UNIVERSIDAD NACIONAL DE CÓRDOBA
Facultad de Matemática, Astronomía y Física

Aplicación de redes neuronales en la clasificación de imágenes

Trabajo Especial de Licenciatura en Ciencias de la Computación

Florencia Mihaich

Director: Dr. Oscar H. Bustos

Córdoba, 21 de julio de 2014

Resumen

La eficiencia de la combinación entre los ojos y cerebro humano en resolver problemas de reconocimiento de patrones permiten a los científicos considerar la posibilidad de aplicar, en los algoritmos de clasificación, sistemas computacionales basados en modelos simples del cerebro humano.

La ingeniería del software provee un enfoque sistemático y disciplinado que permite crear esos sistemas de forma robusta y fiable. Se garantizan estas características a través del seguimiento de estándares definidos.

En este proyecto se pretende exponer un marco teórico sobre la categorización de imágenes digitales, y sobre la estructura y funcionamiento de la redes neuronales *perceptrón multicapas* y *SOM* (mapas de auto organización de Kohonen).

A su vez, en base a estos conceptos, se desea desarrollar un sistema de software de clasificación de imágenes que permita explorar algoritmos de categorización que utilicen redes neuronales, para comparar su efectividad y eficiencia respecto a métodos estándares de clasificación estática.

Palabras claves:

Imágenes digitales, clasificación, red neuronal, Perceptrón, K-means, SOM, ingeniería del software, ESA, requerimientos, arquitectura de software, diseño.

Clasificación:

F.1.1 Models of Computation (Theory of Computation, Computation by abstract devices).
I.2.6 Learning (Computing Methodologies, Artificial intelligence).
I.5.1 Models (Computing Methodologies, Pattern Recognition).
I.5.3 Clustering (Computing Methodologies, Pattern Recognition).

Agradecimientos

A mis padres, Jorge y María, por ser las personas en quienes siempre voy a encontrar un cariño sincero y apoyo incondicional para todo lo que decida emprender.

A mi novio, Christian, quien me hizo dar cuenta de la importancia de terminar este camino, y me acompañó continuamente compartiendo conmigo el entusiasmo por el software.

A mi director de tesis, Oscar Bustos, por su inmensa paciencia, su comprensión, su predisposición y su motivación constante a seguir adelante.

A quien me hizo elegir esta carrera, que realmente me apasiona, Javier Blanco.

Índice general

CAPÍTULO 1

Introducción

Las actividades de investigación desarrolladas en torno al estudio de *redes neuronales artificiales*, o simplemente redes neuronales, están motivadas en modelar la forma de procesar la información por sistemas nerviosos biológicos, especialmente, por el cerebro humano.

El funcionamiento del cerebro humano es completamente distinto al funcionamiento de un computador digital convencional. La actividad del cerebro se corresponde con un sistema altamente complejo, no-lineal y paralelo; ya que es capaz de realizar múltiples operaciones de manera simultánea.

Una red neuronal está construida por un conjunto de unidades sencillas de procesamiento llamadas neuronas. Se caracteriza por:

- Adquirir el conocimiento a través de la experiencia,
- Demostrar flexibilidad de adaptación frente a las variaciones del entorno,
- Exponer una inmensa plasticidad, evidente en su capacidad para responder correctamente frente a un estímulo nunca antes recibido,
- Poseer un alto nivel de tolerancia a fallas, y
- Lograr una elevada tasa de computabilidad basada a su paralelismo masivo.

Debido a las propiedades antes mencionadas, las neuroredes se han convertido en una herramienta de gran contribución para obtener soluciones de aquellos problemas en los que no se conoce a priori el algoritmo a utilizar.

Áreas como el reconocimiento de patrones plantean situaciones con estas características. En particular, la clasificación de imágenes digitales basada en procedimientos que incorporen redes neuronales artificiales es el objetivo de estudio del presente trabajo.

Actualmente se conocen numerosos métodos de categorización de imágenes con un excelente rendimiento computacional, pero éstos se encuentran sujetos a precondiciones respecto a los datos de entrada. Contrariamente, las redes neuronales son descriptas como no paramétricas, es decir, no requieren asumir una distribución estática de la información ingresada. Durante la fase de entrenamiento, la red "aprende" las regularidades presentes en los datos incorporados y construye reglas que se pueden extender a datos desconocidos.

Este trabajo pretende comprar la efectividad y eficiencia de métodos basados en redes neuronales artificiales respecto a procedimientos ampliamente usados para categorizar imágenes. Arribar a conclusiones será posible mediante la implementación de un software de la clasificación de imágenes **ANNIC** (*Artificial Neural Network Image Classification*) que permita la categorización de imágenes usando, en la fase de entrenamiento, una red neuronal artificial *perceptrón multicapas*, un mapa de auto organización de Kohonen (red *SOM*) o el tradicional

algoritmos *K-means*. El sistema tendrá disponible la posibilidad de verificar la calidad de la clasificación y de ejecutar pruebas de estrés sobre los distintos métodos.

La producción de la aplicación se realizará de acuerdo a estándares de ingeniería de software definidos para pequeños proyectos y aplicando conceptos y tecnologías apropiadas durante el desarrollo.

1.1. Estructura del trabajo

Los temas a desarrollarse en los próximos capítulos se pueden resumir de la siguiente manera:

- **Capítulo 2**: Expone una descripción general acerca de conceptos relacionados con imágenes digitales y su clasificación.
- **Capítulo 3**: Provee un marco teórico sobre las redes neuronales artificiales especificando tanto sus elementos y características principales, como su modo de operación "aprendizaje y recuerdo".
- **Capítulo 4**: Explica detalladamente la red neuronal artificial perceptrón, evidenciando sus propiedades principales y puntualizando el algoritmo de aprendizaje utilizado (*"backpropagation"*).
- **Capítulo 5**: Describe los mapas de auto organización de Kohonen (redes neuronales *SOM*) y el método de competición empleado durante el proceso de aprendizaje.
- **Capítulo 6**: Introduce conceptos generales de la ingeniería de software, incluyendo el ciclo de vida del software y sus distintos modelos.
- **Capítulo 7**: Exhibe el sistema de categorización ANNIC. Determina el estándar de software empleado durante su desarrollo, las tecnologías usadas en el diseño y en la implementación y el paradigma de programación aplicado.
- **Capítulo 8**: Presenta los resultados y conclusiones acerca de la utilización de los diferentes métodos de clasificación de imágenes basados en redes neuronales. También se plantean posibles trabajos futuros.

CAPÍTULO 2

Imágenes digitales y clasificación

2.1. Imágenes digitales

Una imagen natural capturada con una cámara, un telescopio, un microscopio o cualquier otro tipo de instrumento óptico presenta una variación de sombras y tonos continua. Imágenes con estas características se denominan *imágenes analógicas*.

Para que una imagen analógica en blanco y negro, escala de grises o a color, pueda ser 'manipulada' usando un ordenador, primero debe convertirse a un formato adecuado. Este formato es la *imagen digital* correspondiente.

2.1.1. Representación

Una imagen se representa por una función en dos dimensiones $f(x, y)$, cuyo valor corresponde a la intensidad de luz en cada punto del espacio de las coordenadas (x, y). En el caso de una imagen monocromática, al valor de $f(x, y)$ se le denominará *nivel o escala de gris* en el punto de coordenadas (x, y). Las imágenes a color están formadas por la combinación de imágenes 2-D.

En base a este concepto, una imagen es analógica si el dominio (valores de (x, y)) y el rango (valores de $f(x, y)$) son continuos; mientras que una imagen es digital si el dominio y el rango son discretos.

Para convertir una imagen de tonos continuos en formato digital, la imagen analógica es dividida en valores de brillos individuales a través de dos procesos denominados *muestreo (sampling)* y *cuantización (quantization)*.

La conversión de las coordenadas a un dominio discreto está asociada al concepto de muestreo y la conversión de la amplitud a un rango discreto está asociada al concepto de cuantización (niveles de grises).

Desde el punto de vista práctico, una imagen puede considerarse como un conjunto de celdas que se organizan en las posiciones correspondientes a una matriz bidimensional $M \times N$.

Asumiendo que $f(x, y)$ es muestreada a una imagen que tiene M filas y N columnas, se dice que la imagen tiene tamaño $M \times N$. El origen de la imagen se define en $(x, y) = (0, 0)$. La siguiente coordenada a lo largo de la primera fila es $(x, y) = (0, 1)$. Es decir, que de acuerdo con la notación de matrices, el eje vertical (y), recorre la imagen de arriba hacia abajo, mientras que eje horizontal (x) la recorre de izquierda a derecha.

De esta forma se puede representar una imagen digital como la siguiente matriz $M \times N$:

$$f(x,y) = \begin{pmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{pmatrix}$$

El lado derecho de la igualdad es por definición una imagen digital. Cada elemento de esta matriz se denomina *píxel* (*picture element*) y representa el menor componente no divisible de la imagen. Al valor numérico de cada píxel se lo conoce como *Nivel Digital* (*ND*).

En el proceso de digitalización se deben tomar decisiones sobre los valores de M, N y el número de niveles de grises L permitido para cada píxel. No hay restricciones sobre M y N, sólo deben ser enteros positivos. Sin embargo, debido al tipo de procesos, almacenamiento y hardware de muestreo, el número de niveles de grises si tiene restricciones: es en general un entero potencia de 2 ($L = 2^k$, para algún $k \in \mathbb{N}$). Se asume también que estos niveles son enteros equidistantes en el intervalo $[0, L-1]$.

Resumiendo, el muestreo es la conversión que sufren las dos dimensiones de la señal analógica generando la noción de píxeles. La cuantización es la conversión que sufre la amplitud de la señal análoga en niveles de grises. Los niveles de grises corresponden al valor que toman los elementos matriciales. Si se tienen 256 niveles de grises (de 0 a 255), el 0 representa que el píxel está en su mínima intensidad (negro) y el 255 que el píxel está en su máxima intensidad (blanco).

2.1.2. Resolución espacial y profundidad del color

Las dos principales causas de pérdida de información cuando se captura una imagen digital son la naturaleza discreta de los píxeles y el rango limitado de los valores de intensidad luminosa que puede tener cada uno de estos elementos.

En base a estas dos razones, surgen los conceptos de *resolución espacial* y *profundidad del color*.

Resolución espacial

El muestreo determina la resolución espacial de una imagen. La resolución espacial define el menor detalle discernible de ésta, es decir, el menor número de pares comprendidos en una unidad de distancia (por ejemplo, 100 pares por milímetro).

Cada píxel no representa sólo un punto en la imagen, sino una región rectangular. Por lo tanto, con píxeles grandes no sólo la resolución espacial es baja, sino que el valor del nivel de gris correspondiente hace aparecer discontinuidades en los bordes de los píxeles. A medida que los píxeles se hacen más pequeños, el efecto se hace menos pronunciado, hasta el punto en que se tiene la sensación de una imagen continua. Esto sucede cuando el tamaño de los píxeles es menor que la resolución espacial de nuestro sistema visual.

(a) 32x32 píxeles3 (b) 64x64 píxeles (c) 128x128 píxeles

Figura 2.1: Diferencias al variar la resolución espacial.

Profundidad del color

El efecto de la cuantización viene dado por la imposibilidad de tener un rango infinito de valores para la intensidad o brillo de los píxeles. Después de que la imagen de un objeto ha sido capturada, a cada píxel se le asigna una intensidad que será un número entero. La apreciación de este valor es

directamente proporcional al número de bits que utiliza el dispositivo con que se captura la imagen para representar los enteros.

La *profundidad de color* se refiere al número de bits necesarios para codificar y guardar la información de color de cada píxel en una imagen. Un bit es una posición de memoria que puede tener el valor 0 ó 1. Cuanto mayor sea la profundidad de color en bits, la imagen dispondrá de una paleta de colores más amplia.

Si se utiliza un bit, la imagen será en blanco/negro, sin grises (0=color negro, 1= color blanco); mientras que si se utilizan 8 bits la imagen tendrá 256 niveles de grises.

(a) 1 bit (b) 3 bits (c) 5 bits (d) 6 bits (e) 8 bits

Figura 2.2: Diferencias al variar la profundidad del color.

2.1.3. Modelos de color

Un *modelo de color* es un modelo matemático abstracto que describe la forma en la que los colores pueden representarse como tuplas de números. El objetivo de un modelo de color es facilitar la especificación de los colores de una forma normalizada y aceptada genéricamente.

A continuación se describirán algunos de los modelos de color utilizados con más frecuencia en el procesamiento de imágenes digitales.

Modo monocromático

El modo monocromático se corresponde con una profundidad de color de un bit. Son imágenes formadas por píxeles blancos o píxeles negros puros, sin tonos intermedios entre ellos.

Modo escala de grises

Las imágenes en modo escala de grises manejan un sólo canal: el negro. Este canal podrá tener una gama de 256 tonos de grises.

El tono de gris de cada píxel se puede obtener asignándole un valor de brillo entre 0 (negro) y 255 (blanco). Este valor también se puede expresar como porcentaje de negro, donde 0 % es igual a blanco y 100 % es igual a negro.

Modo color indexado

En este modo, la gama de colores de la imagen se adapta a una paleta con un máximo de 256 colores (2^8). Su principal inconveniente es que la mayoría de las imágenes del mundo real se componen con una cantidad mayor de tonos.

Modo RGB

En el modelo RGB cada color de la imagen se forma por la combinación de tres canales correspondientes con los colores primarios: rojo (**R**ed), verde (**G**reen) y azul (**B**lue).

Es un modelo de color basado en la síntesis aditiva: un color se representa mediante la suma de los colores primarios, siendo el blanco la suma de todos ellos (Figura 2.3).

Este modelo no define por sí mismo lo que significa exactamente rojo, verde y azul; por lo que los mismos valores RGB pueden mostrar tonos notablemente diferentes en distintos dispositivos.

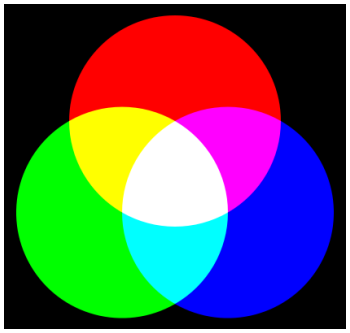


Figura 2.3: Modelo aditivo de colores rojo, verde, azul [4].

Para indicar con qué proporción se mezcla cada color, se asigna un valor a cada uno de los colores primarios. El valor 0 significa que ese color primario no interviene en la mezcla y mientras más aumenta ese valor, se entiende que dicho color primario aporta más intensidad.

Asumiendo 8 bits de profundidad, cada color primario puede tener un valor máximo de 255. En base a esta precondición, el rojo se representa con la tupla $(255, 0, 0)$, el verde con $(0, 255, 0)$, el azul con $(0, 0, 255)$, el blanco con $(255, 255, 255)$ y el negro con $(0, 0, 0)$.

La combinación de dos de los colores primarios a nivel 255 con el tercero en nivel 0 da lugar a tres colores intermedios: amarillo $(255, 255, 0)$, cian $(0, 255, 255)$ y magenta $(255, 0, 255)$.

El conjunto de todos los colores se puede representar en forma de cubo. Cada color es un punto de la superficie o del interior de éste. La escala de grises estaría situada en la diagonal que une al color blanco con el negro (Figura 2.4).

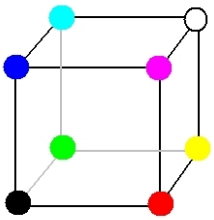


Figura 2.4: Cubo RGB [4].

Modo CMY

En el modelo CMY el espacio de color es el inverso exacto del modelo RGB: en este caso el origen es el blanco y los ejes primarios son los colores cyan (**C**yan), magenta (**M**agenta) y amarillo (**Y**ellow).

A continuación se detallan las ecuaciones que permiten pasar de un sistema a otro:

$$c = max - r \quad m = max - g \quad y = max - b$$
$$r = max - c \quad g = max - m \quad b = max - y$$

donde:

- max es el valor máximo de la intensidad.

Si se muestra una imagen en CMY como si fuera RGB se podrá observar una imagen con todos sus colores invertidos o negativos.

El modelo CMY es un modelo sustractivo: la suma de todos los colores produce el negro (Figura 2.5).

Se usa principalmente en la industria de la impresión debido a que las imágenes empiezan sobre papel blanco y la tinta se aplica para obtener los colores. Se han desarrollado técnicas para obtener

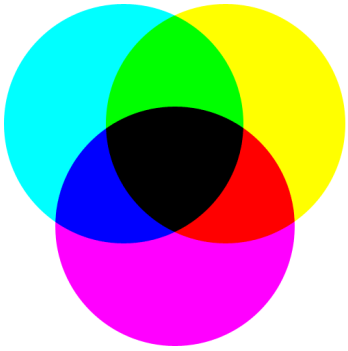


Figura 2.5: Modelo sustractivo cian, magenta y amarillo [5].

imágenes de mayor calidad a un menor costo. Una de ellas modifica el modelo CMY en *CMYK*, que agrega el color negro (bla**K**) para lograr su óptima representación.

