11}^2 \qquad \textbf{4.12}$$

$$\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \qquad \textbf{4.13}$$

4.14

$$\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$

$$\phi_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \quad \textbf{4.15}$$

$$\phi_6 = (\eta_{20} - \eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + 3\eta_{21}(3\eta_{30} - \eta_{12})(\eta_{21} + \eta_{03}) \quad \textbf{4.16}$$

$$\phi_7 = (3\eta_{21} - \eta_{02})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{12} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \quad \textbf{4.17}$$

Para que todos los momentos contribuyan por igual en la función de discriminación, y por tanto estén dentro del mismo orden de magnitud, a veces es necesario proceder a una normalización del tipo,

$$\phi_n^{'} = abs[\ln(abs(\phi_n))] \quad \textbf{4.18}$$

Si se quiere tener más información de esto, véase [03].

4.5.3 El algoritmo de descripción de objetos

Para la implementación del algoritmo de descripción de objetos trabajamos con los momentos invariantes de Hu de orden $(p+q)=3$, por lo tanto corresponden a los 7 momentos invariantes de Hu. Mas adelante se hará una explicación mas detallada acerca de cómo se elaboro el proceso de descripción de las imágenes. A continuación mostramos el resultado de una imagen cualquiera sometida a este tipo de descriptor. Esta imagen también fue escalada y rotada:

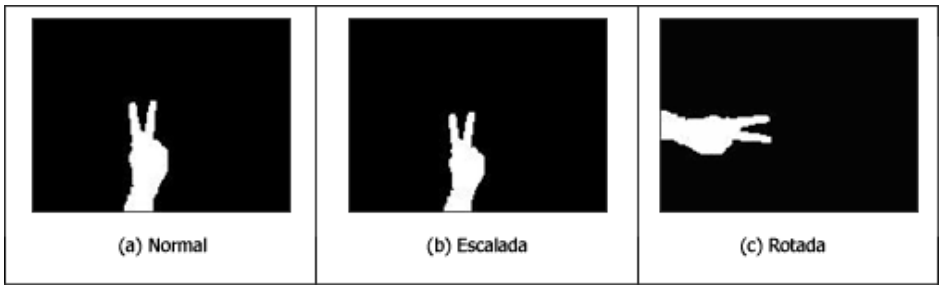


Figura 4-10. Momentos invariantes de una imagen

```
H1a [1.182 2.842  3.366  3.366  6.731  1.930  6.770]
H2b [1.179 2.842  3.658  3.158  6.715  1.934  6.360]
H3c [1.182 2.849  3.366  3.366  6.730  1.941  6.983]
```

Obsérvese la similitud de valores entre las dos imágenes la escalada y la rotada frente a los valores de la normal.

4.6 Reconocimiento de los objetos

Esta etapa del proceso consiste en darle a la maquina la capacidad de interpretar y reconocer un objeto en una escena, para esto definimos el concepto de patrones, que se refiere a las características que identifican a un objeto, por ende, cuando reconocemos un objeto, lo reconocemos por los patrones que los caracterizan y que lo determinan como un objeto único y diferente.

Un sistema de reconocimiento de patrones esta compuesto por 2 Etapas. La primera etapa es la extracción de características y la segunda etapa consiste en la clasificación. El fin de la extracción es el de buscar características que permitan la rápida clasificación de los objetos (descripción del objeto). Para la clasificación existen varios métodos entre los cuales se encuentran:

4.6.1 Clasificador Bayesiano:

"La teoría de decisión de Bayes, es un sistema que minimiza el error de clasificación", véase [11]. Esta teoría juega un rol a priori. Esto es, cuando hay información prioritaria acerca de algo que queremos clasificar. Además, este método se basa, en el supuesto de que el problema de la decisión se enfoca en términos probabilísticas y que todas las probabilidades relevantes resultan conocidas.

Por ejemplo, supongamos que no sabemos mucho acerca de lo que contiene una canasta de frutas. La única información que sabemos es que el 80% de la canasta son manzanas, y el resto son naranjas. Si esta es la única información que tenemos, entonces podemos clasificar que una fruta cogida al azar de la canasta es una manzana. En esta, la información a priori es la probabilidad de que ni la manzana ni la naranja estén en la canasta. Si nosotros solo tenemos esta información, entonces podríamos tener la siguiente regla:

Decidir 'manzana' si P (manzana) > P (naranja), de otra manera decide 'naranja'

Entonces, P (manzana) es la probabilidad de sacar una manzana de la canasta. Esto significa que P (manzana) = 0.8 (80%). Esta es una probabilidad extraña porque si seguimos la regla dada, entonces, estaremos clasificando una fruta al azar como manzana. Pero si nosotros usamos esta regla, entonces estaremos bien el 80% del tiempo.

4.6.2 Clasificación o discriminación linear:

La meta de este método consiste en asignar mediante observaciones una regla de discriminación que puede luego de varias observaciones responder de manera lineal a requerimientos de clasificación. La clasificación lineal esta compuesta por una formula matemática para predecir un resultado binario. Este resultado es un falso o verdadero (positivo o negativo). En general asumimos que el resultado es una variable booleana.

Para hacer esto, usamos una formula linear con la información de entrada. La forma lineal es calculada por las entradas y el resultado es comparado contra una constante base. Esto es dependiendo de el resultado que nosotros estemos dispuestos a predecir verdadero o falso. La siguiente ecuación puede considerarse como un discriminador:

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n > x_0 \qquad \textbf{4.19}$$

Donde a_1, a_2,… son variables que corresponden a una observación. Y x_1, x_2,… junto con x_0 son el vector de solución mas la constante base. Para más información véase [11] Cáp. 2.

4.6.3 Redes Neuronales:

Las redes neuronales hacen parte de la investigación para la creación de una IA (Inteligencia Artificial) que es un campo de la ciencia que trata de darle a los computadores las habilidades de los humanos. Una de las respuestas a esta interrogante son las redes neuronales. El cerebro humano es un ejemplo de las redes neuronales.

Las redes neuronales posee varias características deseables para cualquier sistema de reconocimiento, tales como:

- Es robusto y tolerante a fallas, diariamente mueren neuronas sin afectar su desempeño.
- Es flexible, se ajusta a nuevos ambientes por aprendizaje, no hay que programarlo.
- Puede manejar información difusa, con ruido o inconsistente.
- Es altamente paralelo.

Las RNA son una teoría que aún esta en proceso de desarrollo, su verdadera potencialidad no se ha alcanzado todavía; aunque los investigadores han desarrollado potentes algoritmos de aprendizaje de gran valor práctico, las representaciones y procedimientos de que se sirve el cerebro, son aún desconocidas. Tarde o temprano los estudios computacionales del aprendizaje con RNA acabarán por converger a los métodos descubiertos por evolución, cuando eso suceda, muchos datos empíricos concernientes al cerebro

comenzarán súbitamente a adquirir sentido y se tornarán factibles muchas aplicaciones desconocidas de las redes neuronales. Si se quiere tener más información de esto, véase [20] .

Comentario [FMB2]: Esta no existe ¿?

La teoría de las RNA ha brindado una alternativa a la computación clásica, para aquellos problemas, en los cuales los métodos tradicionales no han entregado resultados muy convincentes, o poco convenientes. Las aplicaciones más exitosas de las RNA son:

1. Procesamiento de imágenes y de voz
2. Reconocimiento de patrones
3. Planeamiento
4. Interfaces adaptivas para sistemas Hombre/máquina
5. Predicción
6. Control y optimización
7. Filtrado de señales

El reconocimiento de patrones es uno de los usos más comunes de las redes neuronales. Reconocimiento de patrones es una simple habilidad de reconocer un patrón. El patrón debe ser reconocido aunque el patrón este distorsionado de cualquier forma, por ejemplo, una persona que maneje debe ser capaz de identificar las luces del semáforo. Este es un proceso crítico de reconocimiento de patrones realizado por los conductores todos los días pero, no todas las luces de tráfico siempre son iguales, la visibilidad de estas luces puede ser distorsionada

dependiendo del estado del tiempo o de la estación. Esta no es una tarea complicada para el ser humano.

Analogía con las redes neuronales biológicas: Las neuronas se modelan mediante unidades de proceso. Cada unidad de proceso se compone de una red de conexiones de entrada, una función de red (de propagación), encargada de computar la entrada total combinada de todas las conexiones, un núcleo central de proceso, encargado de aplicar la función de activación, y la salida, por dónde se transmite el valor de activación a otras unidades.

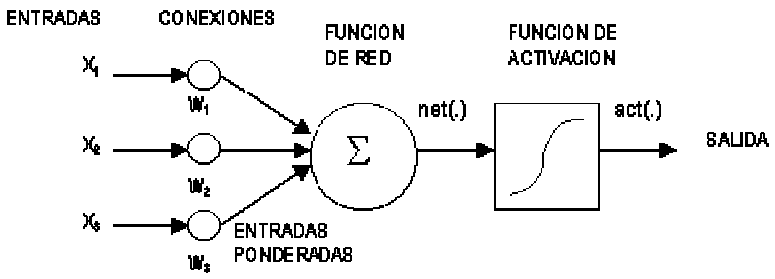


Figura 4-11. Unidad de Proceso típica

La función de red es típicamente la sumatoria ponderada, mientras que la función de activación suele ser alguna función de umbral o una función sigmoidal.

- **Función de propagación o de red:** Calcula el valor de base o entrada total a la unidad, generalmente como simple suma ponderada de todas las entradas recibidas, es decir, de las entradas multiplicadas por el peso o valor de las conexiones. Equivale a la combinación de las señales excitatorias e inhibitorias de las neuronas biológicas.

- **Función de activación:** Es quizás la característica principal o definitoria de las neuronas, la que mejor define el comportamiento de la misma. Se usan diferentes tipos de funciones, desde simples funciones simples de umbral a funciones no lineales. Se encarga de calcular el nivel o estado de activación de la neurona en función de la entrada total.
- **Conexiones ponderadas:** hacen el papel de las conexiones sinápticas, el peso de la conexión equivale a la fuerza o efectividad de la sinápsis. La existencia de conexiones determina si es posible que una unidad influya sobre otra, el valor de los pesos y el signo de los mismos definen el tipo (excitatorio/inhibitorio) y la intensidad de la influencia.
- **Salida:** calcula la salida de la neurona en función de la activación de la misma, aunque normalmente no se aplica más que la función identidad, y se toma como salida el valor de activación. El valor de salida cumpliría la función de la tasa de disparo en las neuronas biológicas. De momento consideramos el caso más simple, aunque en el apartado de sistemas neurofuzzy veremos un caso en que se utiliza una función de salida diferente a la identidad.

Estructura y formas de interconexión

Dentro de una red neuronal, los elementos de procesamiento se encuentran agrupados por capas, una capa es una colección de neuronas; de acuerdo a la ubicación de la capa en la RNA, esta recibe diferentes nombres:

Capa de entrada: Recibe las señales de la entrada de la red, algunos autores no consideran el vector de entrada como una capa pues allí no se lleva a cabo ningún proceso.

Capas ocultas: Estas capas son aquellas que no tienen contacto con el medio exterior, sus elementos pueden tener diferentes conexiones y son estas las que determinan las diferentes topologías de la red

Capa de salida: Recibe la información de la capa oculta y transmite la respuesta al medio externo.

La función de propagación (de red o de base): Como ya hemos comentado, se encarga de calcular la entrada total de la neurona como combinación de todas las entradas. Dentro de las funciones de propagación más comunes, encontramos:

- **Función lineal de base (LBF):** Consiste en la sumatoria de las entradas ponderadas. Se trata de una función de tipo plano, esto es, de primer orden. Esta función suele ser la más usada.
- **Función radial de base (RBF):** Función de tipo esférico, de segundo orden, no lineal. El valor de red representa la distancia a un determinado patrón de referencia.

La función de activación: Se suele distinguir entre funciones lineales, en las que la salida es proporcional a la entrada; funciones de umbral, en las cuales la salida es un valor discreto (típicamente binario 0/1) que depende de si la estimulación

total supera o no un determinado valor de umbral; y funciones no lineales, no proporcionales a la entrada.

Función de umbral: En un principio se pensó que las neuronas usaban una función de umbral, es decir, que permanecían inactivas y se activaban sólo si la estimulación total superaba cierto valor límite; esto se puede modelar con una función escalón: la más típica es el escalón unitario: la función devuelve 0 por debajo del valor crítico (umbral) y 1 por encima.

Después se comprobó que las neuronas emitían impulsos de actividad eléctrica con una frecuencia variable, dependiendo de la intensidad de la estimulación recibida, y que tenían cierta actividad hasta en reposo, con estimulación nula. Estos descubrimientos llevaron al uso de funciones no lineales con esas características, como la función sigmoidal, con un perfil parecido al escalón de una función de umbral, pero continúa.

Función sigmoidal o logística: Es probablemente la función de activación más empleada en la actualidad. Se trata de una función continua no lineal con bastante plausibilidad fisiológica. La función sigmoidal posee un rango comprendido entre 0 y 1. Esto, aplicado a las unidades de proceso de una red neuronal artificial significa que, sea cual sea la entrada, la salida estará comprendida entre 0 y1.

Métodos de entrenamiento

El aprendizaje consiste en la presentación de patrones a la red, y la subsiguiente modificación de los pesos de las conexiones siguiendo alguna regla de aprendizaje que trata de optimizar su respuesta, generalmente mediante la minimización del error o la optimización de alguna "función de energía".

Normalmente, el entrenamiento se suele clasificar en:

Entrenamiento supervisado: Es el método de aprendizaje más sencillo consiste en la presentación de patrones de entrada junto a los patrones de salida deseados (targets) para cada patrón de entrada, por eso se llama aprendizaje supervisado. Esto permite al algoritmo de entrenamiento supervisado a ajustar la matriz de pesos de las conexiones basados en la diferencia de la salida esperada y la real.