Modo HSI

El modelo HSI se basa en la percepción humana del color y describe sus características fundamentales (Figura 2.6):

- Tono (***H****ue*): Es el color reflejado o transmitido a través de un objeto. Se mide como la posición en la rueda de colores estándar y se expresa en grados entre 0° y 360°. Normalmente, el tono se indica por el nombre del color (rojo, naranja o verde).
- Saturación (***S****aturation*): También denominada cromatismo. Es la 'fuerza' o pureza del color. La saturación representa la cantidad de gris que existe en proporción al tono y se mide como porcentaje comprendido entre 0 % (gris) y 100 % (saturación completa). En la rueda de colores estándar, la saturación aumenta a medida que nos aproximamos al borde de la misma y disminuye a medida que nos acercamos al centro.
- Brillo (***I****ntensity*): Es la luminosidad u oscuridad relativa del color y se suele medir como un porcentaje comprendido entre 0 % (negro) y 100 % (blanco).

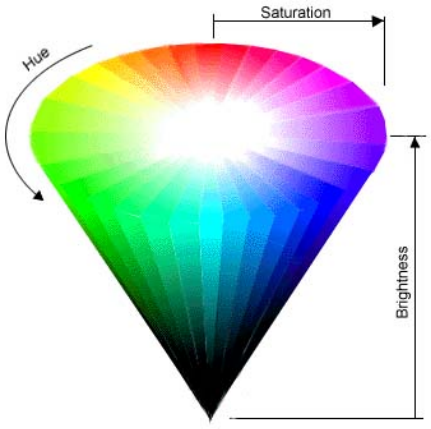


Figura 2.6: Cono de colores del espacio HSI [7].

2.2. Clasificación de imágenes

Una de las tareas más importantes en el procesamiento y análisis de imágenes es clasificar cada píxel como perteneciente a una cierta categoría o tema.

Como fruto de la clasificación digital se obtiene una cartografía e inventario de las categorías que son objeto de estudio. Por ejemplo, se puede obtener el número de píxeles, y por lo tanto la superficie, asignada a cada categoría. La imagen multibanda se convierte en otra imagen, del mismo tamaño y características que la original, con la importante diferencia que el ND que define cada píxel no tiene relación con la radiancia detectada por el sensor, sino que se trata de una etiqueta que identifica la categoría asignada a ese píxel.

La clasificación de imagen se beneficia notablemente con algunos procesos de corrección. Sin embargo, conviene considerar que puede abordarse una clasificación exclusivamente a partir de los ND de la imagen original, ya que las categorías temáticas suelen definirse de modo relativo a las condiciones específicas de la escena a clasificar. Con este planteamiento, no resulta preciso conocer detalladamente las condiciones de adquisición, basta con identificar en la imagen las clases a discriminar, sin pretender que esa identificación sea extrapolable a otras situaciones.

Al referirse a clases digitales, es preciso distinguir entre clases de información y clases espectrales. Las primeras son aquellas categorías de interés que la persona está tratando de identificar en la imagen. Las segundas son grupos de píxeles uniformes en valores de brillo en las diferentes bandas. El objetivo final en la clasificación es crear una correspondencia entre las clases espectrales y las clases de información que son de interés.

En muy pocas ocasiones existe una correspondencia uno a uno entre estos dos distintos tipos de clases. Por ejemplo, puede haber clases espectrales que no correspondan a ninguna clase temática de interés. Inversamente, clases temáticas amplias podrían tener subclases espectrales separables. Por ello, el trabajo final del analista de la imagen es decidir sobre la utilidad de las diferentes clases espectrales con respecto a las clases temáticas de interés.

En la clasificación digital de imágenes pueden distinguirse las siguientes fases:

1. Fase de entrenamiento: Definición digital de las categorías,
2. Fase de asignación o clasificación: Agrupación de los píxeles de la imagen en una de esas clases, y
3. Obtención y verificación de resultados.

2.2.1. Fase de entrenamiento

La clasificación digital de imágenes se inicia definiendo las categorías que se pretenden identificar.

En la captura de una imagen, diversos factores introducen cierta dispersión en torno al comportamiento espectral medio de cada cubierta. En términos de clasificación digital, esto supone que existe una determinada variación en torno al ND medio de cada categoría. Por lo tanto, las distintas clases no se pueden definir por un solo ND, sino que debe hacerse considerando un conjunto de ellos.

En base a la afirmación anterior, en la fase de entrenamiento es necesario seleccionar una muestra de píxeles de la imagen que representen adecuadamente a las categorías de interés. A partir de esos píxeles es posible calcular los NDs medios y la variabilidad numérica de cada categoría en todas las bandas que intervienen en la clasificación.

Al igual que en cualquier otro muestreo, el objetivo de esta fase es obtener los resultados más precisos con el mínimo coste.

Las estimaciones posteriores se basan sobre la muestra elegida, por lo que una incorrecta selección de ésta conducirá a resultados pobres en la clasificación posterior. Varios autores han comprobado que los resultados de la clasificación están mucho más influidos por la definición previa de las categorías, que por el criterio con el que éstas son posteriormente discriminadas. La fase de entrenamiento constituye el eje de la clasificación.

Tradicionalmente se han dividido los métodos de clasificación en dos grupos de acuerdo con la forma en que son obtenidas las estadísticas de entrenamiento:

- Método supervisado y
- Método no supervisado.

El método supervisado parte de un conocimiento previo, a partir del cual se seleccionan las muestras para cada una de las categorías. Por otra parte, el método no supervisado procede a una búsqueda automática de grupos de valores homogéneos dentro de la imagen. Queda al usuario, en este caso, encontrar la correspondencia entre esos grupos y sus categorías de interés.

El método supervisado pretende definir clases informacionales, mientras que el no supervisado tiende a identificar clases espectrales presentes en la imagen. Ninguno de los dos métodos proporciona una solución inmediata a todos los problemas presentes en una clasificación digital. Por un lado, el método supervisado puede catalogarse de subjetivo y artificial, ya que probablemente 'fuerza' al ordenador a discriminar categorías que no tengan un claro significado espectral. Por otro, el método no supervisado proporciona, en ocasiones, resultados de difícil interpretación y pocos conectados con las necesidades del usuario final del producto. Asimismo, resulta poco claro que este método sea capaz de identificar las agrupaciones naturales de la imagen.

Con el objetivo de paliar los inconvenientes de ambos métodos, han surgido diversas alternativas que los combinan de alguna forma. Así varios autores consideran una tercera manera de obtener las clases de entrenamiento:

- Método mixto.

En resumen, la elección del método a utilizar dependerá de los datos, medios disponibles y de las propias preferencias personales.

2.2.2. Fase de asignación o clasificación

En esta fase se trata de adscribir cada uno de los píxeles de la imagen a una de las clases previamente seleccionadas. Esta asignación se realiza en función de los NDs de cada píxel, para cada una de las bandas que intervienen en el proceso. Fruto de esta fase será una nueva imagen, cuyos NDs expresen la categoría temática a la que se ha adscrito cada uno de los elementos de la imagen original.

Desde el punto de vista estadístico, las técnicas de clasificación de imágenes definen un área de dominio en torno al centro de cada categoría a diferenciar mediante un conjunto de funciones discriminantes. Estas ecuaciones pueden considerarse como las fronteras que determinan cada categoría. Cada píxel será asignado a una clase i si su ND se encuentran dentro del área de dominio de dicha clase.

Criterios más comunes de clasificación

Los criterios más comunes para establecer las fronteras estadísticas entre clases son:

- Mínima distancia: Cada píxel se asigna a la clase más cercana.
- Mínima distancia a las medias (*K-means*): Cada píxel se asigna a la clase con la media más cercana.
- Paralelepípedos: Permite determinar al usuario umbrales de dispersión asociados a cada clase.
- Máxima verosimilitud: Cada píxel se asigna a aquella clase a la que posee mayor probabilidad de pertenencia. Este clasificador está basado en la suposición de que los valores correspondientes a cada categoría se esparcen según alguna distribución multivariada. Habitualmente se considera la distribución gaussiana.

Clasificación contextual

Las formas más simples de clasificación digital de imágenes consideran a cada píxel individualmente, asignándolo a una clase en base a sus valores medidos en cada una de las bandas espectrales sin importar como son clasificados los píxeles vecinos. Sin embargo, en cualquier imagen real, los píxeles adyacentes están relacionados o correlacionados.

En base a este concepto, los métodos de clasificación contextual asignan un píxel a cierta categoría teniendo en cuenta a que clases pertenecen los píxeles en la vecindad del mismo. Se obtiene así un mapa temático que es consistente tanto espectral como espacialmente.

Otra ventaja de estos métodos es una mejor confección de clases temáticas al posibilitar la corrección de errores provenientes de 'ruido' o mal desempeño de una data técnica de clasificación.

Un caso particular de este tipo de mapeo es:

- Método ICM: Clasificación contextual por modas condicionadas iteradas [2].

Otros criterios de asignación

Algunos otros métodos de clasificación de imágenes que no fueron mencionados con anterioridad son:

- Clasificador en árbol (*decision tree classifier*): Discrimina secuencialmente cada categoría de acuerdo a ciertos criterios seleccionados por el analista de la imagen. Puede considerarse como un caso particular de un sistema experto que se ha extendido actualmente dentro de las llamadas 'técnicas de inteligencia artificial'.

- Redes Neuronales (*artificial neural networks*): Es una de las nuevas técnicas empleada dentro de la clasificación de imágenes. En esencia las redes neuronales se utilizan para predecir un cierto comportamiento complejo, habitualmente a partir de una muestra de entradas y salidas observadas. Con los datos de esa muestra la red 'aprende' a reconocer el resultado a partir de los valores de entrada, clasificando el resto de las observaciones de acuerdo a esas reglas. Está técnica de mapeo será el centro del presente trabajo.

- Clasificación 'borrosa' (*fuzzy classification*): Esta técnica considera más de una categoría potencial para cada elemento de la imagen. Por lo tanto, cada píxel se etiqueta en varias categorías, con un valor más o menos elevado en función de su similitud espectral. Convencionalmente, la función de pertenencia corresponde a una distribución binaria: 0 (no pertenece) y 1 (pertenece). También puede aceptarse una función de pertenencia comprendida entre 0 y 1, lo que permitiría una asignación simultánea a varias categorías con diferentes grados.

Filtrado de ruido como etapa post-clasificación

Las imágenes obtenidas después de aplicar un proceso de clasificación presentan ruido (apariencia granular) debido a la variabilidad encontrada por la regla de clasificación. Esa falta de homogeneidad puede ser causada, por ejemplo, por el hecho de que algunos píxeles dentro de una cierta clase en una subárea de la imagen fueron clasificados como pertenecientes a otra clase. En tal situación es deseable 'suavizar' o 'filtrar' la imagen a fin de destacar solamente los aspectos dominantes de la clasificación.

Uno de los procesos de filtrado más usado en esta etapa es el llamado *filtro por mayoría*. En éste se desplaza a lo largo de toda la imagen una ventana cuadrada de cierto tamaño ($3x3$, $5x5$, entre otros). Dentro de ella al píxel central se lo reclasifica, en caso de ser necesario, como perteneciente a la clase representada por la mayoría (mitad más uno) de los píxeles en la ventana. Si no existe una tal clase mayoritaria, el píxel central no es alterado. En este proceso siempre son usados los valores no alterados a medida que se desplaza la ventana.

Este filtro suele modificar de manera tal que se preserven los bordes y/o se obtengan áreas de cada clase de mayor o igual tamaño que un cierta dimensión previamente fijado por el analista.

2.2.3. Obtención y verificación de resultados

Independientemente del método empleado en la clasificación digital, los resultados se almacenan en una nueva imagen, similar a la original en cuanto a estructura y tamaño, pero en la que el ND de cada píxel corresponde a la categoría a la que se asignó. Esta nueva imagen puede ser el producto final del trabajo o servir como estadio intermedio de un proyecto más amplio.

Toda clasificación conlleva un cierto margen de error en función de la calidad de los datos o de la rigurosidad del método empleado. Por ello, resulta conveniente aplicar algún procedimiento de verificación que permita medir ese error y, en base a éste, valorar la calidad final del trabajo y su aplicabilidad operativa.

Medidas de fiabilidad

La estimación de la exactitud alcanzada en la clasificación puede realizarse por diversos criterios, entre ellos:

- Comprobando el inventario de la clasificación con el obtenido por otras fuentes convencionales.
- Estudiando la fiabilidad obtenida al clasificar las áreas de entrenamiento.
- Seleccionando áreas de verificación para las cuales se conoce con exactitud la clase a la cual pertenece.

El método más sencillo para estimar la precisión obtenida por un mapa se basa en calcular las diferencias entre el inventario ofrecido por la clasificación y el brindado por otras fuentes que se consideren fiables (como por ejemplo, estadísticas oficiales o cartografía de detalle). Suponiendo al documento de referencia como plenamente fiable, esta medida sólo indica el porcentaje de error pero no su localización.

Otra opción para verificar los resultados consiste en clasificar los campos de entrenamiento para comprobar si se ajustan correctamente a las categorías que se pretenden definir. Ésta es una medida de fiabilidad sesgada ya que, dado que las áreas de entrenamiento sirven para definir estadísticamente a las distintas categorías, los píxeles incluidos en ellas tienen mayor probabilidad de clasificación certera que el resto de los píxeles de la imagen. Sin embargo, esta práctica resulta útil para determinar la precisión de los campos de entrenamiento: si los píxeles presentes en estas áreas se asignan a otras clases, conviene delimitar nuevos campos de entrenamiento.

La tercer vía de trabajo consiste en seleccionar, con posterioridad a la clasificación, una serie de áreas de test. Para éstas se realiza un muestreo del área de estudio a fin de obtener las medidas de campo necesarias para verificar los resultados de la categorización. A partir de la realización del muestreo, puede construirse una *tabla* o *matriz de confusión*, en donde se resuman los acuerdos y desacuerdos entre las clases del mapa y del área de estudio. Esta matriz puede analizarse estadísticamente con la finalidad de obtener una serie de medidas sobre la fiabilidad del inventario: global y para cada una de las categorías.

Diseño del muestreo para la verificación

El diseño del muestreo para la verificación supone la columna vertebral de este proceso. La principal virtud de un buen muestreo es seleccionar adecuadamente una parte del área de estudio de tal forma que, siendo lo más pequeña posible, sea suficientemente representativa del conjunto.

La calidad de la estimación depende de una serie de factores que deben considerarse al planificar el muestreo. Entre ellos se debe tener en cuenta el método de selección de la muestra, el tamaño y distribución de la misma, y el nivel de confianza otorgado a la estimación.

Tipos de muestreo

Los esquemas empleados con mayor frecuencia en el proceso de verificación son:

- Aleatorio simple: Los elementos a verificar se eligen de tal forma que todos cuenten con la misma probabilidad de ser seleccionados y que la elección de cada uno no influya en el siguiente.
- Aleatorio estratificado: La muestra se realiza dividiendo la población en regiones o estratos de acuerdo a alguna variable auxiliar.
- Sistemático: La muestra se distribuye a intervalos regulares a partir de un punto de origen seleccionado aleatoriamente.

- Sistemático no alineado: Modifica el modelo anterior al variar aleatoriamente una coordenada en cada fila y columna de la imagen clasificada, pero manteniendo fija la otra.
- Por conglomerados: Se selecciona como unidad de muestra un grupo de observaciones denominado conglomerado (del inglés, *cluster*). En base a este concepto, por cada punto a verificar (elegido aleatoriamente) se considera, también, un conjunto de sus vecinos de acuerdo a un esquema prefijado.

Tamaño de la muestra

En cuanto al tamaño de la muestra, Congalton (1988) sugiere una superficie aproximada al 1 % de cada superficie clasificada. Sin embargo, también es preciso considerar el nivel de confianza que quiera otorgarse a la estimación así como la propia variabilidad de la imagen considerada.

Como se trata de medir una variable binomial (acierto-error), se emplea normalmente la fórmula:

$$n = \frac{z^2 * p * (1 - p)}{E^2}$$

donde:

- z es la abcisa de la curva normal para un nivel determinado de probabilidad,
- p indica el porcentaje de aciertos, y
- E es el nivel de error permitido.

Es aconsejable además, realizar el muestreo para todas las clases por separado, partiendo de la clase con menor extensión. Esta marcará la proporción del área a muestrear para el resto de las categorías.

Una vez diseñado el método y tamaño de la muestra, y localizados los puntos de verificación, la fase siguiente consiste en obtener, para cada punto, la clase real y la obtenida por la clasificación.

2.2.4. Matriz de confusión

Consecuencia de la fase de muestreo será un listado de puntos de test, para los que se conoce tanto su cobertura real como la deducida por la clasificación. Con estos datos puede formarse una matriz denominada *matriz de confusión* ya que resume los conflictos que se presentan entre las categorías. En esta matriz, las filas se ocupan por las clases de referencia y las columnas por las categorías deducidas de la clasificación. Lógicamente, ambas tendrán el mismo número y significado. Por lo tanto, se trata de una matriz $n \times n$, donde n es el número de categorías.

La diagonal de la matriz expresa el número de puntos de verificación en donde se produce un acuerdo entre las dos fuentes (mapa y realidad), mientras que los marginales suponen errores de asignación. La relación entre el número de puntos correctamente asignados y el total expresa la fiabilidad global del mapa. Los residuales en las filas indican los tipos de cubierta real que no se incluyeron en el mapa, mientras que los residuales en las columnas implican cubiertas del mapa que no se ajustaron a la realidad. En definitiva, representan los errores de omisión y comisión respectivamente.

2.2.5. Análisis estadístico de la matriz de confusión

El interés de las tablas de confusión proviene de su capacidad para plasmar los conflictos entre categorías. Con su lectura, no sólo se conoce la fiabilidad global de la clasificación, sino también la exactitud conseguida para cada una de las clases y los principales conflictos entre ellas.

Medidas globales de fiabilidad

A partir de la matriz de confusión pueden desarrollarse toda una serie de medidas estadísticas que concluyan el proceso de validación. La más simple de calcular es la fiabilidad global del mapa, relacionando los elementos de la diagonal con el total de puntos muestreados:

$$\hat{F} = \frac{\sum_{i=1}^{n} X_{ii}}{\sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}}$$

Gracias a la teoría del muestreo pueden calcularse los umbrales inferior y superior en los que se encontraría la fiabilidad real alcanzada por la clasificación, a partir de conocer el valor estimado. Ese intervalo se obtiene, para un determinado nivel de significación α, a partir del error de muestreo (ES) y del nivel de probabilidad $1 - \alpha$:

$$F = \hat{F} \pm z * ES$$

donde:

- z indica la abcisa del área bajo la curva normal para el nivel de probabilidad $(1 - \alpha)$.
- ES es el error estándar de muestreo en función del porcentaje de aciertos (p), de fallos $(1-p)$ y del tamaño de la muestra (n):

$$ES = \sqrt{\frac{p(1-p)}{n}}$$

Errores de omisión y comisión

Resulta necesario tener en cuenta que la fiabilidad global del mapa puede ocultar importantes diferencias entre categorías. Por ello, un análisis más riguroso debe considerar también los elementos marginales de la matriz de confusión.

En el caso de las filas, los marginales indican el número de píxeles que, perteneciendo a una determinada categoría, no fueron incluidos en ella. Éstos se denominan errores de omisión (E_o). Para cada clase se calculan como:

$$E_{o,i} = \frac{X_{i+} - X_{ii}}{X_{i+}}$$

donde:

- X_{i+} indica el marginal de la fila i.
- X_{ii} es el elemento de la fila i de la diagonal de la matriz.

De igual manera, los valores no diagonales de las columnas expresan los errores de comisión, es decir, los píxeles que se incluyeron en una determinada categoría perteneciendo realmente a otra:

$$E_{c,j} = \frac{X_{+j} - X_{jj}}{X_{+j}}$$

donde:

- X_{+j} indica el marginal de la columna j.
- X_{jj} es el elemento de la columna j de la diagonal de la matriz.

Los errores de omisión y comisión expresan dos enfoques del mismo problema: los primeros se refieren a la definición imperfecta de la categoría mientras que los segundos, a una delimitación excesivamente amplia.

Análisis categórico multivariable: coeficiente Kappa

Hasta ahora se ha considerado únicamente lo que ocurre en la diagonal y en los residuales de las filas y columnas de la matriz de confusión. Sin embargo, también resulta de interés analizar las relaciones múltiples entre las distintas categorías.

Con este objetivo, uno de los índices más empleados es el coeficiente Kappa (κ), que mide la diferencia entre el acuerdo mapa-realidad observado y el que se esperaría simplemente por azar. La estimación de κ se obtiene a partir de la siguiente fórmula [8]:

$$\kappa = \frac{n * \sum_{i=1}^{n} X_{ii} - \sum_{i=1}^{n} X_{i+} X_{+i}}{n^2 - \sum_{i=1}^{n} X_{i+} X_{+i}}$$

donde:

- n es el número de categorías,
- X_{ii} indica el acuerdo observado, y
- El producto de los marginales ($X_{i+}X_{+i}$) hace referencia al acuerdo esperado en cada categoría i.

Este test pretende evaluar si la clasificación ha discriminado las categorías de interés con precisión significativamente mayor a la que se hubiese obtenido con una asignación aleatoria. Así, un valor κ igual a 1 indica un acuerdo pleno entre la realidad y el mapa, mientras que un valor cercano a 0 sugiere que el acuerdo observado es puramente debido al azar. De igual modo, un valor negativo indica también una clasificación pobre.

Una de las principales aplicaciones del coeficiente κ es comparar clasificaciones realizadas por distintos métodos con el objetivo de determinar si difieren significativamente en cuanto a su grado de ajuste con la realidad. Para ello, se puede calcular el error de muestreo asociado a κ ($\sigma^2(\kappa)$) aplicando luego la distribución normal para estimar intervalos de confianza [9]:

$$z = \frac{\kappa_1 - \kappa_2}{\sqrt{\sigma^2(\kappa_1) - \sigma^2(\kappa_2)}}$$

Análisis categórico multivariable: normalización de la matriz de confusión

Cuando se desea comparar la fiabilidad de dos mapas con distintos tamaños de muestreo, el estadístico κ no ofrece una valoración adecuada. Para solucionar este problema, Congalton [10] propuso aplicar un procedimiento multivariado para normalizar una matriz cuadrada. Se trata de un método iterativo que ajusta los totales de las filas y columnas a un valor común, mediante sucesivos incrementos o reducciones en los elementos de la matriz. El proceso se detiene cuando los marginales de cada fila y columna sumen 1, o un valor cercano.

Este proceso ofrece una nueva medida de fiabilidad global: basta calcular el valor medio de los elementos de la diagonal, los cuales siguen indicando el acuerdo entre mapa y realidad.

La situación ideal sería que todos los elementos de la diagonal de la matriz sean iguales a 1: esto indicaría un acuerdo perfecto entre realidad y mapa. Una clasificación pobre se evidenciaría con valores diagonales muy bajos.

Es importante tener en cuenta que las medidas obtenidas con este método pueden representar una estimación baja de de la fiabilidad real debido a las características propias un proceso de normalización.

CAPÍTULO 3

Redes Neuronales

3.1. Introducción

Las redes neuronales representan modelos simples del sistema nervioso central; son un conjunto de elementos altamente interconectados que tienen la habilidad de responder simultáneamente a distintas entradas y aprender en entornos cambiantes.

Las redes neuronales artificiales han demostrado ser efectivas como procesos computacionales en varias tareas, como por ejemplo, en el reconocimiento de patrones. Éstas exponen un gran número de características deseables, algunas de las cuales no se encuentran en los sistemas convencionales. A continuación se enumeran estas propiedades:

- Robustez y tolerancia a fallas.
- Posibilidad de manejar información difusa, con ruido, incompleta o inconsistente.
- Alto grado de paralelismo.
- Capacidad de generalizar.
- Aprendizaje adaptativo.

3.2. Redes neuronales artificiales

Las Redes Neuronales Artificiales o *RNAs*, en inglés, *Artificial Neuronal Networks* o *ANNs*, son modelos computacionales que surgieron como intento de conseguir formalizaciones matemáticas acerca de la estructura y el comportamiento del cerebro humano. Se basan en el aprendizaje a través de la experiencia, con la consiguiente extracción del conocimiento a partir de la misma.

El fin perseguido por una *RNA* es la emulación del sistema central biológico a través de procesadores artificiales, que incluso permitan evitar fallas o errores humanos. Así una *RNA* puede considerarse como un modelo de las actividades mentales, basado en la explotación del procesamiento local en paralelo y en las propiedades de la representación distribuida.

Los elementos básicos de un sistema neuronal biológico son las neuronas, agrupadas en redes compuestas por millones de ellas y organizadas a través de una estructura de capas. En un sistema neuronal artificial puede establecerse una estructura jerárquica similar, de forma que una *RNA* puede concebirse como una colección de procesadores elementales (*neuronas artificiales*), conectados entre sí o bien a entradas externas y con una salida que permite propagar la señal por múltiples caminos. Un conjunto de neuronas artificiales, tales que sus entradas provienen de la misma fuente y sus salidas se dirigen al mismo destino, conforma lo que se denomina *capa* o *nivel*. La agrupación de estos conjuntos constituye el sistema neuronal completo.

Cada procesador pondera las entradas que recibe. La modificación de estas ponderaciones es la clave del aprendizaje de la red. De esta forma, la red neuronal artificial aprende de sus propios errores a través de un procedimiento inductivo basado en la presentación de un conjunto de patrones informativos que permiten al sistema la generalización de conceptos a partir de casos particulares.

Una red neuronal puede definirse como un grafo dirigido con las siguientes propiedades:

- A cada nodo j se le asocia una variable de estado x_j,
- A cada conexión (i, j), entre los nodos i y j, se le asocia un peso $w_{ij} \in \mathbb{R}$.
- En muchos casos a cada nodo se le asocia un umbral de disparo θ_j.
- Para todo nodo j se define una función $f_j(x_i, w_{ij}, \theta_j)$, que depende del estado de todos los nodos unidos a él, de los pesos de sus conexiones y del umbral de activación para proporcionar un nuevo estado.

Considerando el lenguaje habitual de los grafos pueden establecerse las siguientes equivalencias:

- Un *nodo* se representa mediante una neurona.
- Una *conexión* se representa mediante una sinapsis.
- Una *neurona de entrada* es aquella sin conexiones entrantes.
- Una *neurona de salida* es aquella sin conexiones salientes.
- Las neuronas que no son de entrada ni de salida se denominan *neuronas ocultas*.

3.3. La neurona y la sinapsis

El elemento fundamental de los sistemas neuronales biológicos es la *neurona*. Una neurona es una célula pequeña que recibe estímulos electroquímicos de distintas fuentes, como de células sensoriales, y responde con impulsos eléctricos que son transmitidos a otras neuronas o a células efectoras (células, por lo general, del sistema inmunológico, que desempeñan una función específica en respuesta a un estímulo).

Existen aproximadamente 10^{12} neuronas en un ser humano adulto, cada una de las cuales se conecta en promedio con otras 10^5 unidades.

En el córtex cerebral se aprecia una organización neuronal horizontal en capas, que coexiste con una organización vertical en forma de columnas de neuronas. Asimismo, hay grupos neuronales localizados en zonas específicas del cerebro y especializados en ciertas tareas: área visual, córtex senso-motor, área auditiva y área olfativa, entre otros. El procesamiento de la información involucra la actuación conjunta de varios de estos subsistemas, que intercambian y comparten datos entre sí.

Considerando su tamaño microscópico, resulta sorprendente la capacidad de la neurona como procesador de señales eléctricas y de actividad bioquímica. Desde un punto de vista funcional, las neuronas constituyen unidades computacionales sencillas, integradas por: (Figura 3.1)

1. Un canal de recepción de información, las *dendritas*, que obtienen señales de entrada (*inputs*) procedentes de otras células (*interneuronas*) o del exterior (*neuronas receptoras* o *sensoriales*).
2. Un órgano de cómputo, el *soma*, que combina e integra todos los *inputs* recibidos (generalmente a través de funciones no lineales), emitiendo señales de salida en forma de estímulos nerviosos.
3. Un canal de salida, el *axón*, que envía el resultado generado por el soma a otras neuronas o bien, directamente al músculo. Las neuronas receptoras luego combinarán todas sus entradas para producir nuevas salidas.

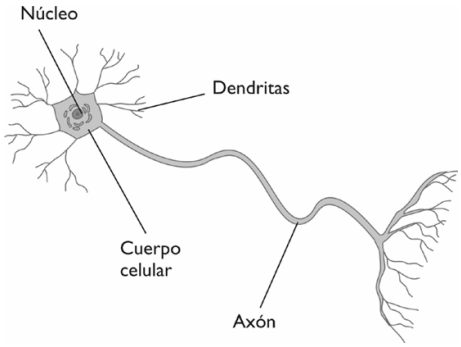


Figura 3.1: Partes de una neurona [15].

La conexión entre el axón de una neurona y las dendritas de otra recibe el nombre de *sinapsis*, y determina la fuerza y tipo de relación entre ellas. El impulso nervioso producido por una neurona se propaga por el axón y al llegar a un extremo, las fibras terminales pre-sinápticas liberan compuestos químicos llamados *neurotransmisores*. Éstos luego alterarán el estado eléctrico de la membrana de la neurona post-sináptica. (Figura 3.2)

En función del neurotransmisor liberado, el mecanismo puede resultar *excitador* o *inhibidor* para la neurona receptora. En el soma de una neurona se integran todos los estímulos recibidos a través de sus dendritas. Si como resultado se supera un potencial de activación, la neurona se dispara, generando un impulso que se transmitirá través del axón. Sino, se mantiene en reposo.

La repercusión de un estímulo nervioso en el estado excitatorio de la neurona receptora no es constante y se modifica con el tiempo en el proceso de aprendizaje. A esto se lo denomina *plasticidad sináptica*. No obstante, las sinapsis son unidireccionales, es decir, la información fluye siempre en un único sentido.

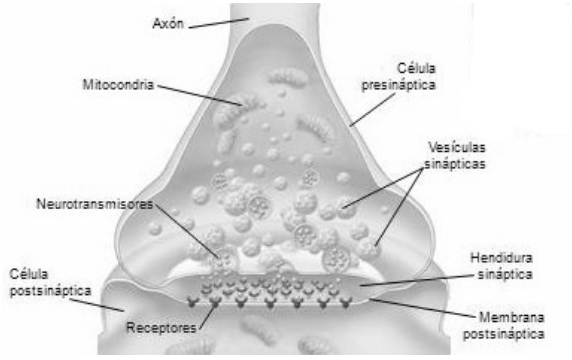


Figura 3.2: Sinapsis entre neuronas [16].

Una de las características que diferencian a las neuronas del resto de las células vivas es su capacidad para comunicarse. Si bien la intensidad de las sinapsis varía a lo largo del tiempo; esta plasticidad sináptica constituye, en gran medida, el proceso de aprendizaje. Durante el desarrollo del ser vivo, el sistema neuronal se va modificando con el objetivo de adquirir condiciones que no son innatas al individuo. De esta forma, se establecen nuevas conexiones, se rompen otras, se modelan las intensidades sinápticas o incluso se produce la muerte neuronal. Este tipo de modificaciones, especialmente las referentes a la intensidad de las conexiones, constituyen la base de las Redes Neuronales Artificiales.

Para construir un modelo formal y simplificado se destacan los siguientes aspectos:

- La neurona posee sólo dos estados: *exitatorio* o *de reposo*.
- Existen dos tipos de sinapsis: *excitatorias* e *inhibidoras*.
- La neurona es un dispositivo integrador: suma los impulsos que le llegan a sus dendritas.

- Cada sinapsis trasmite con mayor o menor intensidad los estímulos eléctricos de acuerdo a su *capacidad sináptica*.
- El aprendizaje consiste en modificaciones en las sinapsis.

3.4. Elementos y características principales de las RNA

Las redes neuronales artificiales, como modelos que intentan reproducir el comportamiento del cerebro, realizan una simplificación del sistema neuronal humano en base a sus elementos más relevantes e imitando su comportamiento algorítmicamente.

Una elección adecuada de las características de cada neurona artificial, de la estructura o arquitectura de la red y del modo de operación o aprendizaje, es el procedimiento convencional utilizado para construir redes capaces de realizar una determinada tarea.

A continuación se describen los principales elementos de las *RNAs*.

3.4.1. La neurona artificial

Las redes neuronales artificiales están formadas por una serie de procesadores elementales, denominados *neuronas artificiales*. Estas constituyen dispositivos simples de cálculo que, a partir de un vector de entradas procedentes del mundo exterior o de un vector de estímulos recibidos de otras neuronas, proporcionan una respuesta única (*salida*). Resulta útil la caracterización de tres tipos de neuronas artificiales: (Figura 3.3)

1. Las *neuronas de entrada*, que reciben señales desde el entorno, provenientes de sensores o de otros sectores del sistema.
2. Las *neuronas de salida*, que envían su señal directamente fuera del sistema una vez finalizado el tratamiento de la información (*salidas de la red*).
3. Las *neuronas ocultas*, que reciben estímulos y emiten salidas dentro del sistema sin mantener contacto alguno con el exterior. En ellas se lleva a cabo el procesamiento básico de la información.

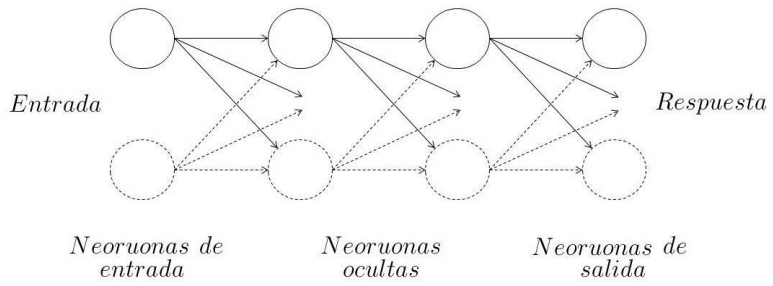


Figura 3.3: Tipos de neuronas artificiales [11].