Entrenamiento no supervisado: Es un método en que la red neuronal contiene unos conjuntos de valores de entrada para el entrenamiento pero no uno de salidas esperadas. El entrenamiento no supervisado es usualmente usado para redes neuronales de clasificación. Un clasificador toma los patrones de entrada, los cuales tiene en las neuronas de entrada. Estos patrones de entrada son procesados, y dados mediante solo una respuesta a una neurona esta en la capa de salida. Este tipo de entrenamiento es muy común para redes de reconocimiento de escritura a mano y minería de datos.

Existen varios métodos sencillos para encontrar códigos económicos que al mismo tiempo permitan una buena reconstrucción de la entrada: el aprendizaje por

componentes principales, el aprendizaje competitivo y los códigos demográficos (Hinton, 1992).

Entrenamiento reforzado: La idea es similar a la del aprendizaje supervisado, sólo que aquí la información dada por el maestro es mínima, se limita a indicar si la respuesta de la red es correcta o incorrecta. En este sentido se asimila a la noción tomada de la psicología de condicionamiento por refuerzo, que en resumen defiende que se aprenden (en el sentido de que tienen más probabilidad de repetirse) las conductas reforzadas positivamente y viceversa (las conductas castigadas o reforzadas negativamente reducen la posibilidad de aparecer).

Una noción estrechamente ligada a la de aprendizaje reforzado es la de Redes Basadas en la Decisión. Son redes para clasificación de patrones, similares a los asociadores de patrones, en las cuales el maestro, en vez de indicar la salida deseada exacta (target), indica sólo la clase correcta para cada patrón de entrada.

En las redes basadas en la decisión se utiliza una función discriminante para determinar a que categoría o clase pertenece el patrón de entrada, si durante el entrenamiento, una respuesta no coincide con la dada con el maestro, se cambian los pesos, si no se dejan igual (de aquí viene el nombre de DBNN). El aprendizaje consiste en encontrar los pesos que dan la clasificación correcta.

Tipo de redes neuronales:

Asociadores de patrones o memorias heteroasociativas: Son redes de dos o más capas cuyo objetivo es asociar, generalmente a través de un proceso de

aprendizaje supervisado, pares de estímulos o ítems distintos, llamados patrón de entrada y patrón de salida. Se trata de conseguir que la presentación de un patrón de entrada provoque la recuperación del patrón de salida con el que fue asociado durante el aprendizaje.

Podemos comparar un heteroasociador con los modelos de regresión estadística, los cuales tratan de hallar la relación entre una serie de variables, llamadas predictores y criterio, a partir de una serie de datos conocidos. De esta manera podemos predecir la variable criterio a partir de las variables usadas como predictores.

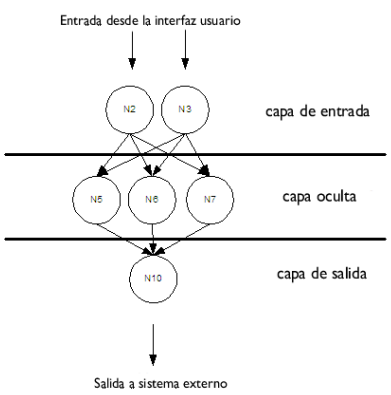
La diferencia fundamental entre un asociador de patrones y un modelo de regresión está en que el primero es capaz de representar relaciones mucho más complejas que un modelo de regresión, y en la forma en que aprende (inductivamente) y representa dichas relaciones (distribuidas por toda la red), además de que va a poseer las cualidades generales de las redes neuronales (generalización, tolerancia al ruido, etc.)

Como ejemplo de un asociador podemos citar la red Feed Forward.

Feed Forward Neural Network: El primer termino “Feed Forward” describe como esta red neuronal procesa los patrones y hace los llamados. Cuando se usa esta las neuronas están conectadas solo hacia delante. Cada capa de la red neuronal contiene conexiones a las otras capaz (desde la entrada hasta la salida), pero no hay conexiones hacia atrás.

Comentario [FMB3]: Esto es parte de la clasificación de que o que?

Esta red utiliza el termino Backpropagation, que describe como este tipo de red neuronal es entrenada. Backpropagation es una forma de entrenamiento supervisado. Cuando se usa un método en entrenamiento supervisado se debe proveer a la red de entradas y salidas esperadas. Estas salidas esperadas se compararan con las reales. Usando estas salidas esperadas el algoritmo de Backpropagation calcula un error y ajusta los pesos desde la capa de salida hacia todas las capas anteriores hasta la de entrada, [13].



Comentario [FMB4]: La imagen es Feed Forward o Kohonen?

Figura 4-12. Red de Feed Forward

Redes competitivas o mapas de auto-organización: Son redes monocapa o multicapa cuyo común denominador es postular algún tipo de competición entre unidades con el fin de conseguir que una de ellas quede activada y el resto no. Esto se consigue mediante aprendizaje no supervisado, presentando algún patrón de entrada y seleccionando la unidad cuyo patrón de pesos incidentes se parezca

más al patrón de entrada, reforzando dichas conexiones y debilitando las de las unidades perdedoras.

La competición entre unidades se puede conseguir simulando una característica neurofisiológica del córtex cerebral llamada inhibición lateral. Esto se logra postulando la existencia de conexiones inhibitorias intracapa y conexiones excitatorias intercapa, de tal manera que la presentación de un patrón de entrada tenderá a producir la activación de una única unidad y la inhibición del resto.

Las redes competitivas se usan típicamente como clasificadores de patrones, ya que cada unidad responde frente a grupos de patrones con características similares.

La principal crítica a estos modelos es que no poseen una de las características generales de las redes neuronales: la información no se halla distribuida entre todas las conexiones, la destrucción de una sola unidad provocaría la pérdida de la información relativa a todo un grupo o categoría de patrones. Para solventar este problema se han desarrollado los códigos demográficos, que representan cada categoría o grupo de patrones mediante un conjunto de unidades próximas entre sí, en vez de mediante una sola unidad.

Como ejemplo de redes competitivas podemos citar las redes de Kohonen (Kohonen,1988) y las arquitecturas ART (Adaptative Resonance Theory, Grossberg, 1987).

Kohonen Neural Network: La red neuronal de Kohonen fue nombrada asi, por honor a su creador Tuevo Kohonen. La red neuronal de Kohonen se caracteriza por ser de entrenamiento supervisado y la salida de esta red no consiste en muchas neuronas. Cuando un patrón se presenta a la red, una de las neuronas de salida es seleccionada como la neurona "vencedora". Esta neurona "vencedora" representa la respuesta de la red neuronal de Kohonen a los datos entrados.

La red neuronal de Kohonen trabaja diferente que la red Feed Forward. La red de Kohonen solo contiene una capa de entrada y una de salida. No existe la capa oculta.

La salida en este tipo de red es muy diferente a la salida de la red Feed Forward que esta dada por valores los mismos valores de entrada. Este no es el caso, puesto que esta red produce un valor que puede ser falso o verdadero, [13].

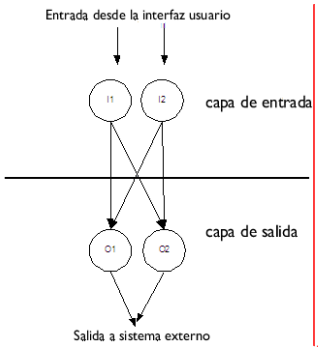


Figura 4-13. Red de Kohonen

Comentario [FMB5]: La imagen es Feed Forward o Kohonen?

Modelos de satisfacción de demanda o de adaptación probabilística: Son redes cuyo objetivo principal es alcanzar soluciones óptimas a problemas que exigen tomar en consideración un gran número de demandas simultáneas. Para ello parten de un estado dinámico o inestable y tratan de alcanzar un estado estable mediante un proceso de relajación, estado en el que la mayoría de las demandas sean satisfechas simultáneamente. Para evaluar el estado de la red se suele definir una "función de energía", de manera que el proceso de relajación consiste en la disminución del "estado energético" de la red.

Como máximos exponentes de este tipo están las redes de Hopfield (Hopfield, 1982) y las máquinas de Boltzman (Hinton y Sejnowski,1986), ambas ejemplos de memorias autoasociativas recurrentes, también denominadas redes retroasociativas. Las redes autoasociativas aprenden a reconstruir patrones de entrada, son útiles cuando tenemos información incompleta o distorsionada, para tratar de reproducir la información original.

> **Hopfield Neural Network:** La red neuronal de Hopfield es quizás la red neuronal más simple que existe. La red neuronal de Hopfield esta completamente conectada en una capa con auto asociaciones. Esto significa que tiene solamente una capa, y en esta, las neuronas están conectadas individualmente unas con otras. Esta red es de fácil entendimiento debido a su pequeño tamaño y esta limitada porque puede reconocer pocos patrones, [13].

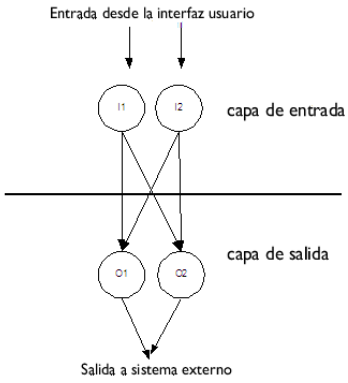


Figura 4-14. Red de Hopfield

Otras redes asociativas de pesos fijos: Además de las redes de adaptación probabilística hay otros tipos de redes que no requieren modificar los pesos de las conexiones. Se trata de memorias asociativas, utilizadas fundamentalmente para filtrar señales distorsionadas o incompletas y recuperar los patrones originales libres de ruido, como las memorias pro-asociativas y las redes de Hamming.

4.6.4 Clasificación usando redes neuronales

Debido a la necesidad de crear red neuronal de gran rapidez, exactitud y autosuficiente, pensamos que la red que mas cumple con nuestras expectativas para el sistema actual es la red neuronal de Feed Forward Backpropagation, puesto que posee características propias, como ser una red de arquitectura sencilla, de fácil construcción y además de esto ser una red de entrenamiento supervisado, puesto que debido al gran numero de diferentes siluetas que corresponde la forma humana, para su clasificación, es necesario supervisar la red

y entrenarla para enfrentar este tipo de situaciones donde por mas que sean dos contornos diferentes, si los dos son humanos, deben ser reconocidos.

Otra ventaja de la red de Feed Forward Backpropagation es ser una red capaz de seguir aprendiendo después del proceso de entrenamiento y aumentar la acertabilidad de sus decisiones en cada proceso de clasificación. Es por estas y muchas otras más razones, que pensamos esta red nos puede traer ventajas para el sistema actual.

Clasificación de formas, proceso de aprendizaje (off-line)

Para el aprendizaje se tomo un muestreo de imágenes con diferentes características pero donde prevalecían las características propias de la forma humana, sin embargo para aumentar la acertabilidad de decisión de la red, se tomo este muestreo con imágenes rotadas, escaladas, de diferente iluminación. Este proceso se hizo con el fin de mejorar la exactitud en la toma de decisiones de la red como el tiempo de respuesta de ella cuando funcione en tiempo real.

Clasificación de formas, proceso de reconocimiento (on-line)

Debido a que la red es preparada previamente en un modo offline (proceso de aprendizaje y entrenamiento), la red en esta etapa solamente clasifica en grupos y aprende por experiencias y autoasociaciones.

CAPITULO 5. IMPLEMENTACIÓN DEL SISTEMA

Cuando se pensó el sistema, las características que decidimos hacer imprescindibles para la implementación fueron la capacidad de que el sistema fuera extensible y que fuera eficiente. La extensibilidad, que se refiere al hecho de que el sistema pueda ser mejorado en el futuro para incorporar nuevas técnicas, la vemos necesaria debido a la gran cantidad de enfoques que se le puede dar a cada una de las etapas que hacen parte del proceso. Un ejemplo es el caso de los distintos tipos de filtrado o de detectores de movimiento. Haciendo al sistema extensible se disminuyen las repercusiones que se causan a el cuando se realiza un cambio o una actualización. La eficiencia, que toma su importancia cuando hablamos de tiempo de respuesta del sistema, la cual se ve afectada directamente por el uso eficiente de la memoria.

Estas son las dos características principales que han influido de gran manera en el diseño del sistema. Todas las soluciones que se crearon, están hechas pensadas en esto,

5.1 Implementación de la clase cámara

La implementación fue hecha usando la JMF (*Java Media Framework*) API para Java. La versión usada fue la JMF2.1.1.e. La idea básica que se siguió fue que la clase presentara las opciones de configurar el video y que tenga la habilidad de reproducir, capturar y presentar imágenes de video.

La clase es la única del sistema de este tipo y centraliza todas las funciones que estén relacionadas con los dispositivos de video y la presentación de imágenes.

Estas son las funciones que comprenden la interfaz de la clase:

```
public class Camara implements ... {
/**
       * Crea el objeto enlazado con el dispositivo especificado y
       * con tamaño default (352x288x24)
       */
public Camara(String newDevice) { ... }

/**
 * Crea el objeto enlazado con el dispositivo y tamaño default
 * (primer dispositivo de la listatamaño 352x288x24)
 */
public Camara() { ... }

/**
       * Crea el objeto enlazado con el dispositivo y configuración
       * especificadas
       */
public Camara(String newDevice, int w,int h, int b) { ... }

/**
       * Crea el objeto enlazado con el dispositivo por default y
       * con la configuración especificada
       */
public Camara( int wei, int hei, int bi )  { ... }

/**
       * Realiza toda la conexión e inicializacion de la camara
       * dejandola lista para usar
       */
      public void connect() { ... }

      /**
       * Detiene la captura de imagenes y libera al dispositivo
       */
      public void disconnect() { ... }

      /**
       * Devuelve la cantidad de frames por segundo que se
       * establecio a capturar
       */
      public int getFPS() { ... }
```

```
      /**
       * Devuelve el ancho de la imagen
       */
public int getWeight() { ... }

/**
       * Devuelve el alto de la imagen
       */
public int getHeight() { ... }

/**
       *  Devuelve todos los formatos de video soportados por el
       *   dispositivo
       */
private Format[] getDeviceFormat() { ... }

      /**
       *  Devuelve el formato de video establecido
       */
public Format getUserFormat() { ... }

/**
       * Devuelve un componente en el que se presentara el video
       */
public Component getComponent() { ... }

/**
       * Establece la cantidad de Frames por segundo a capturar
       */
public void setFPS(int fps) { ... }

/**
       * Retorna una array de Imagen con las imagenes capturadas para
       * su posterior tratamiento
       * @param formato Formato de las imagenes a entregar
       *
       * @return Imagen[] Arreglo de Imagen del formato especificado
       * @see recogn.core.FormatoImagen
       */
public Imagen[] getImagenes(FormatoImagen formato) { ... }

/**
       * Retorna una array de BufferedImage para su posterior
       * tratamiento
       *
       * @return BufferedImage[] Arreglo de BufferedImage
       * @see java.awt.image.BufferedImage;
       */
public BufferedImage[] getBufferedImage() { ... }
```

```
      public void controllerUpdate(ControllerEvent evt) { ... }
}
```

Un ejemplo básico del uso de la cámara lo podemos encontrar en `recogn.test.CamaraTest`:

```
//Se crea la clase
Camara cam = new Camara();
//Hacemos la conexion
cam.connect();
//Capturamos las imagenes
BufferedImage buffImg[] = cam.getBufferedImage();
//Hacemos lo que queramos con las imagenes
...
//Liberamos al dispositivo
cam.disconnect();
```

5.2 Implementación de las imágenes

Para la representación de las imágenes se implementó la clase Imagen. Los datos de esta están comprendidos en un arreglo de datos de tipo `short`. Este tipo de datos se eligió porque es el que computacionalmente menos costo trae dentro de los que permiten manejar valores entre 0 y 255. El número de arreglos y el número de dimensiones varían dependiendo el tipo de la imagen. Para la implementación de las imágenes se realizó una jerarquía de clases, lo cual nos permitió implementar los tipos de imágenes que se planeaban usar y aparte de dejar abierta la posibilidad de más tarde agregar otros tipos de imágenes, permite delegar a cada una manejar su arreglo de datos y su manipulación. La jerarquía de clases quedo de la siguiente forma:

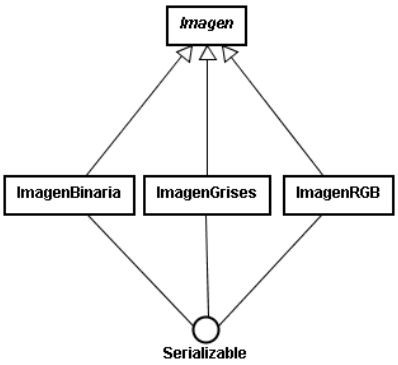


Figura 5-1. Diagrama de clases de las imágenes

Para apoyar a estas clases se creo la clase enumerada `FormatoImagen`. Esto con el fin de tener control sobre las posibles mejoras y actualizaciones, y que no se pierda compatibilidad. Otra clase que apoya a estas y sirve para realizar un resaltado en la imagen es la clase `Highlighter`.

De los tipos de representación de las imágenes, tenemos:

5.2.1 Imagen RGB

Para las imágenes RGB el color de cada píxel se codifica en tres componentes que corresponden a los colores: Rojo, Verde y Azul y donde cada componente tiene un valor de 0 a 255, donde cada valor representa la cantidad de color que compone al píxel (0 ausencia, 255 máxima composición).

```
short[][] rojo, verde, azul;
```

5.2.2 Imagen en escala de Grises

Una imagen en escala de grises codifica la intensidad de cada píxel dándole un valor de 0 a 255, donde 0 es un píxel oscuro (negro) y el 255 es un píxel con la mayor intensidad (blanco). Los valores entre estos dos valores representan los diferentes tipos de grises.

```
short[][] intensidad;
```

5.2.3 Imagen Binaria

Como la característica principal de estas imágenes es que la intensidad de cada píxel se codifica dándole solo uno de dos posibles valores, 0 o 1, donde el 0 indica un píxel sin intensidad (negro) y el 1 indica que es un píxel con la mayor intensidad (blanco), su implementación se hizo usando las características de la imagen en escala de grises pero con la restricción de que acepte solo alguno de los dos valores 0 y 255 (negro y blanco respectivamente).

En el diseño, también se contemplo la conversión entre los diferentes tipos de imágenes. Para esto, se dispusieron de constructores que aceptan el tipo de imagen desde la cual se puede crear la otra. Las conversiones que soporta el diseño encontramos las siguientes posibilidades:

Usando constructores:

- Crear una imagen RGB a partir de una BufferedImage
- Crear una imagen en Grises a partir de una imagen RGB

- Crear una imagen en Grises a partir de una BufferedImage
- Crear una imagen Binaria a partir de una imagen en Grises
- Crear una imagen Binaria a partir de una BufferedImage

A través de métodos:

- De imagen RGB a BufferedImage
- De imagen en Grises a RGB
- De imagen en Grises a BufferedImage
- De imagen Binaria a RGB
- De imagen Binaria a Grises
- De imagen Binaria a BufferedImage

Así de esta forma, es posible ir de una clase a la otra, haciendo o una o varias conversiones intermedias.

5.3 Implementación de los filtros

La implementación de los filtros se hizo utilizando arreglos bidimensionales, que para nuestra aplicación, fueron de 3 filas por 3 columnas.

Para poder realizar todas estas técnicas de filtrado espacial se hizo necesario crear varias clases que soportaran las estructuras de datos de los filtros y las operaciones que se realizarían con ellos. Fue por esto que para el filtrado se crearon las clases de:

- La clase que realiza la operación matemática de la convolución:

```
public class Convolucion {

    /**
     * Realiza la convolucion entre un filtro y una matriz de valores,
     * la cual debera contener valores que van desde 0 a 255.
     *
     * @param matriz Kernel(Filtro) que se le aplicara a la imagen
     * @param in Datos a los que se les aplicara el filtro
     * @return short[][] matriz con el resultado de la operacion
     */
   public static short[][] convoluciona(float[][] matriz, short[][]
in){
        int alto = in.length;
        int ancho = in[0].length;

        int ncol, bcol, nrow, brow;

        double sum = 0;
        short[][] res = new short[alto][ancho];

        for (int row = 1; row < (alto-1); row++) {
            for (int col = 1; col < (ancho-1); col++) {
                nrow = row+1;
                brow = row-1;
                bcol = col-1;
                ncol = col+1;

                res[row][col] = (short)(matriz[0][0]*in[brow][bcol] +
                                        matriz[0][1]*in[brow][col] +
                                        matriz[0][2]*in[brow][ncol] +
                                        matriz[1][0]*in[row][bcol] +
                                        matriz[1][1]*in[row][col] +
                                        matriz[1][2]*in[row][ncol] +
                                        matriz[2][0]*in[nrow][bcol] +
                                        matriz[2][1]*in[nrow][col] +
                                        matriz[2][2]*in[nrow][ncol] );

                if(res[row][col]<0) res[row][col]=0;
                else if(res[row][col]>255) res[row][col]=255;
            }
        }

        return res;
    }
}
```

- La jerarquía de clases de Filtros:

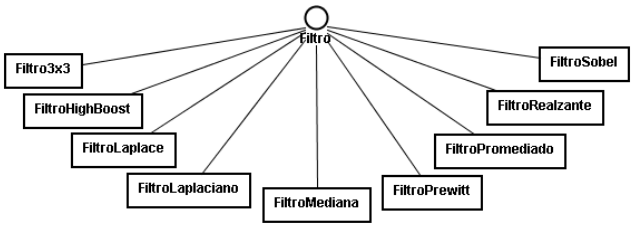


Figura 5-2. Diagrama de clases de los filtros

5.4 Implementación de la detección de movimiento

Para el soporte de los distintos tipos de detectores, se creo una interfaz que deben implementar todos los detectores que se implementen. El sistema ahora cuenta con 4 los cuales cada uno tiene su fundamento. La interfaz a implementar es:

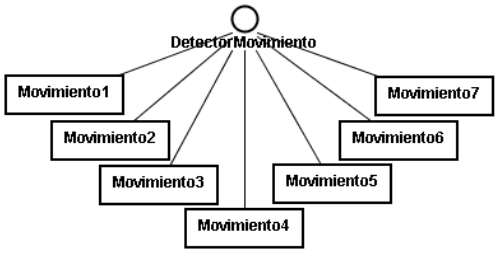


Figura 5-3. Diagrama de clases de los detectores

```
/**
 * Interfaz para implementar la detección de movimiento en un conjunto
 * de frames.
 **/
public interface DetectorMovimiento {

    /**
     * Realiza la deteccion de movimiento comparando los frames de
     * entrada
     * @param frames Arreglo de imagenes en grises (minimo 3)
     * @return boolean true o false si se ha detectado o no movimiento
     *
     * @see recogn.core.images.ImagenGrises
     */
    public boolean doDeteccion(ImagenGrises[] frames);

    /**
     * Devuelve una imagen que contiene el area de movimiento en
     * color blanco. Esta imagen corresponde a la region
     * detectada en la ultima llamada a la funcion <b>doDeteccion</B>
     *
     * @return ImagenBinaria Mascara del area de movimiento
     */
    public ImagenBinaria getRegion();


    /**
     * Establece a la imagen como el fondo inicial
     * @param ImagenGrises imagen con el fondo inicial
     * @see recogn.core.ImagenGrises
     */
    public void setFondo(ImagenGrises backImg);

    /**
     * Devuelve una imagen que contiene el fondo estimado
     *
     * @return ImagenGrises Fondo estimado con las imagenes procesadas
     */
    public ImagenGrises getFondo();
}
```

El detector que más usamos es el de tipo 4, que es el que realiza la detección de movimiento en un conjunto de frames usando la técnica de estimación de fondo con umbral fijo y realizando una verificación de validez simple.

La integración de todo lo podemos generalizar con el diagrama:

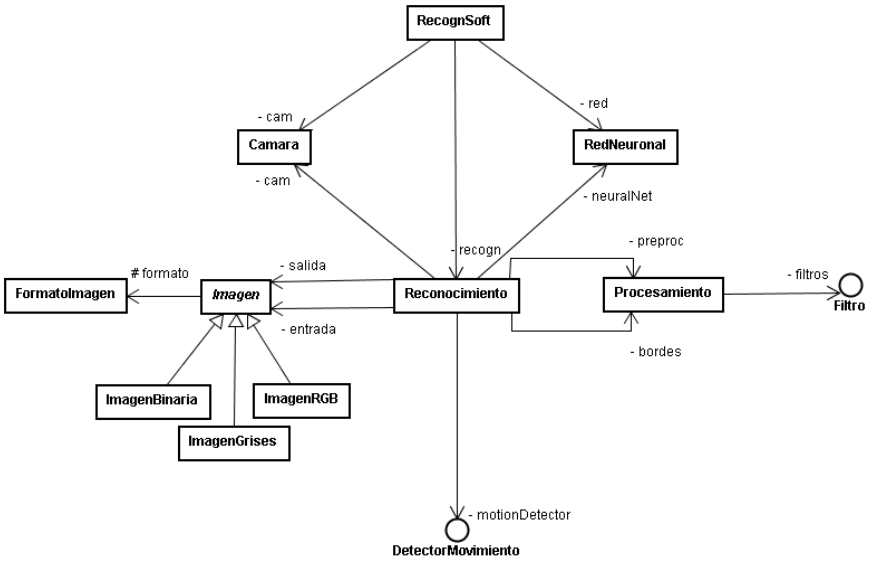


Figura 5-4. Diagrama general de clases

5.5 Implementación de los momentos invariantes

Para este proceso, se utilizaron como descriptores los momentos invariantes de Hu, ya que como habíamos dicho en capítulos anteriores son invariantes a las rotaciones y al escalado. Para realizar esto, implementamos una clase encargada del análisis individual de los píxeles y cálculo de los sus respectivos momentos invariantes, como se muestra a continuación:

```
    for( int i=0; i< Imagen.getPixeles(); i++) {
        y1=Pos[i].getPosY();
        x1=Pos[i].getPosX();
        pMom.setm00(pMom.getm00()+1);
        pMom.setm10(pMom.getm10()+x1);
        pMom.setm01(pMom.getm01()+y1);
    }
```

Que corresponde al cálculo de los momentos m_{pq} de una imagen cualquiera utilizando una clase que simula una estructura de datos para llevar el conteo individual de momentos y que nos ayuda a calcular como sigue los momentos centrales μ_{pq} de orden $p+q$.

```
x10=pMom.getm10()/pMom.getm00();
y01=pMom.getm01()/pMom.getm00();

for( int i=0; i< Imagen.getPixeles(); i++) {
    y1=Pos[i].getPosY();
    x1=Pos[i].getPosX();
    pCentrMom.setm00(pCentrMom.getm00()+1);
    pCentrMom.setm10(pCentrMom.getm10()+(x1-x10));
    pCentrMom.setm01(pCentrMom.getm01()+(y1-y01));
    pCentrMom.setm02(pCentrMom.getm02()+ SqrPar(y1-y01,2));
    pCentrMom.setm11(pCentrMom.getm11()+ (y1-y01)*(x1-x10));
    pCentrMom.setm20(pCentrMom.getm20()+ SqrPar(x1-x10,2));
    pCentrMom.setm30(pCentrMom.getm30()+ SqrPar(x1-x10,3));
    pCentrMom.setm12(pCentrMom.getm12()+ (SqrPar(y1-y01,2))*(x1-x10));
    pCentrMom.setm21(pCentrMom.getm21()+ (y1-y01)*(SqrPar(x1-x10,2)));
    pCentrMom.setm03(pCentrMom.getm03()+ SqrPar(y1-y01,3));
}
...
```