Tratando de mimetizar las características más relevantes de las neuronas biológicas, cada neurona artificial se caracteriza por los siguientes elementos: (Figura 3.4)

- Un valor o estado de activación inicial (a_{t-1}), anterior a la recepción de estímulos.
- Unos estímulos o entradas a la neurona (x_j), con unos pesos asociados (w_{ij}).
- Una función de propagación que determina la entrada total a la neurona (Net_j).
- Una función de activación o de transferencia (f), que combina las entradas a la neurona con el estado de activación inicial para producir un nuevo valor de activación.
- Una función de salida (F), que transforma el estado final de activación en la señal de salida.

- Una señal de salida que se transmite, en su caso, a otras neuronas artificiales (y_j).
- Una regla de aprendizaje, que determina la forma de actualización de los pesos de la red.

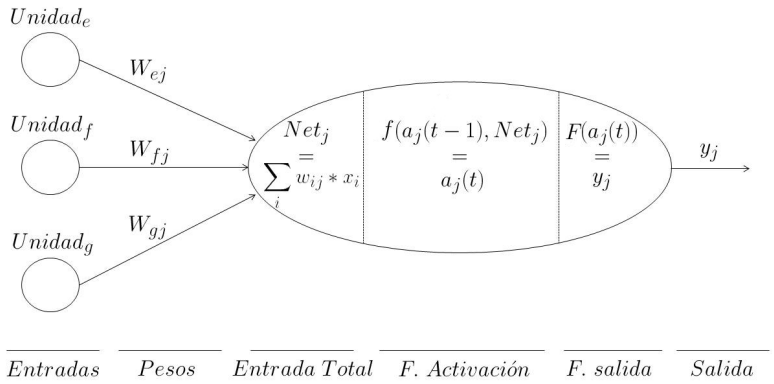


Figura 3.4: Modelo genérico de una neurona artificial [11].

Valor o estado de activación inicial

Todas las neuronas de la red presentan cierto estado inicial, de reposo o de excitación, que depende de su valor de activación. Este valor puede ser continuo (generalmente en el intervalo $[0, 1]$ o $[-1, 1]$) o discreto $(0, 1, -1, 1)$, limitado o ilimitado, según la entrada total recibida y el umbral de la propia neurona.

Si se designa como $a_i(t)$ la activación de la i-ésima unidad U_i respecto al momento de tiempo t, resulta posible definir el vector:

$$A(t) = [a_1(t), ..., a_i(t), ..., a_N(t)]$$

que representa el estado de activación de todas las neuronas de la red (de entrada, ocultas y de salida).

Estímulos o entradas a la neurona

Las variables procedentes del exterior que se presentan a las neuronas de entrada de la red pueden tener naturaleza binaria o continua, dependiendo del tipo de red y de la tarea analizada.

Las neuronas de las capas superiores reciben como entradas las salidas generadas por las unidades de las capas previas, acompañadas de un peso indicativo de su importancia relativa. Estas salidas también pueden ser binarias o continuas según el tipo de neurona que se considere.

De esta forma, cada neurona j-ésima de la red recibe un conjunto de señales que le proporcionan información del estado de activación de todas las neuronas con las que se encuentra conectada. Cada conexión (sinapsis) entre la neurona i y la neurona j está ponderada por un peso w_{ij}.

Algunos de los tipos de neuronas más conocidos son:

- Neuronas de tipo McCulloch-Pits: aquellas cuya salidas pueden tomar los valores $0, 1$.
- Neuronas de tipo Ising: aquellas cuyas salidas pueden tomar los valores $-1, 1$.
- Neuronas de tipo Potts: aquellas que pueden adoptar diversos valores discretos de salida $..., -2, -1, 0, 1, 2, ...$
- Neuronas de salida continua: aquellas cuya salida puede tomar cualquier valor en un intervalo determinado (habitualmente $[0, 1]$ ó $[-1, 1]$).

Función de propagación

Se denomina función de propagación a aquella regla que establece el procedimiento a seguir para combinar los valores de entrada y los pesos de las conexiones que llegan a una unidad.

En la práctica es común el empleo de una matriz W integrada por todos los pesos w_{ij} indicativos de la influencia que tiene la neurona i sobre la neurona j, siendo W un conjunto de elementos positivos, negativos o nulos. Si w_{ij} es positivo, la interacción entre las neuronas i y j es excitadora, esto es, siempre que la neurona i esté activa, la neurona j recibirá una señal que tenderá a activarla. Por el contrario, si w_{ij} es negativo, la sinapsis será inhibidora, por lo que si la unidad i está activa, enviará una señal a la neurona j que tenderá a desactivarla. Finalmente, si $w_{ij} = 0$, se considera que no existe conexión entre ambas neuronas.

La regla de propagación permite obtener, a partir de las entradas recibidas y de sus pesos asociados, el valor del potencial post-sináptico Net_j de la neurona j en un momento t, de acuerdo con una función σ_j tal que:

$$\mathrm{Net}_j(t) = \sigma_j(w_{ij}, x_i(t))$$

La función más habitual es la de tipo lineal y se basa en una suma ponderada de las entradas con los pesos sinápticos relacionados a ellas, es decir:

$$\mathrm{Net}_j(t) = \sum_i w_{ij} * x_i(t)$$

Desde el punto de vista formal puede interpretarse como el producto escalar entre un vector de entrada y los pesos de la red:

$$\mathrm{Net}_j(t) = \sum_i w_{ij} * x_i(t) = w_j^T \times x$$

siendo w_j^T el vector transpuesto representativo de los N pesos de entrada que llegan a la j-ésima neurona.

En algunos casos, el *potencial post-sináptico* considera también un umbral de disparo (θ_j). La inclusión de este parámetro deriva del comportamiento de las neuronas biológicas, que poseen umbrales internos de activación los cuales distorsionan el impacto causado por los estímulos recibidos. En el caso de las neuronas artificiales suele ser habitual agregar este elemento en la definición de Net_j como sigue:

$$\mathrm{Net}_j(t) = \sum_{i=1}^{N} w_{ij}(t) * x_i(t) + \theta_j(t)$$

Si el umbral se representa a través del valor $x_0 = 1$, con un peso asociado w_0 que determina el signo (positivo o negativo) y fuerza del mismo, se obtiene la siguiente expresión:

$$\mathrm{Net}_j(t) = \sum_{i=0}^{N} w_{ij}(t) * x_i(t)$$

Función de activación o transferencia

La función de activación combina el potencial post-sináptico de la j-ésima neurona (Net_j) con el estado inicial de la neurona $a_j(t-1)$ para producir un nuevo estado de activación acorde con la información recibida $a_j(t)$.

$$a_j(t) = f(a_j(t-1), \mathrm{Net}_j(t))$$

En muchos modelos de *RNAs* se considera que el estado actual de una neurona no depende de su estado previo, por lo que la expresión anterior se simplifica:

$$a_j(t) = f(\mathrm{Net}_j(t)) = f\left(\sum_{i=0}^{N} w_{ij}(t) * x_i(t)\right)$$

Generalmente la función de transferencia tiene carácter determinista, y en la mayor parte de los modelos es monótona, creciente y continua respecto al nivel de excitación de la neurona, tal como se observa en los sistemas biológicos. A menudo f_j es de tipo sigmoidal, y suele ser la misma para todas las unidades de cada capa.

Con carácter general, pueden distinguirse seis funciones de transferencia típicas:

1. La función *lineal* o *identidad*, que devuelve directamente el valor de activación de la neurona. Este tipo de función se utiliza en las redes de baja complejidad, como en el modelo Adaline.

2. La función *escalón* o *signo*, que representa salidas binarias (habitualmente $0, 1$ o $-1, 1$). En este caso si la activación de una neurona es inferior a un determinado umbral, la salida se asocia con un determinado *output*, y si es igual o superior al umbral, se asocia con el otro valor. Si bien las neuronas definidas por este tipo de funciones resultan fáciles de implementar, sus aplicaciones son limitadas, al restringirse a problemas binarios. Entre las redes que utilizan funciones de transferencias de tipo escalón cabe destacar el Perceptrón Simple, la red de Hopfield discreta y la neurona clásica de McCulloch Pitts.

3. La función *mixta* o *lineal a tramos*, en la que si la activación de una unidad es menor que un límite inferior preestablecido, la salida se asocia con un determinado valor; si la activación es igual o superior que un límite superior, la salida se asocia con otro valor; si el nivel de activación se encuentra comprendido entre ambos límites, se aplica la función lineal o identidad. Esta alternativa puede considerarse como una función lineal saturada en sus extremos, siendo de sencillez computacional y resultando más plausible desde el punto de vista biológico.

4. La función *sigmoidea*, definida en un determinado intervalo monotónico con límites superiores e inferiores. Entre las funciones sigmoideas de transferencia más aplicadas se destacan la función sigmoide o logística, la función tangente hiperbólica, y la función sigmoide modificada propuesta por Azoff [17]. Las funciones sigmoideas se caracterizan por presentar una derivada simple positiva e igual a cero en sus límites asintóticos, que toma su valor máximo cuando $x = 0$. Así, estas funciones admiten la aplicación de las reglas de aprendizaje típicas de la función escalón, con la ventaja adicional de que la derivada se encuentra definida en todo el intervalo, lo que permite emplear algoritmos de entrenamiento más avanzados.

5. La función *gaussiana*, que adquiere la forma de campana de Gauss cuyo centro, radio y apuntamiento son susceptibles a adaptación, lo que las hace muy versátiles. Las funciones gaussianas se suelen aplicar a redes complejas con m capas ocultas ($m \geq 2$) que requieren reglas de propagación basadas en el cálculo de distintas cuadráticas entre los vectores de entrada y los pesos de la red (por ejemplo, la distancia euclídea al cuadrado).

6. La función *sinusoidal*, que genera salidas continuas en el intervalo $[-1, 1]$. Estas funciones suelen emplearse en los casos en los que se requiere explícitamente una periodicidad temporal.

Función de salida

Cada neurona U_j tiene asociada una función de salida F que transforma el estado actual de activación $a_j = f(\mathrm{Net}_j(t))$ en una señal de salida $y_j(t)$:

$$y_j(t) = F(a_j) = F(f(\mathrm{Net}_j(t)))$$

El vector que contiene las salidas de todas las neuronas en un instante t se representa como:

$$Y(t) = F(a_1(t), a_2(t), ..., a_j(t), ..., a_N(t))$$

$$Y(t) = F(f(\mathrm{Net}_1(t)), f(\mathrm{Net}_2(t)), ..., f(\mathrm{Net}_j(t)), ...f(\mathrm{Net}_N(t)))$$

Habitualmente, la función de salida coincide con la función identidad $F(x) = x$, por lo que el estado de activación de la neurona se asocia con su salida final:

$$y_j(t) = F(a_j(t)) = a_j(t) = f(\mathrm{Net}_j(t)) = f\left(\sum_{i=0}^{N} w_{ij}(t) * x_i(t)\right)$$

Esta situación es típica de las redes más utilizadas en la práctica, como la Adaline, el Perceptrón Simple o el Perceptrón Multicapa.

En otros casos, la salida final de la neurona se calcula mediante una función estocástica del estado de activación inicial, por lo que la neurona presentará un comportamiento probabilístico. Éste es el caso de las funciones de transferencia utilizadas en redes como la Máquina de Boltzmann o la Máquina de Cauchy.

Señal de salida

En el caso de problemas de clasificación suele considerarse un conjunto finito de salidas (en muchos casos binarias), mientras que las tareas de ajuste de regresión suelen precisar salidas continuas en un determinado intervalo. El tipo de salida deseada determinará la función de transferencia y salida que debe implementarse en las neuronas de la última capa de la red.

Regla de aprendizaje

Biológicamente se acepta que la información memorizada en el cerebro depende de los valores sinápticos representativos de las conexiones existentes entre las neuronas. De forma similar, en las *RNAs* se puede considerar que el conocimiento se encuentra representado en los pesos de las conexiones entre las neuronas artificiales, por lo que el proceso de aprendizaje o entrenamiento implica cierto número de cambios en estas conexiones.

Ahora bien, cada modelo neuronal dispone de sus propias técnicas de aprendizaje, que dependen de la arquitectura de la red y del algoritmo de entrenamiento implementado.

3.4.2. La arquitectura de las RNAs

La topología o arquitectura de una *RNA* hace referencia a la organización y disposición de las neuronas de la red y a las conexiones entre ellas.

La arquitectura de una red neuronal depende de cuatro parámetros principales:

1. El número de capas del sistema.
2. El número de neuronas por capa.
3. El grado de conectividad entre las neuronas.
4. El tipo de conexiones neuronales.

Las arquitecturas neuronales pueden clasificarse de acuerdo a distintos criterios que se detallan a continuación.

Clasificación de las RNAs según su estructura en capas

- *Redes monocapas*: Compuestas por una única capa de neuronas entre las cuales se establecen conexiones laterales y en ocasiones autorrecurrentes. Este tipo de redes suele utilizarse para la resolución de problemas de autoasociación y clusterización.
- *Redes multicapa (layered networks)*: Se corresponde con las *RNAs* cuyas neuronas se organizan en varias capas: de entrada, oculta(s) y de salida. La capa a la que pertenece una neurona puede distinguirse mediante la observación del origen de las señales que recibe y el destino de la señal que genera.

Clasificación de las RNAs según el flujo de datos de la red

- *Redes unidireccionales o de propagación hacia adelante (feedforward)*: En éstas, ninguna salida neuronal es entrada de unidades de la misma capa o de capas precedentes. Por lo tanto, la información circula en un único sentido: desde las neuronas de entrada hacia las neuronas de salida de la red.

- *Redes de propagación hacia atrás (feedback)*: En éstas las salidas de las neuronas pueden servir de entradas a unidades del mismo nivel (conexiones laterales), o de niveles previos. Las redes de propagación hacia atrás que presentan lazos cerrados se denominan sistemas recurrentes.

Clasificación de las RNAs según el grado de conexión

- *Redes neuronales totalmente conectadas*: En este caso cada una de las neuronas de una capa se encuentran conectadas con todas las neuronas de la capa siguiente (redes no recurrentes), o con todas las neuronas de la capa anterior (redes recurrentes).
- *Redes neuronales parcialmente conectadas*: En este caso no se da la conexión total entre neuronas de diferentes capas.

Clasificación de las RNAs según el tipo de respuesta de la red

- *Redes heteroasociativas*: Redes entrenadas para que ante la presentación de un determinado patrón A, el sistema responda con otro diferente B. Estas *RNAs* precisan al menos dos capas: una para captar y retener la información de entrada y otra para mantener la salida con la información asociada. Las redes heteroasociativas pueden clasificarse a su vez, según el objetivo pretendido con su utilización, distinguiéndose las *RNAs* destinadas a computar una función matemática a partir de las entradas que reciben, las redes utilizadas para tarea de clasificación y las redes empleadas para la asociación de patrones, entre otras.
- *Redes autoasociativas*: Redes entrenadas para que se asocie un patrón consigo mismo. Su interés reside en que, ante la presentación de un patrón A' afectado por ruido, su respuesta sea el patrón original A. Estas redes pueden implementarse con una única capa de neuronas que comenzará reteniendo la información de entrada y terminará representando la información autoasociada. Si se desea mantener la información de entrada y salida, deberán añadirse capas adicionales. Estos modelos suelen emplearse en tareas de filtrado de información, para analizar las relaciones de vecindad entre los datos considerados (*clustering*) y para resolver problemas de optimización.

3.4.3. Modos de operación: aprendizaje y recuerdo

En el ámbito de las *RNAs*, el aprendizaje puede definirse como el proceso por el cual la red neuronal crea, modifica o destruye sus conexiones (pesos) en respuesta a la información de entrada. Esta característica resulta de crucial importancia ya que los sistemas neuronales tienen la capacidad de generalizar un determinado cómputo en base al conocimiento adquirido al procesar un conjunto de ejemplos.

En la mayoría de los modelos neuronales existen dos modos diferenciados de funcionamiento: el modo *aprendizaje* o *entrenamiento* y el modo *recuerdo*, *ejecución* u *operación*; siendo necesario ejecutar inicialmente la fase de aprendizaje para establecer los pesos de la red y, posteriormente utilizar el modo recuerdo manteniendo los pesos fijos. No obstante, existen modelos neuronales en los que las fases de aprendizaje y recuerdo coinciden, de forma que la red puede aprender y modificar sus conexiones durante todo su ciclo de operación, razón por la cual los pesos varían de forma dinámica cada vez que se presenta al sistema una nueva información.

Fase de aprendizaje o entrenamiento

Cuando se construye un sistema neuronal no sólo se definen tanto el prototipo de neurona como la arquitectura de la red a emplear. También se establecen los pesos iniciales de las conexiones utilizando valores aleatorios o nulos.