Ya teniendo μ_{pq} podemos calcular los momentos centrales normalizados que luego nos darán pasó al cálculo de los momentos invariantes de Hu de orden $p+q$.

```
    f1=Math.sqrt(pCentrMom.getm00());
    f2=pCentrMom.getm00();
    f3=f1*f2;
    f4=f2*f2;
    f5=f4*f1;
    pCentrNormMom.setm00(pCentrMom.getm00()/f2);
    pCentrNormMom.setm10(pCentrMom.getm10()/f3);
    pCentrNormMom.setm01(pCentrMom.getm01()/f3);
    pCentrNormMom.setm02(pCentrMom.getm02()/f4);
    pCentrNormMom.setm11(pCentrMom.getm11()/f4);
    pCentrNormMom.setm20(pCentrMom.getm20()/f4);
    pCentrNormMom.setm30(pCentrMom.getm30()/f5);
    pCentrNormMom.setm12(pCentrMom.getm12()/f5);
    pCentrNormMom.setm21(pCentrMom.getm21()/f5);
    pCentrNormMom.setm03(pCentrMom.getm03()/f5);
```

A continuación, hacemos el cálculo de los momentos invariantes de Hu:

```
Hu[0]=pCentrNormMom.getm20()+pCentrNormMom.getm02();
Hu[1]=SqrPar(pCentrNormMom.getm20()-pCentrNormMom.getm02(),2)+
             4*SqrPar(pCentrNormMom.getm11(),2);
Hu[2]=SqrPar(pCentrNormMom.getm30()-3*pCentrNormMom.getm12(),2)+
      SqrPar(3*pCentrNormMom.getm21()-pCentrNormMom.getm03(),2);
Hu[3]=SqrPar(pCentrNormMom.getm30()+pCentrNormMom.getm12(),2)+
      SqrPar(pCentrNormMom.getm21()+pCentrNormMom.getm03(),2);
Hu[4]=((pCentrNormMom.getm30()-3*pCentrNormMom.getm12())*
       (pCentrNormMom.getm30()+pCentrNormMom.getm12()))*
       (SqrPar(pCentrNormMom.getm30()+ pCentrNormMom.getm12(),2)-
       3*SqrPar(pCentrNormMom.getm21()+pCentrNormMom.getm03(),2))+
       ((3*pCentrNormMom.getm21()-pCentrNormMom.getm03())*
       (pCentrNormMom.getm21()+pCentrNormMom.getm03()))*
       (3*SqrPar(pCentrNormMom.getm30()+pCentrNormMom.getm12(),2)-
       SqrPar(pCentrNormMom.getm21()+pCentrNormMom.getm03(),2));
Hu[5]=((pCentrNormMom.getm20()-pCentrNormMom.getm02())*
      (SqrPar(pCentrNormMom.getm30()+pCentrNormMom.getm12(),2)-
      SqrPar(pCentrNormMom.getm21()+pCentrNormMom.getm03(),2))+
      4*pCentrNormMom.getm11()*
      ((pCentrNormMom.getm30()+pCentrNormMom.getm12())*
      (pCentrNormMom.getm21()+pCentrNormMom.getm03())));
Hu[6]=((3*pCentrNormMom.getm21()-pCentrNormMom.getm03())*
      (pCentrNormMom.getm30()+pCentrNormMom.getm12()))*
      SqrPar(pCentrNormMom.getm30()+pCentrNormMom.getm12(),2)-
      3*SqrPar(pCentrNormMom.getm21()+pCentrNormMom.getm03(),2))+
      ((3*pCentrNormMom.getm12()-pCentrNormMom.getm03())*
      (pCentrNormMom.getm21()+pCentrNormMom.getm03()))*
      (3*SqrPar(pCentrNormMom.getm30()+pCentrNormMom.getm12(),2)-
      SqrPar(pCentrNormMom.getm21()+pCentrNormMom.getm03(),2));
```

Luego de esto realizamos una normalización de los resultados para evitar resultados muy grandes y negativos que se salgan de los parámetros permitidos por la red.

```
for(int i=0; i<7; i++){
    Hu[i]=Math.abs((Math.log(Math.abs(Hu[i]))));
}
```

Estos momentos calculados como se ve a continuación son invariantes a la escala, a las rotaciones, etc.

5.6 Implementación de la red neuronal

Como se dijo anteriormente, la red que mas encaja en el diseño actual del sistema es la Feed Forward Backpropagation. A continuación mostramos con más detalle todo lo que correspondió a la implementación donde se realiza el diseño de la red y configuración del entrenamiento de esta.

Características de la Implementación: en el diseño actual y que esta por default en el sistema, la capa de entrada posee siete neuronas que corresponde a los descriptores del objeto (momentos invariantes de Hu). La capa oculta posee 5 neuronas para obtener un mejor porcentaje de acertabilidad en el entrenamiento (sin correr el riesgo de un sobreentrenamiento o un entrenamiento incompleto). La capa de salida posee 1 neuronas que corresponde a la respuesta de la red neuronal a los patrones que se le entraron. Siendo una neurona encargada de dar respuesta a las formas humanas y la otra las formas extrañas. En cuanto a las

funciones de transferencia que se usaron fueron la función de transferencia lineal para la conexión entre la capa de entrada y la oculta y la función de transferencia sigmoidal para la conexión entre las otras dos capaz.

La respuesta de la red neuronal esta dada por un número de coma flotante que representa el número caracteriza la forma que representa la red.

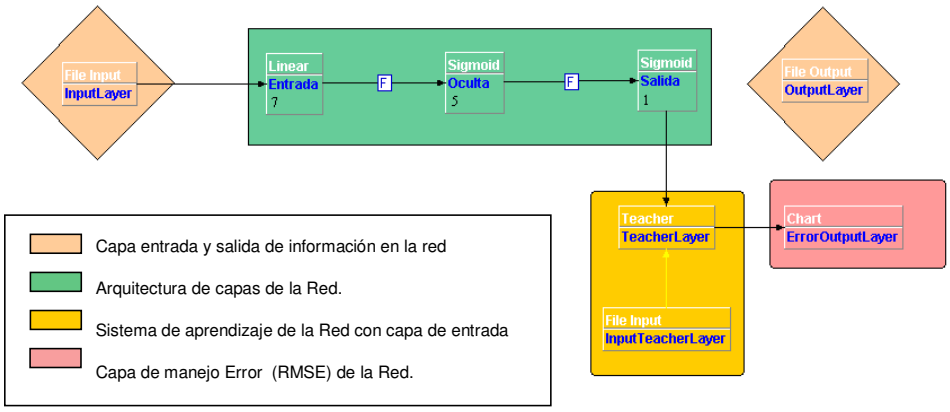


Figura 5-5. Diagrama general de la red

La red neuronal se construyó con ayuda de unas librerías de diseño de redes neuronales en java (JOONE). Para la construcción y uso de la red, se implementó una clase capaz de crear toda la arquitectura actual de la red neuronal, ya que en alto nivel es mas sencillo realizar tareas dinámicas sobre el mismo diseño y aprovechar las ventajas de este motor que nos permite diseñarla, entrenarla y guardarla todo por medio de funciones de fácil implementación.

A continuación mostramos el método donde se construye la red desde la capa de entrada y salida hasta el monitoreo del entrenamiento de la red que es indispensable para el buen funcionamiento de la misma cuando se ponga a prueba en tiempo real. Actualmente esta implementada como una clase aparte capaz de crear y entrenar varias redes de tipo Backpropagation.

```
      LinearLayer input = new LinearLayer();
      SigmoidLayer hidden = new SigmoidLayer();
      SigmoidLayer output = new SigmoidLayer();
      input.setRows(7);
      hidden.setRows(4);
      output.setRows(2);

      /* Las conexiones */
      FullSynapse synapse_IH = new FullSynapse(); /* Input -> Hidden*/
      FullSynapse synapse_HO = new FullSynapse(); /* Hidden -> Output*/
      input.addOutputSynapse(synapse_IH);
      hidden.addInputSynapse(synapse_IH);
      hidden.addOutputSynapse(synapse_HO);
      output.addInputSynapse(synapse_HO);

      /* Los componentes I/O */
      FileInputSynapse inputStream = new FileInputSynapse();
      inputStream.setFileName("C:\\patterns.txt");
      inputStream.setFirstCol(1);
inputStream.setAdvancedColumnSelector(columnSelector);
input.addInputSynapse(inputStream);

      /* El professor y los resultados esperados */
      TeachingSynapse trainer = new TeachingSynapse();
      MemoryInputSynapse samples = new MemoryInputSynapse();
      samples.setFirstRow(1);
      samples.setAdvancedColumnSelector("1"); //avance de los patrones
      samples.setInputArray(getDesired(patternsconunt,total));
      trainer.setDesired(samples);
      output.addOutputSynapse(trainer);

      /* El objeto Neuralnet */
      nnet.addLayer(input, NeuralNet.INPUT_LAYER);
      nnet.addLayer(hidden, NeuralNet.HIDDEN_LAYER);
      nnet.addLayer(output, NeuralNet.OUTPUT_LAYER);
      nnet.setTeacher(trainer);

      /* El monitor para dirigir el entrenamiento*/
      Monitor monitor = nnet.getMonitor();
      monitor.setLearning(true); /* The net must be trained */
      monitor.setLearningRate(0.8);
      monitor.setMomentum(0.3);
      monitor.setSupervisioned(true);
      monitor.setTotCicles(epoch); /*numero de veces en entrenamiento*/
      monitor.setTrainingPatterns(total);/*# colum. dentro del arc.*/
      monitor.setUseRMSE(true);
      monitor.addNeuralNetListener(this);
      nnet.start();
      nnet.getMonitor().Go();
```

Luego de esto ya con toda la red diseñada y entrenada procedemos a exportarla a un archivo externo para su posterior uso:

```
private void saveNeuralNet(String fileName, NeuralNet net){
    try {
        FileOutputStream stream = new FileOutputStream(fileName);
        ObjectOutputStream out = new ObjectOutputStream(stream);
        out.writeObject(net);
        out.close();
    }catch (Exception excp) {
        excp.printStackTrace();
    }
```

Para la restauración se utiliza una función capaz de subir toda la arquitectura como un objeto NeuralNet (objeto de una red creada), como sigue a continuación:

```
NeuralNet restoreNeuralNet(String fileName) {
    NeuralNet nnet = null;
    try {
        FileInputStream stream = new FileInputStream(fileName);
        ObjectInputStream inp = new ObjectInputStream(stream);
        nnet = (NeuralNet)inp.readObject();
    }catch (Exception excp) {
        excp.printStackTrace();
    }
    return nnet;
}
```

Ya teniendo la red entrenada y lista, solamente lo que se necesita es ponerla a prueba, y para esto se implementó un método cuya función principal es cargar la arquitectura en memoria y prepararla para que se le entren patrones a reconocer, como sigue a continuación:

```
...
Layer input = net.getInputLayer();
input.removeAllInputs();

//capa de entrada a memoria
MemoryInputSynapse memInp = new MemoryInputSynapse();
memInp.setFirstRow(1);
memInp.setAdvancedColumnSelector(columnSelector);
input.addInputSynapse(memInp);
memInp.setInputArray(inputArray);
Layer output = net.getOutputLayer();
output.removeAllOutputs();

//capa de salida a memoria
MemoryOutputSynapse memOut = new MemoryOutputSynapse();
output.addOutputSynapse(memOut);
net.getMonitor().setTotCicles(1); // Numero total de ciclos

//Se establece el número de patrones de entrenamiento
net.getMonitor().setTrainingPatterns(trainingPatterns);

//Se establece el tipo de trabajo de la red:
//Aprendizaje: true, Reconocimiento: false
net.getMonitor().setLearning(false);

//Se inicializa la red y se carga
net.start();

//Se corre la red con la configuración entrada
net.getMonitor().Go();

//Obtener los resultados de la red y hacemos algun procesamiento
for (int i=0; i < 1; i++) {
    pattern = memOut.getNextPattern();
}

//Detenemos la red
net.stop();
...
```

CAPITULO 6. ACERCA DEL SISTEMA

6.1 Características del Sistema

Debido a que el sistema se ha hecho de modo que soporte diferentes técnicas, se ha deseado que la interfaz permita hacer estos cambios, de modo que el usuario pueda manipular estas técnicas y sus parámetros.

Este sistema demarca la silueta de las figuras humanas reconocidas en color verde. En cambio, si este no es el caso, entonces no realiza ninguna advertencia.

De los parámetros generales que se pueden configurar se encuentran:

- La cámara que entregará las imágenes.
- El tiempo que actualización de las imágenes.
- Los filtros para mejorar la imagen.

6.2 Interfaz del sistema

Una de las características de este software es que todos los procesos se realizan de forma transparente para el usuario, es por eso que el sistema posee una interfaz sencilla que posee además, una barra de acceso rápido a las funciones principales del sistema, como por ejemplo la captura, configuración, visualización de alertas, etc. A continuación mostramos de manera general como esta compuesta esta ventana:

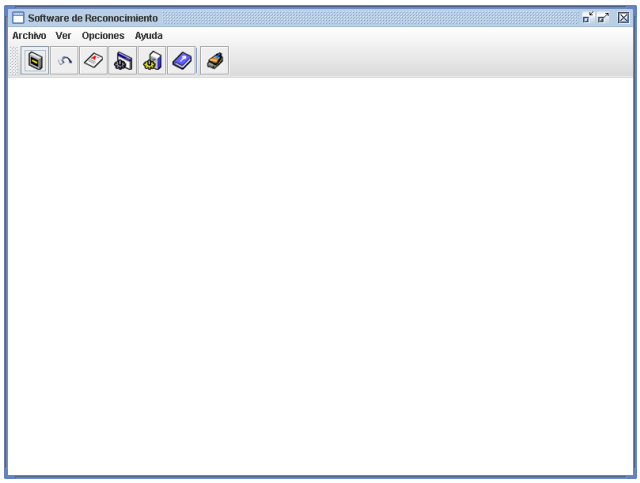


Figura 6-1 Interfaz principal de la aplicación

Como podemos observar, la interfaz del sistema es muy amigable para el usuario y las opciones son de muy fácil manejo con la barra de accesos que se posee en la parte superior de la ventana 

y que como se dijo anteriormente posee todas las características del sistema de una forma rápida y sencilla. La función de cada botón es:

1. **Video Monitor:** Al ser activado, inicia todas las operaciones del sistema (procesamiento, red neuronal y reconocimiento) y además muestra la ventana de captura donde se visualiza lo que esta siendo percibido por la cámara y el resultado del reconocimiento.

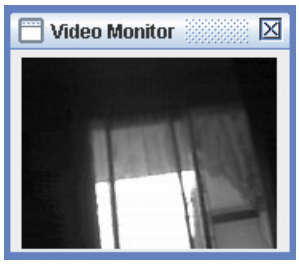


Figura 6-2 Ventana de captura.

2. Reestablecer: Esta función tiene como característica que establece el fondo del ambiente de la captura. Este proceso lo realiza la función captura al ser activada por primera vez pero si el fondo cambia en alguna característica (nuevos objetos en la escena), es necesario activarla para realizar el reconocimiento.

3. Alertas: Esta función permite visualizar las alertas generadas por el sistema desde que se inicio el reconocimiento. Además, genera una ventana donde se muestra la hora en que el sistema realizó y se detuvo la alerta, como la que se muestra a continuación:

Figura 6-3 Ventana de visualización de alertas.

4. Configuración: Muestra todas las opciones disponibles para configurar todos los procesos que intervienen en el reconocimiento del sistema. Estas son:

- **Configuración de la cámara:**

  Muestra las opciones configurables de este dispositivo como los son, el tipo de dispositivo de captura, formatos manejados por el (dimensiones de la resolución y el numero de píxeles) y además el numero de Frames por segundo con que se quiere trabajar en el sistema.

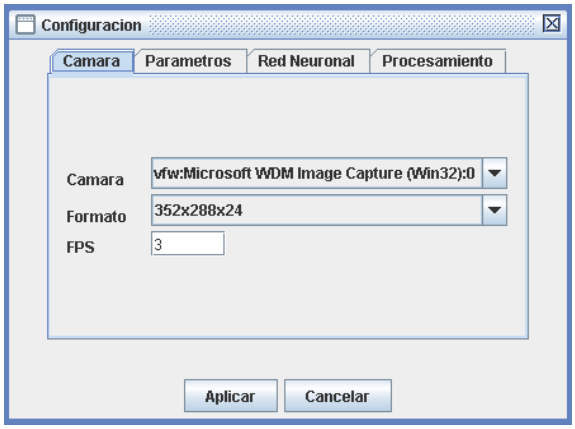


Figura 6-4 Ventana de configuración de la cámara.

- **Configuración de los parámetros:**

  Maneja los valores de los parámetros que se utilizan en el sistema para la labor validación de exactitud del reconocimiento y frecuencia en que se toman imágenes de la cámara.

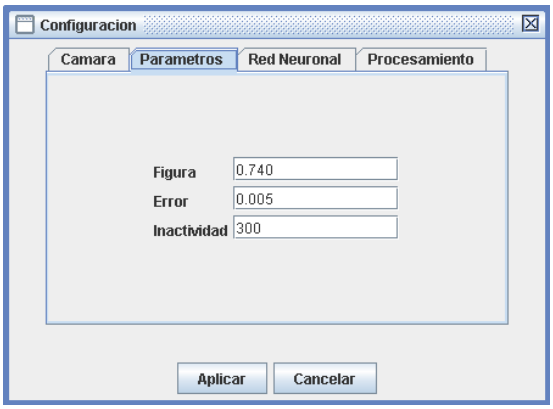


Figura 6-5 Configuración de los parámetros generales.

- **Configuración de la Red neuronal:**

  Posee los nombres de los archivos de registro generados por la red neuronal al ser entrenada, además el archivo físico donde se encuentra la red neuronal creada por el sistema.

Figura 6-6 Configuración de la red neuronal.

- **Configuración del procesamiento:**

  Esta opción brinda la posibilidad de configurar todos los filtros que se van a utilizar en el reconocimiento realizado por el sistema. Además posee la característica de poder permitirle al usuario escoger el algoritmo de detección de movimiento del sistema.

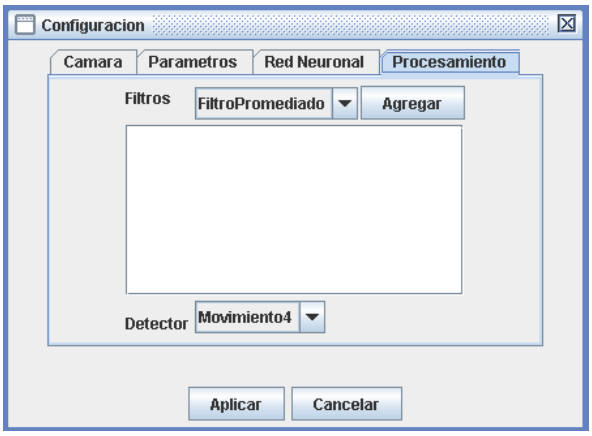


Figura 6-7 Configuración del procesamiento de imágenes.

5. Entrenamiento: Esta opción del sistema permite entrenar la red neuronal que va a utilizar el sistema para realizar el reconocimiento. Posee dos modos: modo entrenamiento que es donde se realiza la entrada del muestreo de imágenes para el aprendizaje y el modo reconocimiento que permite verificar si la red responde bien a todo el entrenamiento realizado por el usuario mediante el resaltado de los objetos reconocidos (opción resaltar siluetas). En ambos modos es posible realizar un mejoramiento

morfológico de la imagen en caso de que no se visualice de manera correcta la imagen.

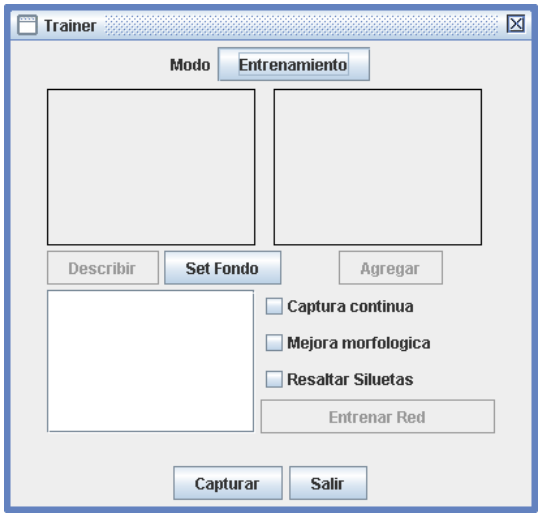


Figura 6-8 Ventana de depuración del sistema

6. Ayuda: Este botón al ser presionado, despliega el manual de usuario del sistema en una ventana. En este manual, se encuentra esta misma ayuda.

7. Limpiar Memoria: Esta función fuerza a que se llame al recolector de basura de Java, lo que ocasiona una mejora en el rendimiento, debido a que se libera memoria que no se este usando.

6.3 Configuración del sistema

En la realización de todas las pruebas y como configuración permanente adoptamos el siguiente hardware:

- Cámara Webcam Logitech
- Procesador Intel Pentium 4 de 2,8GHz
- Memoria RAM de 256 MB

Como configuración de la aplicación utilizamos:

- Imágenes de tamaño 160x130
- Profundidad de color de 24 bits
- Frame rate de 3 fps

Vale aclarar que esta configuración es personalizable a cualquiera otra soportada por la cámara, pero consideramos que esta configuración es la mas ideal debido al buen balance entre consumo de memoria, procesamiento y desempeño.

Otras de las consideraciones a tener en cuenta son: que la cámara siempre esté estática, que el lugar tenga una buena iluminación y que esta iluminación sea constante.

CAPITULO 7. RESULTADOS

Las siguientes páginas muestran algunas imágenes de ejemplo de diferentes pruebas que se le realizaron al sistema y algunas salidas de ejemplo del proceso de clasificación y reconocimiento.

La efectividad del sistema se ha encontrado estar entre el 90%, aún bajo diferentes condiciones de iluminación. Actualmente, el sistema no trabaja muy bien en condiciones de escasa iluminación, debido a que las áreas oscuras interfieren con las medidas de los valores RGB en las imágenes. Por la misma razón, el sistema no trabaja bien con la iluminación ambiental en sitios cerrados temprano en la mañana o al finalizar el atardecer. Los excesos de iluminación también producen algunos resultados no deseados. Tampoco tiene buenos resultados en el reconocimiento de más de una persona, ni en posturas diferentes a las de caminar. Sin embargo estos son problemas que pueden ser resueltos.

También es notable la manera como realizamos el entrenamiento, lo cual nos parece una manera muy fácil, sencilla y novedosa de realizar, debido a que no necesitamos cerrar el sistema, no se necesita realizar ninguna acción compleja y no hay que tener ningún conocimiento sobre la red neuronal. Para lograr esto, hemos usado las mismas capacidades del sistema y el entrenamiento lo realizamos dentro de él mismo y “en tiempo real”. El procedimiento es el siguiente: luego de entrar al depurador del sistema, seleccionamos captura continúa, lo colocamos en modo entrenamiento y empezamos a realizar el muestreo por medio del botón describir. El mismo sistema con los diferentes tipos de procesamiento y

con las imágenes que está obteniendo de la cámara, empieza a tomar las figuras y a describirlas, mostrando los datos en la caja de texto que ahí aparece. Una vez satisfechos con las muestras obtenidas, detenemos el muestreo haciendo clic otra vez en el botón describir. Luego presionamos agregar, para formatear e inicializar todos los datos necesarios y luego presionamos "Entrenar Red" (para este momento debe estar habilitado). Enseguida se mostrará información relativa al entrenamiento, a través de los valores de error relativo entre las muestras y el estado del proceso. Luego de unos minutos (todo depende del número de muestras), se terminará el entrenamiento (aparece el mensaje entrenamiento finalizado) y ya estará lista la red neuronal para empezar a realizar el reconocimiento.

Vale decir que para nuestro sistema, el entrenamiento lo realizamos haciendo este proceso mientras una persona caminaba frente a la cámara, con el fin de que se muestrearan varias figuras humanas de una persona cuando camina. El resultado fue exitoso, y es el que hemos utilizado desde entonces.

En la figura Figura 7-1 podemos ver una prueba del sistema. Aquí una persona pasa caminando por el frente. Lo que se aprecia en la imagen de la izquierda es la imagen actual de entrada, que captura la videocámara; en la imagen de la derecha es la región que se ha detectado estar en movimiento.

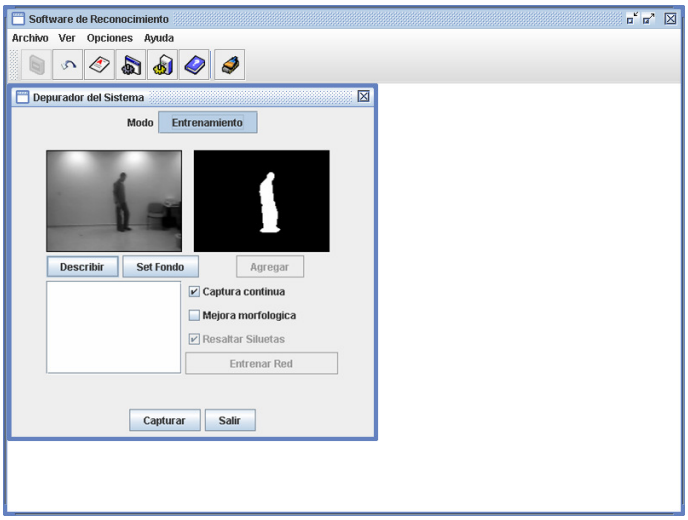


Figura 7-1. Resultados de la detección de movimiento.

En las siguientes figuras se muestran pruebas realizadas al sistema, para comprobar el proceso de reconocimiento. Nótese como el sistema cuando reconoce a una persona, la resalta.