A partir de este modelo es necesario entrenar la *RNA* para solucionar el problema objeto de estudio. El aprendizaje de la red se logra mediante dos procesos diferentes pero complementarios:

- Proceso de modelado de las sinapsis de la red: Los pesos de la *RNA* se ajustan a través de una regla de aprendizaje cuyo objetivo es minimizar una determinada función de error o coste.

Si se denomina $w_{ij}(t)$ al peso que conecta a la neurona pre-sináptica i con la post-sináptica j en el momento t, el estado de dicha sinapsis en el momento $t+1$ se determinara con la siguiente expresión:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

siendo $\Delta w_{ij}(t)$ la variación generada en el peso por la regla de aprendizaje en el instante t.

Generalmente el proceso de aprendizaje es iterativo y finaliza cunado la red obtiene el rendimiento deseado (error máximo permitido) o debido a alcanzar una cantidad límite de ciclos.

- Proceso de creación y/o destrucción de las neuronas en la red: La construcción de neuronas hace referencia a la introducción de nuevas unidades en el sistema con los respectivos pesos asociados (modelos de redes constructivas); mientras que la destrucción implica la eliminación de una neurona de la red con la consiguiente depuración de los pesos ligados a ella (modelos de poda).

Existen dos tipos básicos de reglas de aprendizaje que pueden utilizarse para la actualización de los pesos:

- **Aprendizaje supervisado**: Éste se caracteriza por la presencia de un agente externo (supervisor o maestro) que controla el proceso de entrenamiento al establecer la respuesta que debería generar la red a partir de una entrada determinada.

  El supervisor compara la salida de la red con la esperada y, si existen diferencias, los pesos de las conexiones se ajustan iterativamente en base al el error cometido. Este proceso se reitera hasta que el resultado se aproxime al esperado con cierto grado de confianza.

  Desde el punto de vista formal, sea $E(W)$ la función que representa el error esperado de la red expresado en base a sus pesos sinápticos. El aprendizaje supervisado tiene como objetivo hallar una función multivariable desconocida, $f : \mathbb{R}^N \to \mathbb{R}^M$ a partir de submuestras de patrones de entrada-salida (x, y), donde $x \in \mathbb{R}^N$ e $y \in \mathbb{R}^M$.

  El modelado de esta función se basa en la minimización iterativa de $E(W)$ mediante algún algoritmo de aproximación.

  El tipo de algoritmo de aproximación empleado permite distinguir tres tipos de aprendizaje supervisados: por corrección de error, por refuerzo o de tipo estocástico.

  1. *Aprendizaje por corrección de error*: Constituye el tipo de aprendizaje supervisado más utilizado en la práctica. Su funcionamiento se basa en el ajuste de los pesos de las conexiones de la red a partir de la diferencia entre los valores deseados y los obtenidos por el sistema.
     Una de las reglas más sencillas de aprendizaje por corrección de error es la siguiente:

     $$\Delta w_{ij} = \alpha * x_i(t_j - y_j)$$

     donde:

     - Δw_{ij} es la variación en el peso de la conexión entre las neuronas i y j,
     - x_i es la i-ésima entrada a la neurona j-ésima,
     - t_j es la salida deseada para la neurona j,
     - y_j es la salida obtenida en la j-ésima neurona, y
     - α es el factor o taza de aprendizaje

     Esta regla presenta la restricción de no considerar la magnitud del error global cometido durante el proceso de aprendizaje. Sin embargo es empleada en la *RNA Perceptrón Simple* [18].

     Para superar esta limitación, Widrow y Hoff [19] desarrollaron un nuevo algoritmo de aprendizaje más rápido y con mayor campo de aplicación. Se lo conoce como "*Regla del error mínimo cuadrado*" ("*Least-Mean-Square-Error*") o "*Regla de Widrow-Hoff*" para

las funciones de activación de tipo lineal; y con el nombre de "*Regla delta*" en el caso de funciones de activación de tipo sigmoideo.

El método parte de la función de error global cometido por una red durante su entrenamiento:

$$E(W_{ij}) = \frac{1}{2} * \sum_{k=1}^{P} \sum_{j=1}^{M} (y_j^{(k)} - t_j^{(k)})^2$$

siendo P es el número de patrones que debe aprender la red y M el número de neuronas de salida.

En base a la ecuación anterior, la variación relativa del error puede calcularse de la siguiente manera:

$$\Delta w_{ij} = -\alpha \frac{\partial E(W_{ij})}{\partial W_{ij}} = \alpha(t_j - y_j)x_i$$

o bien de forma acumulativa para todos los patrones:

$$\Delta w_{ij} = -\alpha \frac{\partial E(W_{ij})}{\partial W_{ij}} = \alpha \sum_{k=1}^{P} (t_j^{(\mu)} - y_j^{(\mu)})x_i^{\mu}$$

La generalización de la regla delta constituye el denominado algoritmo de retropropagación del error ('*backpropagation*').

Suponiendo funciones de activación sigmoidales, este método emplea los siguientes mecanismos de ajuste de los pesos de la red, el primero en caso de ser j una neurona de salida y el segundo en caso de ser una neurona oculta:

$$\Delta w_{ij} = \alpha * \delta_j^0 * x_i = \alpha * ((t_j - y_j) * y_j * (1 - y_j)) * x_i$$

$$\Delta w_{ij} = \alpha * \delta_j^h * x_i = \alpha * \left(\sum_k \delta_j^0 * W_{jk} * y_j * (1 - y_j) \right) * x_i$$

donde k hace referencia a todas las neuronas de la capa inmediatamente superior de la neurona j.

2. *Aprendizaje por refuerzo*: En este aprendizaje la tarea del supervisor se limita a indicar mediante una señal de refuerzo (*éxito* $= 1$, *fracaso* $= -1$) si la salida obtenida por la red se ajusta o no a la deseada. En función de ello se procede al ajuste de los pesos utilizando un mecanismo basado en probabilidades.
3. *Aprendizaje estocástico*: Se basa en la introducción de cambios aleatorios en los valores de los pesos de la red, evaluando su efecto a partir de la salida deseada y de una determinada distribución de probabilidad. El aprendizaje consiste en minimizar la energía del sistema a través del ajuste de los pesos: se realizan cambios aleatorios de los valores de los pesos y se determina la energía de la red tras estas modificaciones. Si la energía es menor después del cambio, se acepta la modificación, en caso contrario, la inclusión del cambio depende de la distribución de probabilidad preestablecida.

- **Aprendizaje no supervisado o autosupervisado**: Éste no requiere información externa para ajustar los pesos de las conexiones neuronales. La red, por medio de un algoritmo de aprendizaje predefinido, estima una función de densidad probabilística $p(x)$ $(x \in \mathbb{R}^P)$ que describe la distribución de sus entradas.

  De esta manera, el sistema es capaz de reconocer las peculiaridades, correlaciones o categorías presentes en el conjunto de entradas, extrayendo rasgos o agrupando patrones según su similitud.

  Para que la red obtenga resultados de calidad, es necesario un cierto nivel de redundancia.

  Dado que en este tipo de sistemas no existe una salida deseada, existen distintas formas de interpretar los resultados expuestos. En algunos casos, la salida representa el *grado de*

similitud entre la información que se ha presentado y la que se ha mostrado hasta entonces. En otros casos, la *RNA* puede realizar distintos tipos de tareas tales como tareas de categorización, tareas de *prototipos* (obteniendo ejemplares representativos de las clases a las que pertenecen las entradas), o tareas de codificación (generando salidas que representan valores cifrados de las entradas).

En base a la última aplicación de la regla de aprendizaje autosupervisado, se puede llevar a cabo una *asociación de características* (*feature mapping*) tal que las neuronas de salida simbolicen un mapa de las propiedades de los datos de entrada.

Se destacan dos propuestas diferentes de aprendizaje no supervisado:

- *Aprendizaje hebbiano*: Los algoritmos de aprendizaje no supervisado de carácter hebbiano se basan en el siguiente postulado formulado por Donald O. Hebb [20]:

  "Cuando un axón de una celda A está lo suficientemente cerca para conseguir excitar a una celda B y repetida o persistentemente toma parte en su activación, algún proceso de crecimiento o cambio metabólico tiene lugar en una o ambas celdas, de tal forma que la eficiencia de A aumenta cuando la celda B está activa".

  De esta forma, identificando las celdas con neuronas fuertemente conectadas y la eficiencia con la intensidad o magnitud del peso entre ellas, puede afirmarse que el aprendizaje hebbiano consiste en el ajuste de los pesos de las conexiones a partir de la correlación existente entre las salidas generadas por cada celda:

  $$\Delta w_{ij} = y_i * y_j$$

  La regla de Hebb es de tipo no supervisado ya que la modificación de los pesos depende de las salidas obtenidas tras la presentación de un estímulo determinado, con independencia de que coincidan o no con las deseadas.

  En el aprendizaje hebbiano, múltiples neuronas de salida pueden activarse simultáneamente.

- *Aprendizaje competitivo y cooperativo*: En estas redes, las neuronas compiten (y cooperan) con el objetivo de que cuando se presente cierta entrada, sólo una de las neuronas de salida se active, la denominada 'neurona vencedora' (*'winner-take-all unit'*). El resto de las neuronas quedan anuladas y a ellas se les asigna valores de respuesta mínimos.

  Para llevar a cabo este proceso se establecen conexiones de autoexcitación si el aprendizaje es cooperativo, y conexiones de inhibición si el aprendizaje es competitivo.

  El objetivo de este tipo de aprendizaje es la clasificación de los datos de entrada en grupos de patrones similares entre sí (*'clusters'*). Las clases resultantes son establecidas por la propia red sin supervisión externa.

  Este aprendizaje no supervisado ha sido ampliamente utilizado para el desarrollo de *RNAs*, en particular en los mapas autoorganizados desarrollados por Teuvo kohonen.

Fase de recuerdo, ejecución u operación

En la mayoría de los modelos neuronales la red fija sus pesos y estructura al culminar la fase de entrenamiento, quedando preparada para procesar nuevos datos a partir del conocimiento extraído de la muestra de aprendizaje. Este modo de operación se denomina 'modo recuerdo' (*recall*), 'modo de ejecución' o 'modo de operación'.

La fase de recuerdo presenta características diferentes para las redes unidireccionales y para las redes con retroalimentación. En las primeras, las neuronas responden ante cada patrón de entrada generando directamente la salida del sistema, sin plantearse problemas de estabilidad en el modelo. Contrariamente, en las segundas, se requiere de ciertas condiciones para que la red acabe convergiendo a un estado estable, dado que representan sistemas dinámicos no lineales.

Existen distintos teoremas generales que establecen las condiciones necesarias para garantizar la estabilidad de la respuesta de la red bajo determinados requisitos, tales como el teorema de

Cohen-Grossberg para las redes autoasociativas no adaptativas [21], el teorema de Simpson [22], el teorema de Cohen-Grossberg-Kosko para las redes autoasociativas adaptativas [23]. Con carácter general, estos teoremas establecen que si se define una función de error monótona decreciente en todos los puntos, la red es estable.

3.5. Evaluación del aprendizaje de la red

Uno de los aspectos más importantes en la construcción y desarrollo de las *RNAs* es la capacidad de la red para generalizar a partir de ejemplos, evitando la simple memorización de patrones durante la etapa de aprendizaje y proporcionando una respuesta correcta ante individuos no presentados en la etapa de entrenamiento.

En el proceso de entrenamiento de la red se debe considerar, además del error de aprendizaje, el denominado *error de generalización*, calculado a partir de un conjunto de test distinto al de la muestra de entrenamiento.

Obtener una adecuada generalización de la red resulta de mayor importancia que conseguir un error reducido en la muestra de entrenamiento, dado que esto indicará que el sistema a capturado correctamente las relaciones subyacentes entre los datos.

Es un hecho experimental observable que si se entrena la red para alcanzar un error de aprendizaje muy reducido (por ejemplo, inferior al 1 %), el error de test se degrada, obteniendo una gráfica similar a la de la figura 3.5.

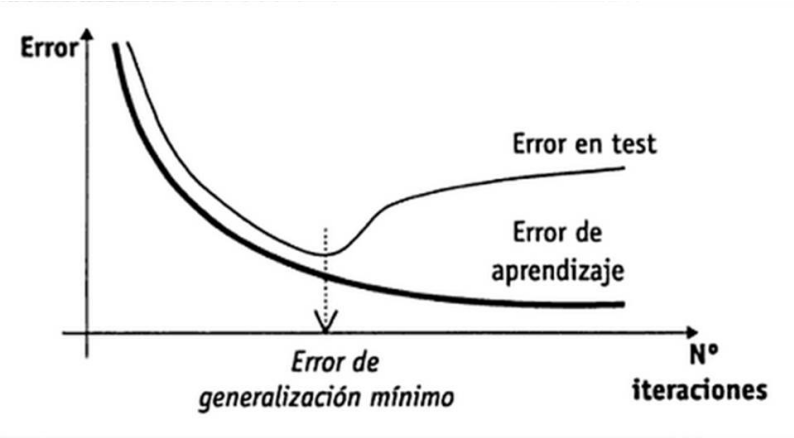


Figura 3.5: Evaluación del aprendizaje de una RNA [11].

Tras una etapa inicial en la que la tasa de error puede oscilar, el error de aprendizaje disminuye monótonamente mientras que el error de generalización se decrementa hasta cierto punto en el cual comienza a incrementarse como consecuencia del excesivo ajuste de la red a las particularidades de los patrones de entrenamiento.

El fenómeno explicado anteriormente se conoce como *sobreaprendizaje* (*overtraining*). Puede evitarse usando procesos de validación cruzada (*cross validation*), es decir, entrenando y validando a la red simultáneamente para detectar un punto óptimo de aprendizaje.

Los procesos de validación cruzada son ampliamente utilizados en el desarrollo de redes supervisadas como por ejemplo en la red Perceptrón Multicapas.

Una vez entrenada la *RNA* resulta necesario evaluar los resultados obtenidos para determinar su validez práctica. McNelis [24] propone dos grandes criterios para esta evaluación:

- Criterios 'dentro de la muestra'
- Criterios 'fuera de la muestra'.

3.5.1. Criterios 'dentro de la muestra'

Los criterios 'dentro de la muestra' tratan de analizar la capacidad de la *RNA* para caracterizar correctamente al conjunto de datos utilizado en su entrenamiento. Se han propuesto distintas medidas alternativas de desempeño. Las más destacadas se analizan a continuación.

Coeficientes de correlación múltiple cuadrática

Los *coeficientes de correlación múltiple cuadrática*, R^2 y R^2_{ajustado} son indicativos de la proximidad existente entre las salidas generadas por la red (y_j) y las deseadas (t_j):

$$R^2 = \frac{\sum_{i=1}^{P}(y_i - \bar{t})^2}{\sum_{i=1}^{P}(t_i - \bar{t})^2} = 1 - \frac{\sum_{i=1}^{P}(t_i - y_i)^2}{\sum_{i=1}^{P}(t_i - \bar{t})^2}$$

$$R^2_{\text{ajustado}} = \frac{P * R^2 - N}{P - N}$$

donde:

- P es el total de individuos empleados para el aprendizaje del sistema,
- $\bar{t}$ es la salida media esperada para el conjunto de ejemplares analizados, y
- N es el número de variables explicativas (o independientes) incluidas en el modelo.

Estos coeficientes toman valores en el intervalo $[0, 1]$, donde 0 indica la inexistencia de correlación entre la variable dependiente y el modelo desarrollado, mientras que el valor 1 informa la existencia de correlación perfecta.

Si se analizan problemas de clasificación, resulta adecuado observar las *tablas de contingencia*. Éstas permiten distinguir entre los distintos tipos de error según las categorías deseadas y las determinadas por la red.

En la siguiente tabla se resumen los diferentes errores para problemas de clasificación binarios mutuamente excluyentes:

	Positivos deseados ($n^+ = \text{TP} + \text{FN}$)	**Negativos deseados** ($n^- = \text{FP} + \text{TN}$)
Positivos observados ($o^+ = \text{TP} + \text{FP}$)	**Positivos verdaderos** (TP)	**Positivos falsos** (FP)
Negativos observados ($o^- = \text{FN} + \text{TN}$)	**Negativos falsos** (FN)	**Negativos verdaderos** (TN)

Tabla 3.1: Tabla de contingencia para un problema de clasificación binario [11].

A partir de la tabla de contingencia pueden obtenerse las siguientes relaciones:

- Sensibilidad, razón de positivos verdaderos o razón de precisión: $\frac{TP}{n^+}$.
- Especificidad o razón de negativos verdaderos: $\frac{TN}{n^-}$.
- Falsa alarma: $1 - \frac{TN}{n^-}$.
- Predicción positiva o razón de recuerdo: $\frac{TP}{o^+}$.
- Predicción negativa: $\frac{TN}{o^-}$.
- Exactitud o desempeño: $\frac{(TP+TN)}{(n^+ + n^-)}$.
- Error total: $1 - \frac{(TP+TN)}{(n^+ + n^-)}$.