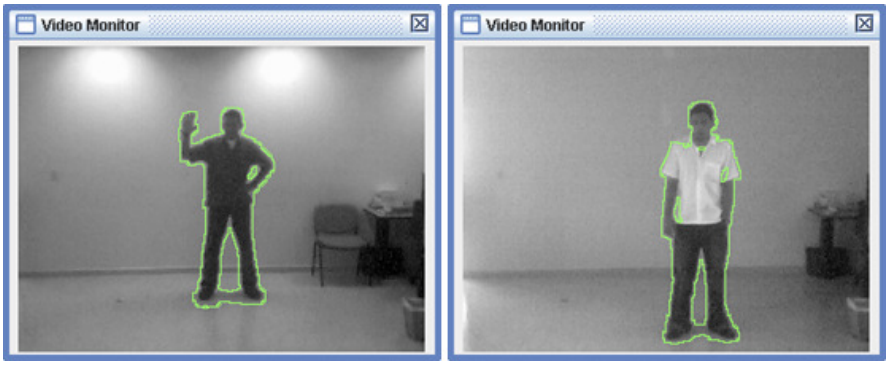


Figura 7-2. Resultado del reconocimiento de personas

Figura 7-3. Resultado del reconocimiento de no-personas

Figura 7-4. Resultados del reconocimiento de poses no entrenadas

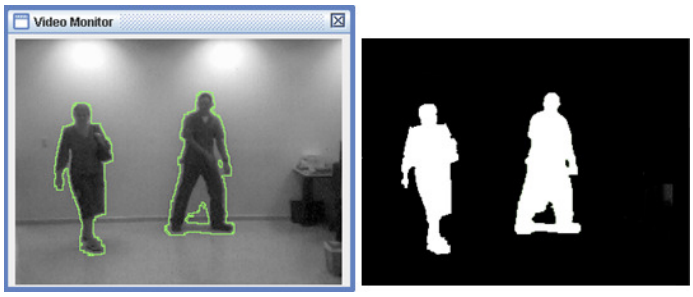


Figura 7-5. Resultado del reconocimiento de varias personas

Vale decir que el resultado que presenta la figura Figura 7-5 es un resultado por error. Dicho de otra forma, el sistema siempre va a generar una alarma cuando dos objetos que se han movido aparezcan dentro de una misma imagen, debido a que en la imagen de detección se encuentra más de una región (como se puede ver en la imagen de la derecha). Esto sucede debido a que el descriptor trabaja sobre las dos regiones, haciendo por consiguiente cálculos errados que describen más de una figura en un mismo instante. Generalmente estos valores que son muy particulares, son aproximados por la red al valor de salida de una persona, por lo cual el sistema cree que es una persona. Esto se puede resolver extrayendo las dos regiones y describirlas por separado, y luego entregando cada resultado (por separado) a la red neuronal.

CONCLUSIONES

Los objetivos planteados al principio del proyecto se han cumplido a satisfacción sin ningún por menor, teniendo como resultado la aplicación de reconocimiento de movimiento de humanos en ambientes no controlados con un alto porcentaje de acertabilidad. Obtener este alto valor de acertabilidad solo se logró mediante el uso de todas las técnicas que se investigaron acerca tanto del procesamiento de imágenes como las de codificación y manejo del sistema de redes neuronales.

Uno de los grandes problemas con los que cuenta la aplicación (y la mayoría de aplicaciones de visión por computador) es la iluminación. Esta característica es inherente al hecho de utilizar imágenes en formato RGB capturadas por una cámara de video, y que se atenúa en ambientes de poca iluminación o de iluminación variable. Esto, para nuestra aplicación, causa la aparición de falsas alarmas o de objetos detectados de manera incompleta. Una de las formas de tratar este problema es usando imágenes HSI, que tratan a la componente de iluminación por separado de las del color. Sin embargo, esto tampoco lo elimina del todo.

Las técnicas de mejora de imágenes son técnicas que son específicas a la aplicación. Cada problema implica una solución particular al contexto del

problema. Aunque esto no es un problema en nuestro caso, si pudimos darnos cuenta de que si se llega a presentar el caso, se haría difícil resolver otros problemas con este mismo diseño. Por eso, hacer una aplicación de visión artificial que trate más de un problema se hace difícil.

Una mejora muy sencilla de implementar al algoritmo de detección por estimación de fondo es la de inicializar el fondo con la imagen del fondo que se abordará. Esto, que no implica mayores retos (la imagen se puede capturar desde antes de empezar a realizar la detección), trae una mejora muy significativa al desempeño del sistema.

Otra posible mejora al sistema es la implementación de técnicas de enlazado de bordes para el detección de los bordes de la figura. Las técnicas implementadas detectan las discontinuidades de intensidad. Idealmente, estas deberían obtener solamente píxeles situados entre el limite de las regiones. Sin embargo, en la práctica esto generalmente no se da debido al ruido o a las interrupciones en el límite por iluminación no uniforme.

Para poder reconocer más de una persona simultáneamente (cada una en una región de detección diferente) es necesario implementar técnicas de conectividad y agrupación de píxeles, de extracción de regiones y de división de imágenes.

Para poder reconocer grupos de personas (más de una persona en una misma región de detección) es necesario hacer un entrenamiento de la red donde se le enseñe a reconocer esto.

El uso la red neuronal de Feed Forward Backpropagation para reconocer el movimiento de humanos fue exitosa. Se tomó un muestreo de imágenes para el proceso de aprendizaje de la red y se logró que esta superara las expectativas que se tenían acerca del tiempo de respuesta y de la exactitud del sistema frente a una condición en tiempo real.

ANEXO 1. IMAGE CAPTURE FROM WEBCAMS USING JAVA MEDIA FRAMEWORK API

Introduction

Java Media Framework (JMF) API 1.0 was intended to enable the playback of audio, video, and other time-based media from Java technology applets and applications. Version 2.0 added the ability to capture, stream, and transcode multiple media formats. This article shows how the JMF API can be used to capture single images from a standard webcam.

In addition to the standard Java Development Kit (JDK), a separate, platform-specific performance pack is required for the JMF API. (See the <JMF page> <http://java.sun.com/products/java-media/jmf/index.html> for further details.) Due to the required support for hardware, the example code in this article was developed and tested on the Microsoft Windows platform using a Logitech USB Webcam.

Device Names

In order to use a particular media device, you must first know the name by which the JMF API references the device. There are two ways of finding the appropriate name, depending on the application you are developing.

The easiest way is to start the JMStudio application that comes with the JMF API. Once the application is running, select Capture from the File menu. This will present you with the options for configuring the video and sound devices. On Microsoft Windows, a typical name for a webcam would be something like vfw:Logitech USB Video Camera:0.

The second way to get this information is programmatically (this can be useful if you want the user of the application to select the device to be used).

The CaptureDeviceManager class is part of the JMF API and provides access to a list of the capture devices available on a system. In order to retrieve information about a specific kind of device, a format object is passed as an argument to the getDeviceList method. Code Example 1 shows a code sample that will print a list of the names of RGB video

devices available on the system.

CODE EXAMPLE:

Code Example 1:

```
 public void printRGBDevices() {

   /*  Create a new default RGB format object (see the JMF
documentation

    *  for other supported formats)

    */

   RGBformat rgbFormat = new RGBFormat();


   /*  Use the Format object to retrieve a Vector of CaptureDeviceInfo

    *  objects from the CaptureDeviceManager

    */

   Vector videoDevs = CaptureDeviceManager.getDeviceList(rgbFormat);


   /*  Loop through the results and print them out for the user  */

   for (int i = 0; i <videoDevs.size(); i++) {

     CaptureDeviceInfo cdi =
(CaptureDeviceInfo)videoDevs.elementAt(i);


     System.out.println("RGB Device[" + i + "]: " + cdi.getName());

   }

 }
```

In order to use a capture device we need a CaptureDeviceInfo object. This can either be taken from the list retrieved in the code example or, if using a specific device name, through the getDeviceName method of the CaptureDeviceManager.

Getting The Right Format

Most webcams support a number of different formats; things like the resolution, color depth, and frames per second can be adjusted. The formats supported by a particular device can be determined using the getFormats method of the CaptureDeviceInfo class. From this list the appropriate format can be selected. Code Example 2 shows sample code for selecting a format.

Code Example 2:

```
/*  We are looking for a format of 160 x 120 pixels with 24 bit
 *  colour depth, RGB
 */

Dimension wantRes = new Dimension(160, 120);

int wantDepth = 24;


/*  Get the Formats supported by the device. (Real code will check
 * for a null from getDevice)
 */

CaptureDeviceInfo device =
   CaptureDeviceManager.getDevice("webcam:0");

Format[] fmts = device.getFormats();

RGBFormat userFormat = null;


for (int i = 0; i <fmts.length; i++) {

  if (fmts[i] instanceof RGBFormat) {

    userFormat = (RGBFormat)fmts[i];

    Dimension d = userFormat.getSize();
```

```
      int cDepth = userFormat.getBitsPerPixel();
      if (wantRes.equals(d) && cDepth == wantDepth)
        break;
      userFormat = null;
    }
  }
```

Getting A DataSource

To grab images from the webcam, we need to need to get a JMF PushBufferStream object. Getting this requires a little work. First, we need a MediaLocator object, which describes the location of media content. We then use this to create a DataSource using the createDataSource method of the Manager class. The Manager class is a general access point provided by the JMF for obtaining system dependant resources.

Before we can use this DataSource, we must ensure that it will provide information in the format we need. Code Example 3 shows a code fragment for creating a DataSource configured for a specific format.

Code Example 3:

```
MediaLocator loc = device.getLocator();
DataSource formattedSource = null;

try {
  formattedSource = Manager.createDataSource(loc);
} catch (IOException ioe) {
  System.out.println("IO Error creating dataSource");
  System.exit(1);
} catch (NoDataSourceException ndse) {
```

```
     System.out.println("Unable to create dataSource");

     System.exit(1);

   }



   /*  Setting the format is rather complicated. Firstly we need to get
    *  the format controls from the dataSource we just created. In
order
    *  to do this we need a reference to an object implementing the
    *  CaptureDevice interface (which DataSource objects can).
    */

   if (!(formattedSource instanceof CaptureDevice)) {

     System.out.println("DataSource not a CaptureDevice");

     System.exit(1);

   }



   FormatControl[] fmtControls =
(CaptureDevice)formattedSource).getFormatControls();

    if (fmtControls == null || fmtControls.length == 0) {
     System.out.println("No FormatControl available");
     System.exit(1);
   }

   Format setFormat = null;


   /*  Now we need to loop through the available FormatControls and try
    *  to set the format to the one we want. According to the
documentation
    *  even though this may appear to work, it may fail later on. Since
    *  we know that the format is supported we hope that this won't
happen
    */

   for (int i = 0; i <fmtControls.length; i++) {

     if (fmtControls[i] == null)
       continue;


     if ((setFormat = fmtControls[i].setFormat(userFormat)) != null)
         break;
```

```
    }


    /*  Throw an exception if we couldn't set the format  */

    if (setFormat == null) {
      System.out.println("Failed to set device to specified mode");
      System.out.exit(1);
    }

    /*  Connect to the DataSource  */

    try {

      formattedSource.connect();

    } catch (IOException ioe) {

      System.out.println("Unable to connect to DataSource");
      System.out.exit(1);

    }

    System.out.println("Data source created and format set");
```

Getting A Processor

We now have a usable DataSource. However, we're not done yet. The JMF API allows us to use this DataSource either to display the media (via a player) or manipulate the information (via a processor). Again, we use the Manager to create a processor object for us, passing our DataSource as an argument. Before we can use this processor, it must be in the realized state. Calling the realize method of the processor completes construction of the media dependant parts. Since this method returns immediately, it is necessary to write some code to wait for the processor to reach the realized state. This involves using a listener to wait for the processor to indicate the state change. The code fragment for this is shown in Code Examples 4 and 5.

Code Example 4:

```
Object stateLock = new Object();  //  This must be a global
```

```
   //  Method code

   Processor deviceProc = null;

   try {

     deviceProc = Manager.createProcessor(formattedSource);

   } catch (IOException ioe) {

     System.out.println("Unable to get Processor for device: " +
ioe.getMessage());
     System.exit(1);

   } catch (NoProcessorException npe) {

     System.out.println("Unable to get Processor for device: " +
npe.getMessage());
     System.exit(1);

   }

   /*  In order to use the controller we have to put it in the realized
    *  state. We do this by calling the realize method, but this will
    *  return immediately so we must register a listener (this class)
    *  to be notified when the controller is ready. The class
    *  containing this code must implement the ControllerListener
    *  interface.
    */

   deviceProc.addControllerListener(this);

   deviceProc.realize();



   while (deviceProc.getState() != Controller.Realized) {

     synchronized (stateLock) {

       try {

         stateLock.wait();

       } catch (InterruptedException ie) {

         System.out.println("Device failed to get to realized state");

         System.exit(1);

       }
```

```
    }
  }
  /*  Finally, start the processor  */
  deviceProc.start();
```

Code Example 5:

```
/**
 *  The ControllerListener interface only contains one method,
 *  controllerUpdate
 **/
public void controllerUpdate(ControllerEvent ce) {
  if (ce instanceof RealizeCompleteEvent) {
    System.out.println("Realize transition completed");
    synchronized (stateLock) {
      stateLock.notifyAll();
    }
  }
}
```

The PushBufferStream

Now that the processor is up and running, we can get access to a PushBufferDataSource that can then provide us with a PushBufferStream. The PushBufferDataSource is a special form of a DataSource that manages data in the form of push streams that pass buffer objects. The PushBufferStream provides a way of accessing the buffers that are passed through the stream; in this case, each buffer represents a frame from the webcam.

Code Sample 6 shows the code fragment to complete the construction of the PushBufferStream.

Code Example 6:

```
 private PushBufferStream camStream;  //  Global variable
 private BufferToImage converter;     //  Global variable

   //  Method code

   PushBufferDataSource source = null;

   try {
     source = (PushBufferDataSource)deviceProc.getDataOutput();

   } catch (NotRealizedError nre) {

     /*  Should never happen  */
     throw new VisionJMFException("Internal error: processor not
realized");
   }

   /*  getStreams returns all available streams. For frame grabbing
    *  this is usually only one, but we need to check, just in case
    */

   PushBufferStream[] streams = source.getStreams();

   /*  Pick the first RGBFormat stream available (shouldn't be more
    *  than one)
    */

   for (int i = 0; i <streams.length; i++) {

     if (streams[i].getFormat() instanceof RGBFormat) {

       camStream = streams[i];

       RGBFormat rgbf = (RGBFormat)streams[i].getFormat();

       converter = new BufferToImage(rgbf);

     }

   }
```

Getting An Image

What we end up with is a PushBufferStream (camStream) from which we can read Buffer

objects that represent frames from the webcam. We also create a BufferToImage object which will allow us to convert these buffers to an AWT Image, which can either be manipulated or displayed.