Criterio de información Hannan-Quinn

Por su parte, el criterio de información *'Hannan-Quinn'* [25] resulta muy útil para la evaluación de modelos autorregresivos. Este criterio incluye un término de penalización que considera el número de parámetros del modelo (k). EL objetivo es encontrar la *RNA* que minimice la siguiente expresión:

$$H - Q_{if} = \left(ln(\sum_{i=1}^{P} \frac{(t_i - y_i)^2}{P}) \right) + \frac{k * ln[ln(P)]}{P}$$

Criterios de información de Akaike y de Schwartz

Otras medidas de validación 'dentro de la muestra', que además incluyen términos de penalización, son el *criterio de Akaike* [26] y el *criterio de Schwartz* [27]:

$$Akaike = \left(ln(\sum_{i=1}^{P} \frac{(t_i - y_i)^2}{P}) \right) + \frac{2k}{P}$$

$$Schwartz = \left(ln(\sum_{i=1}^{P} \frac{(t_i - y_i)^2}{P}) \right) + \frac{k * ln(P)}{P}$$

Finalmente, el *análisis de los residuos* o diferencias entre la salida deseada y la salida obtenida por la *RNA* para cada patrón puede proporcionar información muy valiosa sobre la presencia de sesgo en el modelo (distribución sistemática de residuos), simetría (análisis de aleatoriedad de los residuos) y normalidad (presencia o no de residuo blanco en la distribución de los residuos).

3.5.2. Criterios 'fuera de la muestra'

Existen distintos criterios 'fuera de la muestra' que analizan la capacidad de generalización de las *RNAs*, o capacidad para responder correctamente ante la presentación de patrones nuevos a la red.

Resulta necesario definir una función de pérdida L a utilizar para estimar el error de predicción cometido por el modelo. Las funciones más habituales son: el *error absoluto* o *error cuadrático* (en problemas de aproximación de funciones), y el *error total de clasificación* procedete de las tablas de contingencia (en problemas de clasificación) (Tabla 4.2).

Función de error	**Definición**	**Definición alternativa**
Error medio absoluto (*mean absolute error*)	$\text{MAE} = \frac{\sum_{i=1}^{P}\sum_{j=1}^{M}\lvert y_{ij}-t_{ij}\rvert}{P}$	$\text{MAE} = \frac{\sum_{i=1}^{P}\sum_{j=1}^{M}\lvert y_{ij}-t_{ij}\rvert}{P*M}$
Error cuadrático medio (*mean squared error*)	$\text{MSE} = \frac{\sum_{i=1}^{P}\sum_{j=1}^{M}(y_{ij}-t_{ij})^2}{P}$	$\text{MSE} = \frac{\sum_{i=1}^{P}\sum_{j=1}^{M}(y_{ij}-t_{ij})^2}{P*M}$
Raíz del error cuadrático medio (*root mean squared error*)	$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{P}\sum_{j=1}^{M}(y_{ij}-t_{ij})^2}{P}}$	$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{P}\sum_{j=1}^{M}(y_{ij}-t_{ij})^2}{P*M}}$

Tabla 3.2: Funciones de error de predicción [11].

Asimismo se han definido distintas variantes del error cuadrático medio para modelos lineales, tales como el criterio *generalized cross-validation* [28] y el criterio *predicted squared error* [29]. En el caso de los modelos no lineales, la variante más destacada es la medida *generalized prediction error* [30].

Una vez definida la función de error a utilizar pueden distinguirse distintas reglas de validación 'fuera de la muestra', entre los que se destacan:

- Error aparente o de resustitución (*'apparent error'* o *'resubstitution error'*).
- División de los datos o técnicas de 'entrenamiento y test' (*'test-and-train'*).
- Modelos de remuestreo (*'resampling'*).

Error aparente o de resustitución

El error de resustitución, o error aparente, estima el porcentaje de error cometido (Err) sobre la muestra empleada para construir el modelo [31].

Es preciso distinguir el error aparente del error de generalización: el error de generalización se calcula a partir del error aparente más un término de sesgo (generalmente positivo):

$$\text{ErrorVerdadero} = \text{ErrorResustitucion} + \text{Sesgo}(\hat{\beta})$$

$$\hat{Err} = \bar{err} + \hat{\beta}$$

La estimación del error total ($\hat{Err}$) constituye el objetivo principal no sólo de esta técnica, sino también de las presentadas a continuación.

División de los datos o técnicas de 'entrenamiento y test'

La metodología 'entrenamiento y test' divide aleatoriamente la muestra inicial de tamaño P en dos submuestras independientes seleccionadas, habitualmente, de forma aleatoria:

- La submuestra de aprendizaje, dedicada a la construcción del modelo (de tamaño P_1), y
- La submuestra de test, dedicada a la validación del modelo (de tamaño $P_2 = P - P_1$).

El tamaño de la submuestra de test puede oscilar entre el 5 % y el 90 % de la muestra, si bien suele considerarse la regla '$\frac{2}{3}$ (aprendizaje) - $\frac{1}{3}$ (test)' [32].

La estimación del error verdadero adopta la expresión:

$$\hat{Err}_{test-and-train} = \frac{1}{P} \sum_{j=P_1+1}^{P} \mid t_j - y_j \mid$$

Si bien este método resulta sencillo de implementar, en el caso de muestras de tamaño reducido, el número de individuos empleados en el entrenamiento también es bajo, y por consiguiente se decrementa la capacidad de categorización de la red.

Modelos de remuestreo

Las técnicas de remuestreo (del inglés, *'resampling'*) constituyen una propuesta robusta y de gran validez para la estimación de la capacidad de generalización de los prototipos desarrollados.

Estos modelos consideran múltiples muestras de aprendizaje ($k > 1$), adquiridas a partir de la muestra original. Los individuos incluidos en cada submuestra se emplean para la validación de los resultados obtenidos.

De esta forma, se obtienen k estimaciones del error de generalización, las que son posteriormente combinadas para obtener tanto una medida central final, así como intervalos de confianza del error de generalización.

Entre los métodos de remuestreo más utilizados en la práctica se destacan:

- **Validación cruzada y variantes**

  Los procedimientos de validación cruzada (*'cross validation'*) se basan en la eliminación, a partir de la muestra original, de una submuestra de datos de tamaño k ($k < P$). La red neuronal se entrena con los $P - k$ datos restantes, testeando su validez con la submuestra inicial. El proceso se repite hasta que todos los puntos son eliminados una vez, obteniéndose $\frac{P}{k} = G$ modelos parciales mutuamente excluyentes y de tamaño aproximadamente igual.

Si $G = 2$, el proceso se conoce como *'validación cruzada'* ('cross validation') y es coincidente con la técnica 'entrenamiento y test'. Si $G > 2$, el método es denominado *'validación cruzada por grupos'* (*'group cross validation'*):

$$\hat{Err}_{\mathrm{GCV}(\frac{P}{k})} = \frac{1}{P}\sum_{j=1}^{\frac{P}{k}}\sum_{i=1}^{k} \mid t_{(j-1)k+1} - y_{(j-1)k+1} \mid$$

El método de validación cruzada por grupos es más complejo que los anteriores pero evita la pérdida de datos y obtiene resultados más robustos en presencia de muestras de tamaño reducido.

Generalmente se suele considerar 10 grupos distintos de validación cruzada para garantizar una estimación fiable del error de generalización.

Existen distintas variantes del método, entre las que se distinguen las siguientes [32]:

- ***Complete (group) Cross-Validation***: Esta variante considera todas las posibles combinaciones de individuos en submuestras de test de tamaño k a partir de la muestra original de tamaño P, de forma que la estimación del error verdadero se promedia respecto al total de modelos $\binom{P}{k}$.
- ***Multiple (group) Cross-Validation***: Esta variante lleva a cabo una repetición completa del proceso de validación cruzada un número m de veces, tal que en la i-ésima iteración ($i = 1, .., m$) se ejecuta un proceso GCV_i basado en particiones distintas a las iteraciones previas. De esta forma, se incrementa el número de validadores (G^*m) al mismo tiempo que se mantiene un número de patrones suficiente para asegurar la validez de los resultados obtenidos.
- ***Stratified (group) Cross-Validation***: Esta variante obtiene réplicas estratificadas a partir de la muestra original, de forma que cada una de ellas contiene aproximadamente la misma proporción de individuos de cada clase que la muestra inicial.

- ***'Jackknife'***

  El modelo *'jackknife'* [33, 34] calcula P subconjuntos distintos de datos a partir de la muestra original, mediante la eliminación secuencial de un ejemplar en cada muestra. Cada subconjunto de $P - 1$ elementos se utiliza para el aprendizaje de la red, mientras que el individuo eliminado es usado para contrastar el modelo.

  Finalmente, el error de generalización se estima como:

$$\hat{Err}_{\mathrm{jackknife}} = \frac{1}{P}\sum_{j=1}^{P} \mid t_j - y_j \mid$$

  La metodología *'jackknife'* resulta costosa computacionalmente, sin embargo, las estimaciones conseguidas para el error verdadero son más robustas que en los casos anteriores.

- ***'Bootstrap'***

  El auténtico potencial de las técnicas de *'resampling'* procede del remuestreo con reemplazo desarrollado a través de las diferentes variantes del método *'bootstrap'* [35, 36]. La sustitución de las observaciones permite crear tantas submuestras como se desee (B), de tamaño P, que pueden analizarse de forma independiente y permiten estimar medidas robustas de error e intervalos de confianza asociados con los resultados obtenidos.

  Las submuestras de *'bootstrap'* se generalizan mediante un modelo no paramétrico:

  Sea $\hat{F}$ la distribución empleada de X_P, con un peso $\frac{1}{P}$ sobre $x_1, ..., x_P$ y sea X_P^* una muestra aleatoria de tamaño P obtenida de forma independiente e idénticamente distribuida de $\hat{F}$, donde x_i es una observación aleatoria $x_i = (x_i, t_i)$. Entonces el error verdadero se estima a partir de las distintas submuestras de entrenamiento obtenidas mediante 'bootstrap' de la siguiente manera:

$$\hat{Err}_{\text{boostrap}} =$$

$$\frac{1}{P}\sum_{i=1}^{P} \mid t_i - \hat{f}[X_P, x_i] \mid + \frac{1}{B}\sum_{b=1}^{B}\left(\frac{1}{P}\sum_{i=1}^{P} \mid t_i - \hat{f}[X^{*b}, x_i^*] \mid - \frac{1}{P}\sum_{i=1}^{P} \mid t_i^* - \hat{f}[X^{*b}, x_i^*] \mid\right)$$

La estimación del error verdadero considera, en primer lugar, el error de resustitución del modelo y, en el segundo término, la diferencia promediada entre el error medio respecto a la muestra original y el error medio respecto a las submuetras de *'bootstrap'*.

A medida que B tiende a infinito, el error estimado se aproxima al error real de generalización. No obstante, a efectos prácticos, se considera que si B varía entre 25 y 200 réplicas, los resultados obtenidos son suficientemente robustos [37].

Este método permite obtener estimaciones precisas del error verdadero, aunque requiere un esfuerzo computacional adicional debido a la necesidad de considerar B modelos diferenciados.

En el caso de problemas de clasificación, se han desarrollado dos versiones más sofisticadas de *'bootstrap'* que permiten reducir el costo computacional sin perder la robustez de los resultados.

1. ***'Bootstrap .632E'***: Esta variante combina el error de resustitución con la estimación del error de test derivado de las muestras de *'bootstrap'*:

   $$\hat{Err}_{,632} = 0{,}368 * \bar{err} - 0{,}632 * \hat{Err}_{EO}$$

   $$\hat{Err}_{,632} = 0{,}368 * \left(\frac{1}{P}\sum_{i=1}^{P} \mid t_i - \hat{f}[X_P, x_i] \mid\right) - 0{,}632 * \left(\frac{\sum_{b=1}^{B}\sum_{A_b} \mid t_i^* - \hat{f}[X^{*b}, x_i] \mid}{\sum_B \mid A_b \mid}\right)$$

   Siendo $A_b = i \mid P_i^{*b} = 0$ el número de vectores muestrales que no pertenecen a la b-ésima replica de *'bootstrap'*.

2. ***'Bootstrap .632+'***: Esta variante trata de evitar el sesgo observado del modelo *.632* para la estimación del error de generalización:

   $$\hat{Err}_{,632+} = (1 - \hat{w} * \bar{err} + \hat{w} * \hat{Err}_{EO})$$

   donde se definen:

   - $\hat{w} = \frac{0{,}632}{1-0{,}368\hat{R}}$
   - $\hat{R} = \frac{\hat{Err}_{EO} - \bar{err}}{\hat{\gamma} - \bar{err}}$
   - $\hat{\gamma} = \sum_l p_l(1 - q_l)$, siendo l el total de clases consideradas, y p_l, q_l las respectivas probabilidades a *priori* y a *posteriori* de la l-ésima clase.

   Para la definición de intervalos de confianza pueden utilizarse distintas técnicas a partir de la desviación típica de los errores ($SE_{\hat{Err}}$) y de acuerdo con la aproximación de la distribución normal estándar:

   $$\text{IntervaloDeConfianza95}\,\%_{\hat{Err}} = \hat{Err} \pm 1{,}96 * SE_{\hat{Err}}$$

CAPÍTULO 4

Red Neuronal Artificial Perceptrón

4.1. Introducción

El perceptrón multicapa es la red neuronal artificial más conocida y con mayor número de aplicaciones. Su historia comienza en 1958 cuando Rosenblatt publica los primeros trabajos sobre un modelo neuronal y su algoritmo de aprendizaje llamado 'perceptrón'.

El perceptrón está formando por una única neurona, por lo que su utilización está limitada la clasificación de patrones linealmente separables. Esta restricción es bastante problemática porque imposibilita a resolver un problema tan sencillo como la función lógica *XOR* [38].

La distinción de clases no linealmente separables se consigue a través de la aplicación del modelo 'perceptrón multicapa' que introduce una capa de neuronas entre la entrada y la salida, y utiliza como aprendizaje el *algoritmo de retropropagación* (del inglés *backpropagation*).

El desarrollo de estos conceptos es el objetivo principal de este capítulo.

4.2. Perceptrón simple

El caso más sencillo de red neuronal es el que presenta una sola neurona de cómputo. A esta estructura se le denomina *perceptrón* [18] y su estudio resulta esencial para profundizar en redes neuronales más complejas. El esquema general de este sistema se presenta en la Figura 4.1.

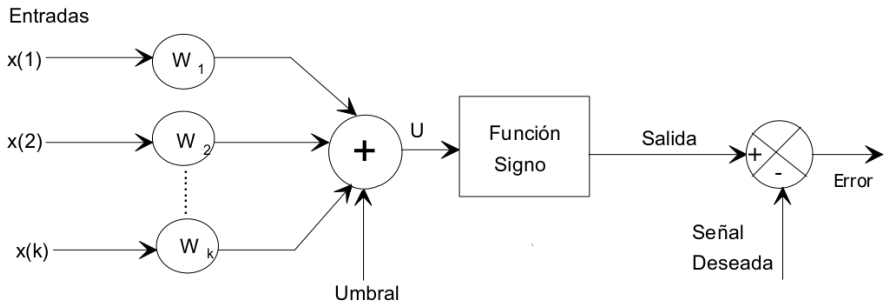


Figura 4.1: Esquema de un perceptrón.

El funcionamiento elemental de la red neuronal perceptrón se basa en comprar la salida del sistema con la señal deseada. El algoritmo debe ser supervisado ya que es necesario un agente externo que determine la clase de pertenencia de cada elemento de entrada.

4.2.1. Algoritmo de aprendizaje

El procedimiento de aprendizaje comienza con la inicialización aleatoria de los pesos de la red para ajustarlos conforme a errores detectados en la asignación de la categoría de los vectores de entrada.

El algoritmo de aprendizaje se resume en los siguientes pasos:

1. Inicializar los pesos de la red.
2. Determinar la salida de la red para la neurona x:
$$y = \sum_{k=1}^{N} (w_k * x_k)$$
3. Comparar la salida con el umbral:
$$u = y - umbral$$
4. Aplicar la función signo:
$$o = sgn(u) = \begin{cases} 1 & \text{si } u > 0 \\ 0 & \text{si } u = 0 \\ -1 & \text{si } u < 0 \end{cases}$$
5. Comparar la salida respecto a la señal deseada.
6. Actualizar los coeficientes o pesos de la red en caso de existir error:
$$w_k = w_k + \alpha * (d - o) * x_k$$
donde:
   - α es una ponderación constante tal que $0 < \alpha < 1$,
   - d denota la salida esperada.

Es posible simplificar algunas de las operaciones definidas anteriormente. Esto permite, tanto facilitar el entendimiento y la eficiencia del funcionamiento de la red perceptrón como evidenciar que esta red sólo permite la resolución de problemas linealmente separables.