Code Example 7 shows a method that can be used to deliver an Image to an application.

Code Example 7:

```
/**
 *  Get an image from the camera
 **/
public Image getImage() {
  Buffer b = new Buffer();
  try {
    camStream.read(b);
  } catch (IOException ioe) {
    System.out.println("Unable to capture frame from camera");
    return null;
  }
  Image i = converter.createImage(b);
  return i;
}
```

A complete class, called FrameGrabber, is available as a zip file download at http://java.sun.com/dev/evangcentral/totallytech/FrameGrabber.zip. This uses a properties file to specify the device name and format to use.

Please direct any comments or questions to simon.ritter@sun.com.

The use of the code above, is bound by the license terms founds at http://developers.sun.com/techtopics/mobility/berkeley_license.html.

Anexo 2. **USING JOONE FOR ARTIFICIAL INTELLIGENCE PROGRAMMING**

Introduction

Few programmers have not been intrigued by Artificial Intelligence programming at one point or another. Many programmers who become interested in AI are quickly put off by the complexity of the algorithms involved. In this article, we will examine an open source project for Java that can simplify much of this complexity.

The Java Object Oriented Neural Network (JOONE) is an open source project that offers a highly adaptable neural network for Java programmers. The JOONE project source code is covered by a Lesser GNU Public License (LGPL). In a nutshell, this means that the source code is freely available and you need to pay no royalties to use JOONE. JOONE can be downloaded from http://joone.sourceforge.net/.

JOONE can allow you to create neural networks easily from a Java program. JOONE supports many features, such as multithreading and distributed processing. This means that JOONE can take advantage of multiprocessor computers and multiple computers to distribute the processing load.

Neural Networks

JOONE implements an artificial neural network in Java. An artificial neural network seeks to emulate the function of the biological neural network that makes up the brains found in nearly all higher life forms found on Earth. Neural networks are made up of neurons. A diagram of an actual neuron is shown in Figure 1.

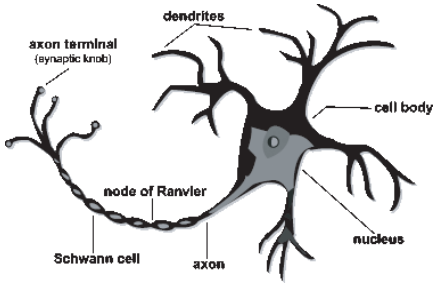


Figure 1: A biological neuron

As you can see from Figure 1, the neuron is made up of a core cell and several long connectors, which are called synapses. These synapses are how the neurons are connected amongst themselves. Neural networks, both biological and artificial, work by transferring signals from neuron to neuron across the synapses.

Using JOONE

In this article, you will be shown a simple example of how to use JOONE. The topic of neural networks is very broad and covers many different applications. In this article, we will show you how to use JOONE to solve a very simple pattern recognition problem. Pattern recognition is a very common use for neural networks.

Pattern recognition presents the neural network with a pattern, to see whether the neural network is able to recognize that pattern. The pattern should be able to be distorted in some way and the neural network still is able to recognize it. This is similar to a human's ability to recognize something such as a traffic signal. The human should be able to recognize a traffic signal in the rain, daylight, or night. Even though each of these images looks considerably different, the human mind is able to determine that they are the same image.

When programming JOONE, you are generally working with two types of objects. You are given Neuron layer objects that represent a layer of one or more neuron that share similar characteristics. Neural networks usually will have either one or two layers of neurons. These layers are connected together by synapses. The synapses carry the pattern, which is to be recognized, from layer to layer.

Synapses do not just transmit the pattern from one neuron layer to the next. Synapses will develop biases towards elements of the pattern. These biases will cause certain elements of the pattern to be transmitted less effectively to the next layer than they would otherwise be. These biases, which are usually called weights, form the memory of the neural network. By adjusting the weights, which are stored in synapses, the behavior of the neural network is altered.

Synapses also play another role in JOONE. In JOONE, it is useful to think of synapses as data conduits. Just as synapses carry patterns from one neuron layer to another,

specialized versions of the synapse are used to carry patterns both into and out of the neural network. You will now be shown how a simple single layer neural network can be constructed to recognize a pattern.

Training the Neural Network

For the purposes of the article, we will teach JOONE to recognize a very simple pattern. For this pattern, we will examine a binary boolean operation, such as XOR. The XOR operation's truth table is summarized below.

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

As you can see from the preceding table, the XOR operator will only be true, indicated by a value of one, when X and Y hold different values. In all other cases, the XOR operator evaluates to false, indicated by a zero. By default, JOONE takes its input from text files stored on your system. These text files are read by using a special synapse called the FileInputSynapse. To train for the XOR problem, you must construct an input file that contains the data shown above. This file is shown in Listing 1.

Listing 1: Input file for the XOR problem

```
0.0;0.0;0.0
0.0;1.0;1.0
1.0;0.0;1.0
1.0;1.0;0.0
```

We will now examine a simple program that teaches JOONE to recognize the XOR operation and produce the correct result. We will now examine the process that must be carried out to train the neural network. The process of training involves presenting the XOR problem to the neural network and observing the result. If the result is not what was

expected, the training algorithm will adjust the weights, stored in the synapses. The difference between the actual output of the neural network and the anticipated output is called the error. Training will continue until the error falls below an acceptable level. This level is generally a percent, such as 10%. We will now examine the code that must be used to train a neural network.

The training process begins by setting up the neural network. The input, hidden, and output layers must all be created.

```
// First, creates the three Layers
input = new SigmoidLayer();
hidden = new SigmoidLayer();
output = new SigmoidLayer();
```

As you can see, each of the layers are created using the JOONE object SigmoidLayer. Sigmoid layers produce an output based on the natural logarithm. JOONE contains additional layers, other than the sigmoid layer type, that you may choose to use.

Next, each of these layers is given a name. These names will be helpful to later identify the layer during debugging.

```
input.setLayerName("input");
hidden.setLayerName("hidden");
output.setLayerName("output");
```

Each layer must now be defined. We will specify the number of "rows" in each of the layers. This number of rows corresponds to the number of neurons in the layer.

```
input.setRows(2);
hidden.setRows(3);
output.setRows(1);
```

As you can see from the preceding code, the input layer has two neurons, the hidden layer has three hidden neurons, and the output layer contains one neuron. It makes sense for the neural network to contain two input neurons and one output neuron because the XOR operator accepts two parameters and results in one value.

To make use of the neuron layers, we must also construct synapses. In this example, we will have several synapses. These synapses are crated with the following lines of code.

```
// input -> hidden conn.
FullSynapse synapse_IH = new FullSynapse();
// hidden -> output conn.
FullSynapse synapse_HO = new FullSynapse();
```

Just as was the case with the neuron layers, synapses can also be given names to assist in debugging. The following lines of code name the newly created synapses.

```
synapse_IH.setName("IH");
synapse_HO.setName("HO");
```

Finally, we must connect the synapses to the appropriate neuron layers. The following lines of code do this.

```
// Connect the input layer with the hidden layer
input.addOutputSynapse(synapse_IH);
hidden.addInputSynapse(synapse_IH);

// Connect the hidden layer with the output layer
hidden.addOutputSynapse(synapse_HO);
output.addInputSynapse(synapse_HO);
```

Now that the neural network has been created, we must create a Monitor object that will regulate the neural network. The following lines of code create the Monitor object.

```
// Create the Monitor object and set the learning parameters
monitor = new Monitor();
monitor.setLearningRate(0.8);
monitor.setMomentum(0.3);
```

The learning rate and momentum are parameters that are used to specify how the training will occur. JOONE makes use of the backpropagation learning method. For more information on the learning rate or the momentum, you should refer to the backpropagation algorithm.

This monitor object should be assigned to each of the neuron layers. The following lines of code do this.

```
input.setMonitor(monitor);
hidden.setMonitor(monitor);
output.setMonitor(monitor);
```

Like many of the Java objects themselves, the JOONE monitor allows listeners to be added to it. As training progresses, JOONE will notify the listeners as to the progress of the training. For this simple example, we use:

```
monitor.addNeuralNetListener(this);
```

We must now set up the input synapse. As previously mentioned, we will use a FileInputSynapse to read from a disk file. Disk files are not the only sort of input that JOONE can accept. JOONE is very flexible with regard to the input sources that it will

accept. To cause JOONE to be able to accept other input types, you simply must create a new specialized synapse to accept the input. For this example, we will simply use the FileInputSynapse. The FileInputSynapse is first instantiated.

```
inputStream = new FileInputSynapse();
```

Next, the FileInputSynapse must be informed of which columns to be used. The file shown in Listing 1 uses the first two columns as the inputs. The following lines of code set up the first two columns as the input to the neural network.

```
// The first two columns contain the input values
inputStream.setFirstCol(1);
inputStream.setLastCol(2);
```

Next, we must provide the name to the input file. This name will come directly from the user interface. An edit control was provided to collect the name of the input file. The following lines of code set the filename for the FileInputSynapse.

```
// This is the file that contains the input data
inputStream.setFileName(inputFile.getText());
```

As previously mentioned, a synapse is just a conduit for data to travel between neuron layers. The FileInputSynapse is the conduit through which data enters the neural network. To facilitate this, we must add the FileInputSynapse to the input layer of the neural network. This is done by the following line.

```
input.addInputSynapse(inputStream);
```

Now that the neural network is set up, we must create a trainer and monitor. The trainer is used to train the neural network because the monitor runs the neural network through a set number of training iterations. For each training iteration, data is presented to the neural network and the results are observed. The neural network's weights, which are stored in the synapse connection that go between the neuron layers, will be adjusted based on this error. As training progresses, this error level will drop. The following lines of code set up the trainer and attach it to the monitor.

```
trainer = new TeachingSynapse();
trainer.setMonitor(monitor);
```

You will recall that the input file provided in Listing 1 contains three columns. So far, we have only used the first two columns, which specify the input to the neural network. The third column contains the expected output when the neural network is presented with the numbers in the first column. We must provide the trainer access to this column so that the

error can be determined. The error is the difference between the actual output of the neural network and this expected output. The following lines of code create another FileInputSynapse and prepare it to read from the same input file as before.

```
// Setting of the file containing the desired responses,
// provided by a FileInputSynapse
samples = new FileInputSynapse();
samples.setFileName(inputFile.getText());
```

This time, we would like to point the FileInputSynapse at the third column. The following lines of code do this and then set the trainer to use this FileInputSynapse.

```
// The output values are on the third column of the file
samples.setFirstCol(3);
samples.setLastCol(3);
trainer.setDesired(samples);
```

Finally, the trainer is connected to the output layer of the neural network. This will cause the trainer to receive the output of the neural network.

```
// Connects the Teacher to the last layer of the net
output.addOutputSynapse(trainer);
```

We are now ready to begin the background threads for all of the layers, as well as the trainers.

```
input.start();
hidden.start();
output.start();
trainer.start();
```

Finally, we set some parameters for the training. We specify that there are four rows in the input file, that we would like to train for 20,000 cycles, and that we are learning. If you set the learning parameter to false, the neural network would simply process the input and not learn. We will cover input processing in the next section.

```
monitor.setPatterns(4);
monitor.setTotCicles(20000);
monitor.setLearning(true);
```

We are now ready to begin the training process. Calling the Go method of the monitor will start the training process in the background.

```
monitor.Go();
```

The neural network will now be trained for 20,000 cycles. When the neural network is finished training, the error level should now be at a reasonably low level. An error level below 10% is acceptable.

Running the Neural Network

Now that the neural network has been trained, we can test it by presenting the input patterns to the neural network and observing the results. The method used to run the neural network must first prepare the neural network to process data. Currently, the neural network is in a training mode. To begin with, we will remove the trainer from the output layer. We will replace the trainer with an FileOutputSynapse so that we can record the output from the neural network. The following lines of code do this.

```
output.removeOutputSynapse(trainer);
FileOutputSynapse results = new FileOutputSynapse();
results.setFileName(resultFile.getText());
```

Now we must reset the input stream. We will use the same file input stream that we used during training. This will feed the same inputs that were used during training to the neural network.

```
inputStream.resetInput();
samples.resetInput();
results.setMonitor(monitor);
output.addOutputSynapse(results);
```

Next, we must restart all of the threads that correspond to the neural network.

```
input.start();
hidden.start();
output.start();
trainer.start();
```

Now that the threads have been restarted, we must set some basic configuration information for the recognition. The following lines of code do this.

```
monitor.setPatterns(4);
monitor.setTotCicles(1);
monitor.setLearning(false);
```

First, the number of input patterns is set to four. This is because we want the neural network to process each of the four input patterns that you originally used to train the neural network. Finally, the mode is set to learning. With this completed, we can call the "Go" method of the monitor.

```
monitor.Go();
```

When training completes, you will see that the output file produces will be similar to Listing 2.

Listing 2: The output from the neural network

```
0.012549763955262739
0.9854631848890223
0.9853159647305264
0.01783622084836082
```

You can see that the first line from the listing is a number which is reasonably close to zero. This is good because the first line of the input training file, as seen in Listing 1, was supposed to result in zero. Similarly, the second line is reasonably close to one, which is also good because the second line in the training file was also supposed to produce one.

Conclusion

As you can see, the JOONE engine encapsulates much of the complexities of neural network programming. The example presented here shows the basic process by which a neural network can be used. Though real-world implementations of neural networks will be much more complex, the basic process remains the same. Data is presented to the neural network for training, and then new patterns are presented for recognition. The example program provides a good starting point for exploration with JOONE. The complete example source file is shown in Listing 3.