En primer instancia, el umbral se puede englobar dentro de la salida del sistema mediante un peso adicional de valor 1 conectado a una entrada de la red, es decir:

$$u = y - umbral = \sum_{k=1}^{N} (w_k * x_k) - 1 * umbral = \sum_{k=0}^{N} (w_k * x_k)$$

donde:

- $w_0 = 1$, y
- $x_0 = umbral$.

Luego, considerando que el algoritmo sugiere aplicar la función signo a la expresión anterior, se deduce que la frontera de decisión se toma para $o = 0$.

Analizando en detalle un caso de dos neuronas de entradas, x_1 y x_2, y considerando $o = 0$:

$$0 = sgn(u) = w_0 + w_1 * x_1 + w_2 * x_2$$

$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2} * x_1$$

La ecuación previa es la de una recta en el espacio definido por los patrones de entrada. Por lo tanto, la superficie de separación entre categorías diferentes es una recta.

Se deduce entonces que es posible una clasificación perfecta mediante una red neuronal artificial perceptrón si los patrones de entrada son linealmente separables [39].

4.2.2. Ejemplos AND, OR y XOR

Un ejemplo sencillo de aplicación de una red perceptrón es el diseño de una puerta 'AND' de dos entradas [13].

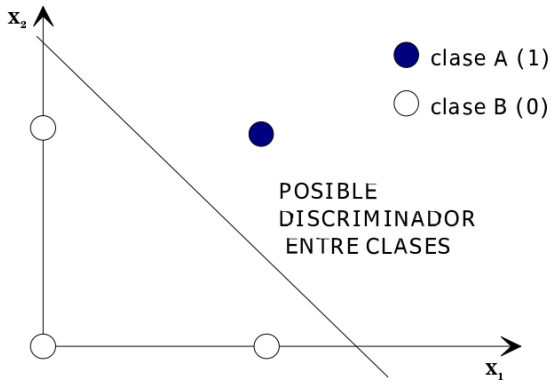


Figura 4.2: Representación gráfica de una puerta 'AND'.

Como se aprecia en la Figura 4.2, la implementación de una puerta 'AND' es un simple problema de clasificación. Una posible solución a nivel neuronal se visualiza en la Figura 4.3.

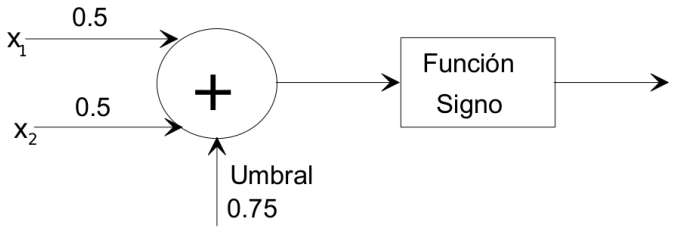


Figura 4.3: Implementación neuronal de una puerta 'AND'.

La implementación de una puerta 'OR' es análoga al caso de la puerta 'AND'. (Figura 4.4)

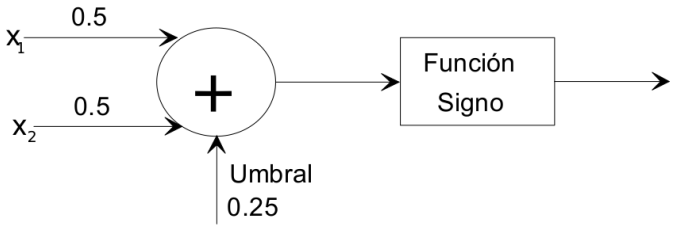


Figura 4.4: Implementación neuronal de una puerta 'OR'.

Los dos ejemplos anteriores describen problemas linealmente separables. Ahora se considera el diseño de la función lógica 'XOR':

X_1	0	1	0	1
X_2	0	0	1	1
Salida deseada	0	1	1	0

Tabla 4.1: Definición de la función lógica 'XOR'.

En la Figura 4.5 se evidencia que los patrones no son linealmente separables, es decir, no existe ninguna recta que ubique los elementos de una misma clase en un mismo lado.

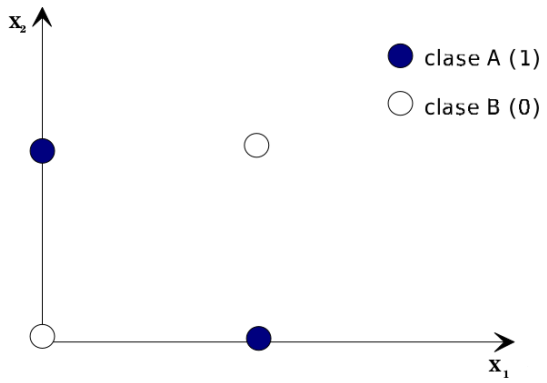


Figura 4.5: Representación gráfica de una puerta 'XOR'.

En este punto se tiene dos posibles soluciones:

- Definir otras superficies de separación: Se podría definir una elipse como delimitador de las categorías (Figura 4.6).

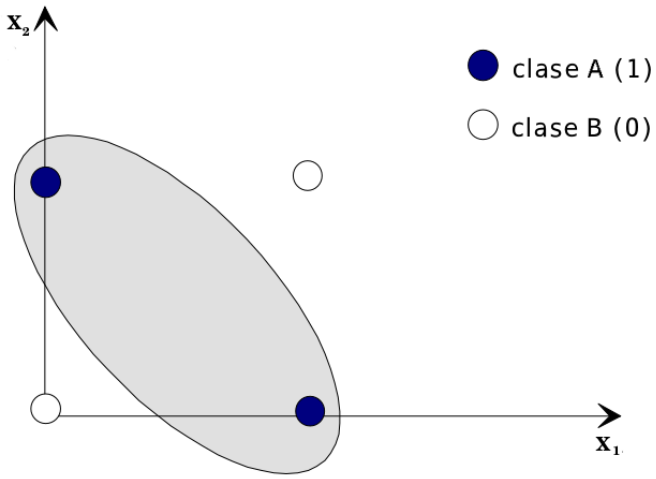


Figura 4.6: Representación gráfica de una posible solución a la puerta 'XOR'.

  Los pesos de la red debería determinarse de tal forma que las clases quedaran definidas de acuerdo a la siguiente expresión:

$$sgn(w_1 * x_1^2 + w_2 * x_2^2 + w_3 * x_1 * x_2 + w_4 * x_1 + w_5 * x_2 + w_6)$$

  Luego la superficie de separación se resuelve por la siguiente ecuación:

$$sgn(w_1 * x_1^2 + w_2 * x_2^2 + w_3 * x_1 * x_2 + w_4 * x_1 + w_5 * x_2 + w_6) = 0$$

- La segunda solución surge de la combinación de diferentes perceptrones para dar lugar a la solución de la puerta lógica 'XOR' ya que toda función lógica puede ser expresada a partir de las puertas 'AND' y 'OR'.

Para hallar soluciones a problemas de patrones no linealmente separables, el algoritmo de aprendizaje de la red perceptrón puede ocasionar oscilaciones en los valores de los pesos. Con el

objetivo de contrarrestar estas variaciones se plantean diferentes alternativas, una de ellas es el 'perceptrón multicapa' y otra el 'algoritmo de bolsillo' (del inglés *'pocket algorithm'*).

La red neuronal perceptrón multicapa es el objeto de estudio de la próxima sección de este capítulo [40].

El 'algoritmo de bolsillo' consiste en la aplicación del algoritmo perceptrón pero guardando dos vectores de pesos que coinciden con los dos mejores resultados presentados por la red. Si el vector siguiente en el algoritmo perceptrón clásico obtiene un mejor resultado que los almacenados, se reemplaza el vector guardado por el vector que supera los resultados. De esta manera, siempre se encontrará una solución, aunque no sea la óptima, evitando la inestabilidad que provoca el algoritmo perceptrón.

4.3. Perceptrón multicapa

El perceptrón multicapa propone una solución a la limitación planteada por la red neuronal perceptrón y permite arribar a resultados satisfactorios para problemas no linealmente separables. Esto se consigue introduciendo al menos una nueva capa de neuronas entre la entrada y la salida.

4.3.1. Arquitectura del perceptrón multicapa

El perceptrón multicapa es una red neuronal formada por una capa de entrada, al menos una capa oculta y una capa de salida. Su estructura se muestras en la Figura 4.7.

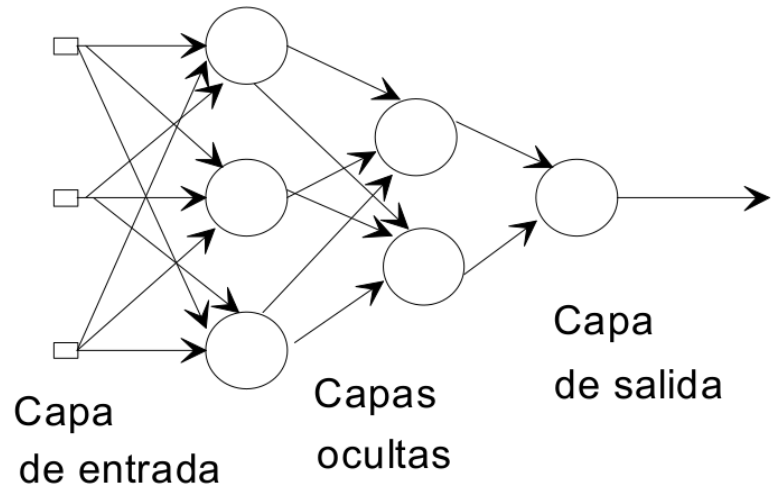


Figura 4.7: Esquema de un perceptrón multicapa.

Las características fundamentales de esta arquitectura son:

- Es una estructura altamente no lineal,
- Presenta tolerancia a fallos, y
- El sistema es capaz de establecer una relación entre dos conjuntos de datos.

En la Figura 4.7 se destaca una estructura formada por nodos o neuronas que propagan la señal hacia la salida. Las conexiones entre las neuronas se denominan pesos sinápticos, que son optimizados por el algoritmo de aprendizaje.

La propagación se realiza de manera que cada neurona hace una combinación lineal de las señales precedentes de las neuronas de la capa anterior siendo los coeficientes de esta combinación los pesos sinápticos. A continuación aplica una función de activación no lineal. (Figura 4.8)

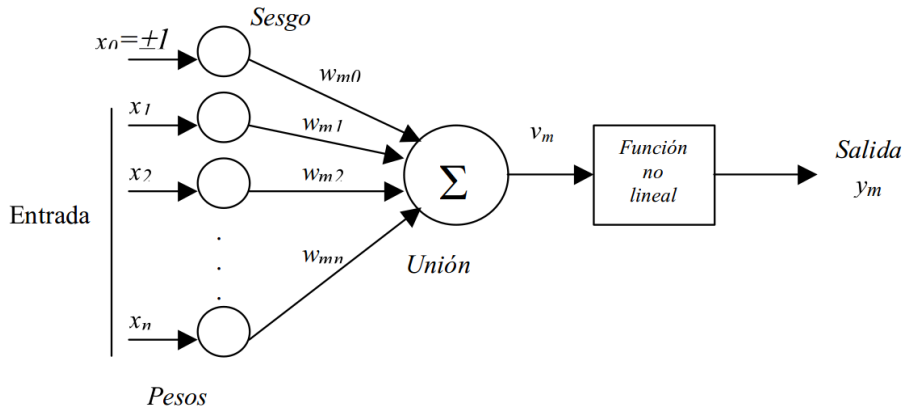


Figura 4.8: Esquema de una neurona no lineal.

Función de activación

La función no lineal que se aplica a la salida de la neurona se conoce como **'función de activación'** y debe ser continua y diferenciable.

Las tres funciones más utilizadas son:

- **Sigmoide**: Toma valores entre 0 y 1 para la variación de una variable independiente x entre $-\infty$ e ∞.

$$f_1(x) = \frac{1}{1+e^{-x}}$$

- **Tangente hiperbólica**: Toma valores entre -1 y $+1$ para la variación de una variable independiente x entre $-\infty$ e ∞. Con -1 se codifica la mínima actividad de la neurona y con $+1$ la máxima.

$$f_2(x) = \frac{1-e^{-x}}{1+e^{-x}}$$

- **Función lineal de a tramos**: Este caso trata de una función definida según tres tramos lineales que conjuntamente forman una función no lineal. Habitualmente esta función contiene valores entre -1 y $+1$ como se muestra en la siguiente ecuación:

$$f_3(x) = \begin{cases} 1 & \text{si } x > 1 \\ x & \text{si } -1 \leq x \leq 1 \\ -1 & \text{si } x < -1 \end{cases}$$

También suele considerarse una función que varíe entre 0 y 1:

$$f_3'(x) = \begin{cases} 1 & \text{si } x > 1 \\ x & \text{si } -1 \leq x \leq 1 \\ 0 & \text{si } x < -1 \end{cases}$$

Distribución de las neuronas

En cuanto a la forma de disponer las neuronas, existen gran cantidad de posibilidades. El número de neuronas que forman las capas de entrada y salida está determinado por el problema, mientras que el número de capas ocultas y de neuronas en cada una de ellas no se establece ni por el problema ni por ninguna regla teórica, razón por la cual, el diseñador es quien decide esta arquitectura en función de la aplicación que va a cumplir la red.

Únicamente está demostrado que dado un conjunto de datos conexo, con una capa oculta es posible establecer una relación entre sus elementos, aunque no se especifica el número de neuronas necesarias. Si el conjunto no es conexo, son necesarias al menos dos capas ocultas.

Existen algunos métodos empíricos para la caracterización de las capas ocultas, pero cada uno de ellos se aplica correctamente a determinados tipos de problemas. Intuitivamente resulta lógico

suponer que antes esos inconvenientes, la solución idónea es implementar una red con muchas capas ocultas y una gran cantidad de neuronas en cada una de ellas, sin embargo esto tiene una serie de inconvenientes:

- Aumento de la carga computacional: Esto implica una mayor dificultad de implementación en tiempo real y un crecimiento considerable en el tiempo de aprendizaje de la red.
- Pérdida de capacidad de generalización: Al aumentar el número de neuronas en la capa oculta, aumenta el número de pesos sinápticos, por lo que la red está conformada por más parámetros. Esto permite una mejor modelización de los patrones utilizados pero disminuye la capacidad de generalización ya que un patrón no usado en el modelo tiene más dificultades en el momento de ajustarse a un modelo con gran número de parámetros.

4.3.2. Algoritmo de aprendizaje 'Backpropagation'

Un algoritmo óptimo de aprendizaje debe cumplir las siguientes características:

- Eficiencia,
- Robustez para adaptarse a una amplia diversidad de problemas,
- Independencia respecto a las condiciones iniciales,
- Alta capacidad de generalización,
- Coste computacional bajo, y
- Sencillez en los razonamientos empleados.

Existen diferentes algoritmos de aprendizaje que optimizan las conexiones entre las neuronas en base al error cometido por la red, es decir, en base a la diferencia que existe entre la salida ofrecida por la red y la deseada.

Función de coste

Los algoritmos más utilizados son los algoritmos de descenso por el gradiente que se basan en la minimización o maximización de una determinada función. Generalmente se minimiza una función monótona creciente del error, como por ejemplo, el valor absoluto del error o el error cuadrático medio. Esta función a minimizar se denomina **'función de coste'**.

La función de coste más usada es la función cuadrática:

$$J = \frac{1}{2M} \sum_{i=1}^{M} \sum_{j=1}^{N} e_j^2(i) = \frac{1}{2M} \sum_{i=1}^{M} \sum_{j=1}^{N} (d_j(i) - y_j(i))^2$$

donde:

- M es el número de patrones utilizados para entrenar la red,
- N es el número de neuronas de la capa de salida,
- $d_j(i)$ es la salida esperada en la j-ésima neurona para el i-ésimo patrón de entrenamiento,
- $y_j(i)$ es la salida de la red en la j-ésima neurona para el i-ésimo patrón de entrenamiento.

Esta función de coste supone una distribución de errores de tipo normal, situación que generalmente está presente en problemas de modelización.

Existen también otras funciones de coste, como por ejemplo la función de coste entrópica (que supone una distribución de los errores de tipo binomial) y funciones de coste basadas en la norma de Minkowski (que posibilitan minimizar distintas funciones de error).

Aprendizaje de la red

Una vez definida la función de coste a utilizar, se debe aplicar un procedimiento de minimización de dicha función. Este proceso recibe el nombre de **'aprendizaje de la red'**.

Existen dos tipos de aprendizajes:

- **Aprendizaje 'on-line'**: Se realiza patrón a patrón. Durante todo el entrenamiento se incorpora a la red cada entrada junto con la salida deseada. Se mide el error, y en base a este se adaptan los pesos sinápticos mediante el algoritmo de aprendizaje seleccionado.
- **Aprendizaje 'off-line'**: Se realiza época a época. En este tipo de aprendizaje se provee a la red todos los patrones de entrenamiento, se evalúa el error total cometido y se adaptan los pesos en función del error total promediado según la cantidad de patrones [41].

Algoritmo

El algoritmo de aprendizaje 'backpropagation' [43] es un algoritmo de descenso por el gradiente que retropropaga las señales desde la capa de salida hasta la capa de entrada optimizando los valores de los pesos sinápticos mediante un proceso iterativo que se basa en la minimización de la función de coste.