Listing 3: The complete example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

import org.joone.engine.*;
import org.joone.engine.learning.*;
import org.joone.io.*;

/**
 * Example: The XOR Problem with JOONE
 *
 * @author Jeff Heaton
 * @version 1.0
 */
public class XorExample extends JFrame implements
ActionListener,NeuralNetListener {

  FullSynapse t1,t2;
  JButton btnTrain;
  JButton btnRun;
  JButton btnQuit;
```

```
      JTextField inputFile;
      JTextField resultFile;
      JLabel status;

/**
 * Constructor. Set up the components.
 */
public XorExample()
{
  setTitle("XOR Solution");

  Container content = getContentPane();

  GridBagLayout gridbag = new GridBagLayout();
  GridBagConstraints c = new GridBagConstraints();
  content.setLayout(gridbag);

  c.fill = GridBagConstraints.NONE;
  c.weightx = 1.0;

  // Training input label
  c.gridwidth = GridBagConstraints.REMAINDER; //end row
  c.anchor = GridBagConstraints.NORTHWEST;
  content.add(
            new JLabel(
                      "Enter the name of the training input
                       file:"),c);

  // Training input filename
  c.gridwidth = GridBagConstraints.REMAINDER; //end row
  c.anchor = GridBagConstraints.NORTHWEST;
  content.add(
            inputFile = new JTextField(40),c);
  inputFile.setText("./train.txt");

  // Training input label
  c.gridwidth = GridBagConstraints.REMAINDER; //end row
  c.anchor = GridBagConstraints.NORTHWEST;
  content.add(
            new JLabel("Enter the name of the result file:")
                       ,c);

  // Training input filename
  c.gridwidth = GridBagConstraints.REMAINDER; //end row
  c.anchor = GridBagConstraints.NORTHWEST;
  content.add(
            resultFile = new JTextField(40),c);
  resultFile.setText("./result.txt");

  // the button panel
  JPanel buttonPanel = new JPanel(new FlowLayout());
  buttonPanel.add(btnTrain = new JButton("Train"));
  buttonPanel.add(btnRun = new JButton("Run"));
  buttonPanel.add(btnQuit = new JButton("Quit"));
```

```
  btnTrain.addActionListener(this);
  btnRun.addActionListener(this);
  btnQuit.addActionListener(this);

  // Add the button panel
  c.gridwidth = GridBagConstraints.REMAINDER; //end row
  c.anchor = GridBagConstraints.CENTER;
  content.add(buttonPanel,c);

  // Training input label
  c.gridwidth = GridBagConstraints.REMAINDER; //end row
  c.anchor = GridBagConstraints.NORTHWEST;
  content.add(
            status = new JLabel("Click train to begin
                                training..."),c);

  // adjust size and position
  pack();
  Toolkit toolkit = Toolkit.getDefaultToolkit();
  Dimension d = toolkit.getScreenSize();
  setLocation(
            (int)(d.width-this.getSize().getWidth())/2,
            (int)(d.height-this.getSize().getHeight())/2 );
  setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
  setResizable(false);
}

/**
 * The main function, just display the JFrame.
 *
 * @param args No arguments are used.
 */
public static void main(String args[])
{
  (new XorExample()).show(true);
}

/**
 * Called when the user clicks one of the three
 * buttons.
 *
 * @param e The event.
 */
public void actionPerformed(ActionEvent e)
{
  if ( e.getSource()==btnQuit )
    System.exit(0);
  else if ( e.getSource()==btnTrain )
    train();
  else if ( e.getSource()==btnRun )
    run();
}
```

```
 /*
 * Called when the user clicks the run button.
 */
protected void run()
{
  output.removeOutputSynapse(trainer);

  inputStream.resetInput();
  samples.resetInput();
  FileOutputSynapse results = new FileOutputSynapse();
  results.setFileName(resultFile.getText());
  results.setMonitor(monitor);
  output.addOutputSynapse(results);

  input.start();
  hidden.start();
  output.start();
  trainer.start();

  // number of rows (patterns) contained in the input file
  monitor.setPatterns(4);
  // How many times the net must be trained on the input
  // patterns
  monitor.setTotCicles(1);
  // The net must be trained
  monitor.setLearning(false);
  // The net starts the training job
  monitor.Go();
  status.setText("Results written to " + resultFile.getText());
}

/**
 * The input layer of neurons.
 */
SigmoidLayer input;

/**
 * The hidden layer of neurons.
 */
SigmoidLayer hidden;

/**
 * The output layer of neurons.
 */
SigmoidLayer output;

/**
 * The monitor. Used to pass parameters to all of the
 * JOONE objects.
 */
Monitor monitor;

/**
 * The file input stream.
```

```
 */
FileInputSynapse inputStream;

/**
 * Used to train the neural network.
 */
TeachingSynapse trainer;

/**
 * The training data.
 */
FileInputSynapse samples;


/**
 * Called when the user clicks the train button.
 */
protected void train()
{
  // First, creates the three Layers
  input = new SigmoidLayer();
  hidden = new SigmoidLayer();
  output = new SigmoidLayer();
  input.setLayerName("input");
  hidden.setLayerName("hidden");
  output.setLayerName("output");

  // sets their dimensions
  input.setRows(2);
  hidden.setRows(3);
  output.setRows(1);

  // Now create the two Synapses
  // input -> hidden conn.
  FullSynapse synapse_IH = new FullSynapse();
  // hidden -> output conn.
  FullSynapse synapse_HO = new FullSynapse();

  synapse_IH.setName("IH");
  synapse_HO.setName("HO");
  t1=synapse_IH;
  t2=synapse_HO;


  // Connect the input layer with the hidden layer
  input.addOutputSynapse(synapse_IH);
  hidden.addInputSynapse(synapse_IH);

  // Connect the hidden layer with the output layer
  hidden.addOutputSynapse(synapse_HO);
  output.addInputSynapse(synapse_HO);

  // Create the Monitor object and set the learning parameters
  monitor = new Monitor();
```

```
monitor.setLearningRate(0.8);
monitor.setMomentum(0.3);

// Pass the Monitor to all components
input.setMonitor(monitor);
hidden.setMonitor(monitor);
output.setMonitor(monitor);

// The application registers itself as monitor's listener
// so it can receive the notifications of termination from
// the net.

monitor.addNeuralNetListener(this);

inputStream = new FileInputSynapse();

// The first two columns contain the input values
inputStream.setFirstCol(1);
inputStream.setLastCol(2);

// This is the file that contains the input data
inputStream.setFileName(inputFile.getText());
input.addInputSynapse(inputStream);


trainer = new TeachingSynapse();
trainer.setMonitor(monitor);

// Setting of the file containing the desired responses,
// provided by a FileInputSynapse
samples = new FileInputSynapse();
samples.setFileName(inputFile.getText());

// The output values are on the third column of the file
samples.setFirstCol(3);
samples.setLastCol(3);
trainer.setDesired(samples);

// Connects the Teacher to the last layer of the net
output.addOutputSynapse(trainer);

// All the layers must be activated invoking their method
// start; the layers are implemented as Runnable objects, then
// they are allocated on separated threads. The threads will
// stop after training and will need to be restarted later.
input.start();
hidden.start();
output.start();
trainer.start();

// number of rows (patterns) contained in the input file
monitor.setPatterns(4);
// How many times the net must be trained on the input
// patterns
```

```
    monitor.setTotCicles(20000);
    // The net must be trained
    monitor.setLearning(true);
    // The net starts the training job
    monitor.Go();
  }

  /**
   * JOONE Callback: called when the neural network
   * stops. Not used.
   *
   * @param e The JOONE event
   */
  public void netStopped(NeuralNetEvent e) {
  }

  /**
   * JOONE Callback: called to update the progress
   * of the neural network. Used to update the
   * status line.
   *
   * @param e The JOONE event
   */
  public void cicleTerminated(NeuralNetEvent e) {
    Monitor mon = (Monitor)e.getSource();
    long c = mon.getCurrentCicle();
    long cl = c / 1000;
    // print the results every 1000 cycles
    if ( (cl * 1000) == c )
      status.setText(c + " cycles remaining - Error = "
                         + mon.getGlobalError());
  }

  /**
   * JOONE Callback: Called when the network
   * is starting up. Not used.
   *
   * @param e The JOONE event.
   */
  public void netStarted(NeuralNetEvent e) {
  }

}
```

About the Author

Jeff Heaton is the author of *Programming Spiders, Bots, and Aggregators in Java* (Sybex, 2002). Jeff works as a software designer for the Reinsurance Group of America.

GLOSARIO

Autoasociación: Relación formada por los mismos componentes de una red para el de reconocimiento de un patrón.

Capa: Conjunto, sección o colección de neuronas.

Contorno: Conjunto de líneas que limitan un cuerpo o una figura. Forma que presenta un objeto o cuerpo sobre el fondo en que se destaca.

Difuminar: Atenuar un color al extenderlo, disminuyendo su intensidad para hacerlo vaporoso e indeciso.

Discriminación: Diferenciación de un objeto o de un patrón determinado.

Distorsión: Deformación de la onda de imágenes, sonidos o señales durante su propagación.

Framework: es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un framework puede incluir soporte de programas, librerías y un lenguaje de programación entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Inductividad: Inducción a la activación de otras neuronas.

Invariante: No cambia por transformaciones en el ambiente o escenario.

Intercapa: Procesos realizados fuera de una capa.

Intracapa: Procesos realizados dentro de una capa.

Máscara: En el tratamiento de imágenes, una máscara es el marco de selección de una zona de la imagen, así se puede tratar la imagen en el interior o en el exterior de la máscara, sin afectar al resto de la imagen.

Multicapa: Existencia de dos o mas capas en la red.

Normalización: Ajustar a un tipo, modelo o norma.

Paralelo: Proceso desarrollado a un mismo tiempo.

Recurrente: Proceso que reaparece o se realiza con cierta frecuencia o de manera iterativa.

Sinapsis: Es el punto de contacto entre un una neurona y otra.

Umbral: En tratamiento de imágenes, aplicar un umbral es marcar un punto a partir del cual el efecto sobre la imagen se produce y, por debajo del cual, no. Lo más usual es que ese efecto sea la mera visibilidad.

Unicapa: Termino que en las redes neuronales significa la existencia de una sola capa.

BIBLIOGRAFÍA

[01]. González, Rafael C. y Woods, Richard E. (1996). Tratamiento digital de imágenes. Madrid: Addison-Wesley.

[02]. González, Rafael C. y Woods, Richard E. (2002). Digital Image Processing. 2nd Edition. Prentice Hall.

[03]. Pajares, Gonzalo y De la Cruz, Jesús M. (2001). Visión por Computador. Editorial RA-MA.

[04]. Buhmman, Joachim M.; MALIK, Jitendra; PERONA, Pietro. Image recognition: Visual grouping, recognition, and learning. PNAS December 7, 1999 vol. 96 _ no.25 _ 14203 –14204.

[05]. James W. Davis Stephanie R. Taylor. Analysis and Recognition of Walking Movements. Dept. of Computer and Information Science. Ohio State University. Columbus, OH 43210 USA. {jwdavis,taylors}@cis.ohio-state.edu. Paper.

[06]. Lutz Goldmann, Mustafa Karaman and Thomas Sikora. Human Body Posture Recognition Using MPEG-7 Descriptors. Technical University Berlin, Communications System Group, Einsteinufer 17, Berlin, 10587 Germany. Paper

[07]. Miguel Ángel Cazorla Quevedo. Un enfoque Bayesiano para la extracción de características y agrupamiento en visión artificial. Dirigida por: Dr. Francisco Escolano Ruiz. Departamento de Ciencia de la Computación e Inteligencia Artificial. Mayo del 2000. Universidad de Alicante. Tesis doctoral.

[08]. K. Toyama, J. Krumm, B. Brumitt, and B. Meyers. Wallflower: Principles and practice of Background maintenance. In Proc. International Conference on Computer Vision, pages 255–261, 1999.

[09]. A System for Video Surveillance and Monitoring. Robert T. Collins, Alan J. Lipton, Takeo Kanade, Hironobu Fujiyoshi, David Duggins, Yanghai Tsin, David Tolliver, Nobuyoshi Enomoto, Osamu Hasegawa, Peter Burt and Lambert Wixson. The Robotics Institute, Carnegie Mellon University, Pittsburgh PA. 2000

[10]. Detección Jerárquica de Móviles sobre Geometrías de Fóvea Adaptativa. J. A. Rodríguez, C. Urdiales, P. Camacho, F. Sandoval. Dpto. Tecnología Electrónica – E.T.S. Ingenieros de Telecomunicación. Universidad de Málaga – Campus de Teatinos, 29071 - Málaga.

[11]. Handwritten Carácter recognition. Miguel Po-Hsien Wu. Tesis. The University Of Queenland.

[12]. Advances in Multimedia Information Processing - PCM 2004 Proceedings 2: 5th Pacific RIM Conferen... editado por Kiyoharu Aizawa, Yuichi Nakamura, Shin'ichi Satoh

[13]. Tutorial de Redes Neuronales. Maria Isabel Acosta. Harold Salazar. Camilo A. Zuluaga M. Universidad Tecnologica de Pereira. Facultad de Ingenieria Electrica.

Recursos Web

[14]. Digital Image Processing Place

http://www.imageprocessingplace.com/

[15]. Computer Vision Demos.

http://www.cs.cmu.edu/~cil/txtv-demos.html

[16]. Image Processing with Java 2D

http://www.javaworld.com/javaworld/jw-09-1998/jw-09-media.html

[17]. An Introduction To Digital Image Processing

http://www.gamedev.net/

[18]. Video Surveillance And Monitoring

http://www-2.cs.cmu.edu/%7Evsam/vsamhome.html

[19]. Optical Flow (Flujo óptico)

http://www.elai.upm.es/spain/Investiga/GCII/personal/rvaras/opticalflow.htm

[20]. Introducción a las Redes Neuronales Artificiales

http://www.iiia.csic.es