El algoritmo puede dividirse en dos fases:

- **Propagación hacia adelante**: El vector de entrada o patrón de entrada es presentado a los nodos de la capa de entrada. Las señales se propagan desde ésta capa hasta la capa de salida, capa por capa. Se determina la salida de la red y el error cometido comparando la salida con el valor deseado o esperado.
- **Propagación hacia atrás**: En función de los errores cometidos en la capa de salida, el algoritmo se encarga de optimizar los valores de los pesos sinápticos desde la capa de salida hacia la capa de entrada, es decir, el error se retropropaga desde la última capa hacia la inicial a través de las sucesivas capas ocultas.

En la Figura 4.9 se representa la última capa oculta y la capa de salida de una red neuronal multicapa. En ella se identifican los dos tipos de señales y su propagación dentro de la red.

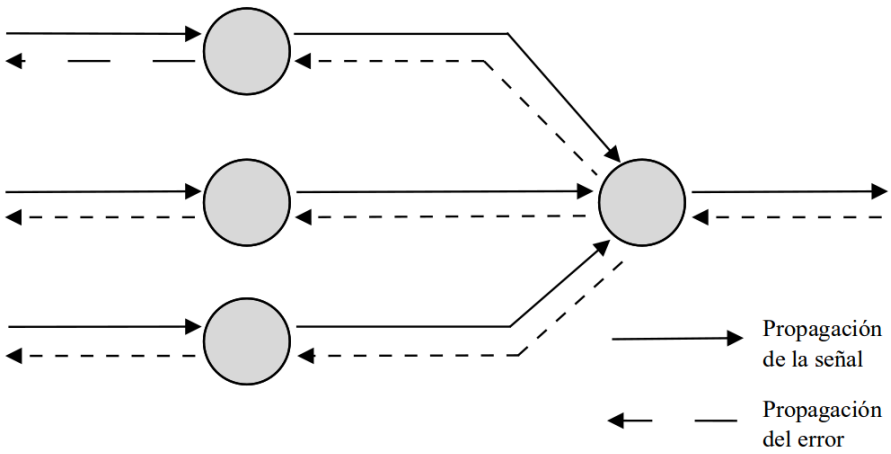


Figura 4.9: Propagación de las señales del algoritmo backpropagation.

La minimización de la función de coste se resuelve por técnicas de optimización no lineales que se basan en ajustar los parámetros siguiendo una determinada dirección. En este método, la dirección elegida es la negativa al gradiente de la función error.

Como fue mencionado anteriormente, existen dos opciones: actualizar los parámetros cada vez que se introduce un patrón de entrenamiento (*on-line*), o solamente restablecerlos cuando se hayan ingresado todos los parámetros de entrenamiento por cada época (*off-line*). De acuerdo con Haykin

[42], el algoritmo de *backpropagation* presenta un mejor desempeño de entrenamiento actualizando los pesos por medio del método (*on-line*).

Para este modo de operación los ciclos del algoritmo, dado el conjunto de entrenamiento $\{x(m), d(m)\}_{m=1}^{M}$ (donde M es la cantidad de patrones de entrada), se pueden resumir en las siguientes cinco fases:

1. **Inicialización**: Asumiendo que no existe información a priori disponible, se establecen pesos sinápticos con valores aleatorios entre 0 y 1, o entre -1 y 1.

2. **Presentación de las muestras de entrenamiento**: Se presenta a la red neuronal el conjunto de vectores o patrones de entrenamiento. Por cada vector de entrenamiento, siguiendo un orden específico, se realiza el cálculo de propagación hacia adelante y hacia atrás como se muestra en las fases tres y cuatro respectivamente.

3. **Cálculo de propagación hacia adelante**: Dado el conjunto de entrenamiento $\{x(m), d(m)\}$, donde $x(m)$ es el vector de entrada o vector de entrenamiento y $d(m)$ es la salida deseada de la red, se calculan los valores de propagación y transferencia de la red capa por capa con dirección hacia adelante. La función de propagación o suma ponderada de las entradas $v_j^{(l)}(m)$ para la neurona j de la capa l es definida como:

$$v_j^{(l)}(m) = \sum_{i=0}^{P} w_{ji}^{(l)}(m) * y_i^{(l-1)}(m)$$

donde:

- P es el número de neuronas de la capa anterior $(l-1)$,
- $y_i^{(l-1)}(m)$ es la señal de salida de la neurona i en la capa $l-1$ para el vector de entrenamiento m, y
- $w_{ji}^{(l)}(m)$ es el peso sináptico que conecta las neuronas j de la capa l e i de la capa $l-1$.

Para $i = 0$ se define:

- $y_0^{(l-1)}(m) = 1$, y
- $w_{j0}^{(l)}(m) = b_j^{(l)}$, donde éste último es el umbral aplicado a la neurona j de la capa l.

Asumiendo el uso de una función sigmoidal como función de transferencia, la señal de salida de la neurona j en la capa l está determinada por:

$$y_j^{(l)}(m) = \frac{1}{1 + e^{-v_j^{(l)}(m)}}$$

Si la neurona j está en la primer capa de la red, entonces:

$$y_j^{(0)}(m) = x_j(m)$$

donde:

- $x_j(m)$ es el j-ésimo elemento del vector de entrada $x(m)$.

Por otra parte, si la neurona j está en la capa de salida, entonces:

$$y_j^{(L)}(m) = o_j(m)$$

donde:

- $o_j(m)$ es el j-ésimo elemento del vector de salida proporcionado por la red.

Con ello, se puede calcular el error entre el vector deseado y el vector obtenido como:

$$e_j(m) = d_j(m) - o_j(m)$$

donde:

- $d_j(m)$ es el j-ésimo elemento del vector de respuestas deseadas $d(m)$.

4. **Cálculo de propagación hacia atrás**: En esta fase se realiza el cálculo de los gradientes de la red y se calcula capa por capa. Si la neurona j está en la capa de salida L, el gradiente se define como:
$$\delta_j^{(L)}(m) = e_j^{(L)}(m) * o_j(m) * [1 - o_j(m)]$$
En otro caso, si la neurona j está en la capa de oculta l, el gradiente se calculado en base a la siguiente expresión:
$$\delta_j^{(l)}(m) = y_j^{(L)}(m) * [1 - y_j^{(l)}(m)] * \sum_{k=1}^{P} \delta_k^{(l+1)}(m) * w_{kj}^{(l+1)}(m)$$
donde:
   - P es el número de neuronas de la capa $l + 1$.

   El ajuste de los pesos sinápticos de la red en la capa l se realiza de acuerdo a la siguiente regla:
$$w_{ji}^{(l)}(m+1) = w_{ji}^{(l)}(m) + \eta * \delta_j^{(l)}(m) * y_j^{(l-1)}(m)$$
donde:
   - η es el índice de aprendiza je de la red neuronal.

5. **Iteraciones**: Las iteraciones se realizan siguiendo las fases tres y cuatro con nuevos patrones de entrenamiento hasta que los parámetros libres o pesos sinápticos se estabilicen en un punto donde la función de coste alcanza un valor aceptable (error máximo permitido), generalmente definido por el investigador.

Las cinco fases descriptas realizan un entrenamiento *on-line* de la red neuronal, ya que por cada patrón de entrenamiento se actualizan todos los pesos sinápticos. No obstante, la generalización a entrenamiento tipo *off-line* es prácticamente inmediata.

En la Figura 4.10 se muestran las variables involucradas en el algoritmo 'backpropagation' y su relación con el resto de los elementos de la red neuronal.

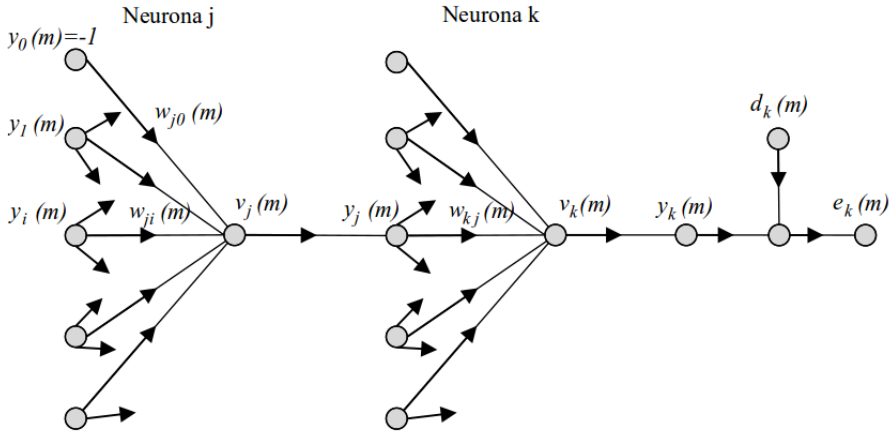


Figura 4.10: Variables involucradas en el algoritmo backpropagation.

4.3.3. Variantes del algoritmo 'Backpropagation'

El algoritmo de aprendizaje *'backpropagation'* es el más conocido para el entrenamiento de la red neuronal perceptrón multicapas. Es un algoritmo de máximo descenso que busca minimizar la función de coste J dada por un error $e_j(i)$. Sin embargo, en la práctica su uso se ve muy limitado debido a algunos inconvenientes que presenta.

- Baja velocidad de convergencia: Se observa cuando la naturaleza del error es no lineal y sólo se encuentra disponible la información del gradiente. El coeficiente de aprendizaje debe mantenerse bastante pequeño para asegurar una convergencia estable. Como consecuencia, se incrementa el tiempo del proceso de aprendizaje.
- Mínimos locales: El parámetro de error en algún momento puede presentar mínimos locales, los cuales provocan que el algoritmo de aprendizaje, por ser descendiente, llegue a detenerse.
- Oscilaciones: En un caso extremo, donde el error presenta un comportamiento en forma de 'onda' inclinándose suavemente hacia el mínimo real, el algoritmo de máximo descenso puede entrar en una situación sin convergencia, variando continuamente sin ir progresivamente a un mínimo.
- La variación de los pesos: Dado un coeficiente de aprendizaje fijo, la variación de los pesos, es directamente proporcional a la magnitud del gradiente. Entonces, este cambio es abrupto cuando la pendiente del error es grande y pequeño cuando la pendiente es chica. Cuando el error presenta altiplanicies y declives que cambian rápidamente, un coeficiente de aprendizaje fijo conlleva a un bajo rendimiento en el entrenamiento de la red.

Debido a las limitaciones planteadas previamente, se han introducido mejoras del algoritmo para acelerar la convergencia del mismo. A continuación se describen brevemente dos de las más importantes, las cuales se pueden considerar como modificaciones heurísticas de 'backpropagation' y técnicas de optimización numérica:

- **Momento**: La regla delta generalizada se basa en la búsqueda del mínimo de la función error mediante el descenso por el gradiente de la misma. Esto puede llevar a un mínimo local de la función error, donde el gradiente vale cero, y por lo tanto los pesos no se ven modificados pero el error cometido por la red es significativo. Esta variante es muy similar al *'backpropagation'* clásico ya que el incremento de los pesos es igual al gradiente de la función de error con signo negativo, y además añade un término que es el incremento de pesos anteriores:

  $$w_{ji}^{(l)}(m+1) = w_{ji}^{(l)}(m) - \eta * \delta_j^{(l)}(m) * y_j^{(l-1)}(m) + \alpha[w_{ji}^{(l)}(m-1)]$$

  donde:

  - α es la constante de **momento**, que puede tomar valores en el intervalo $(0, 1)$.

  La constante de momento es la encargada del nuevo incremento en el valor de w_{ji} en relación al incremento previo del mismo peso. Este nuevo término controla la velocidad de acercamiento al mínimo, acelerándola cuando está lejos y disminuyéndola cuando está cerca.

- **Razón de aprendizaje variable (η)**: Ésta juega un papel muy importante en el comportamiento de los algoritmos de aprendizaje. Si es pequeña, la magnitud del cambio de los pesos sinápticos será pequeña y por lo tanto tardará mucho en converger. Si es demasiado grande el algoritmo oscilará y difícilmente encontrará un mínimo de la función error.

  En algunos casos se ha demostrado que el valor óptimo de la tasa de aprendizaje, para una convergencia rápida sea el valor inverso del mayor autovalor de la matriz Hessiana H. Computacionalmente este proceso es ineficiente dado que para obtener esta matriz es necesario evaluar las derivadas segundas de la función de error o función de coste.

  Por ello, se emplean técnicas heurísticas que varían el valor de la tasa de aprendizaje en cada iteración. Algunas de éstas son:

- Incrementar la tasa de aprendizaje cuando el gradiente $\delta_j(i-1)$ es próximo al gradiente $\delta_j(i)$, así como disminuirla en caso contrario.
- Multiplicar la tasa de aprendizaje por un valor mayor a 1 si los gradientes actual y previo tienen el mismo signo, o por un valor entre 0 y 1 en caso contrario.
- Multiplicar la tasa de aprendizaje por una cantidad mayor a 1 cuando haya decrecido la función error (con el fin de avanzar más rápido), y por una cantidad menor que 1 en caso contrario.

4.3.4. Selección de parámetros

En el algoritmo *'backpropagation'*, uno o más parámetros necesitan ser definidos por el usuario o investigador. La elección de valores para estos parámetros puede tener un efecto muy significativo en el rendimiento de la red neuronal.

Cantidad de capas

Kavzoglu [44] notó que el número de neuronas de las capas intermedias de una red perceptrón multicapas tiene una influencia significativa en la habilidad de la red de generalizar a través de los datos de entrenamiento. Se piensa que la utilización de una sola capa oculta es adecuada para la mayoría de los problemas de clasificación, pero cuando son numerosas las clases de salida se aconseja emplear al menos dos capas ocultas para producir un resultado más exacto.

Kanellopoulos y Wilkinson [45] sugirieron que cuando hay 20 o más clases, dos capas intermedias deben ser utilizadas, y el número de neuronas de las segunda capa oculta debe ser igual a dos o tres veces la cantidad de clases de salida.

Garson [46] propuso que el número de capas ocultas se establezca de acuerdo al siguiente valor:

$$N_P(r(N_i + N_o))$$

donde:

- N_P es el número de muestras de entrenamiento,
- N_i es el número de características de los datos de entrada,
- N_o es el número de clases de salida, y
- El parámetro r está relacionado con el ruido presente en los datos de entrada y con la simplicidad de la clasificación. Los valores típicos de r se encuentran comprendidos entre 5 (escaso ruido) y 10 (mayor cantidad de ruido). Sin embargo, también es posibles trabajar con valores menores a 2 y mayores que 100.

Tasa de aprendizaje y coeficiente del término momento

Si se utiliza la ecuación propuesta en la sección anterior para actualizar los pesos de las conexiones de la red, es necesario establecer el valor de los parámetros η y α.

El valor del parámetro η no debe ser demasiado grande, a pesar de lograrse con esto una minimización más pronunciada, ya que el resultado de la clasificación será demasiado 'pobre'. Tampoco debe ser lo suficientemente pequeño, ya que la fase de entrenamiento consumirá gran tiempo computacional.

La selección del valor de η depende de las características de la clasificación. En base a grandes conjuntos de entrenamientos, Kavzoglu [44] recomendó usar $\eta = 2$ cuando no se agregue el término 'momento'. En caso contrario, el valor de la tasa de aprendizaje debe elegirse en el rango $[0.1, 0.2]$ si el coeficiente del término momento (α) pertenece al intervalo $[0.5, 0.6]$.

También es posible variar el valor de estos parámetros durante el proceso de entrenamiento como fue detallado previamente.

Pesos iniciales

Otra consideración a tener en cuenta es la selección de los valores individuales de la fuerza de interconexión entre las neuronas, ya que ésta puede tener un efecto significativo en la eficiencia de la red. Estos valores son asignados aleatoriamente dentro de un rango específico.

Ardö [47] descubrió que la precisión varía entre un 59 % y 70 % utilizando un conjunto de pesos iniciales seleccionados en el rango $[-1, 1]$.

Kavzoglu [44] notó que la eficiencia de la clasificación mejoraba para pequeños rangos de pesos iniciales, y recomendó que los valores de los pesos deben pertenecer al intervalo $[-0.25, 0.25]$.

Cantidad de iteraciones en el entrenamiento

El entrenamiento utilizando el algoritmo de propagación hacia atrás es un proceso iterativo. Si la red es entrenada con muchas iteraciones, esta 'aprenderá' características muy específicas de los datos de entrenamiento y fallará al intentar reconocer algunos datos genéricos no ingresados durante el aprendizaje.

Por otra parte, si la red no es suficientemente entrenada, la posición de los bordes de clasificación será una tarea difícil de realizar.

Puede utilizarse un enfoque de 'validación cruzada', en el cual el conjunto de datos de entrenamiento es subdividido en validaciones y subconjuntos de entrenamiento. La red 'aprende' a partir de un subconjunto de entrenamiento y se detiene en un determinado punto del entrenamiento. En este momento, la red clasifica las muestras contenidas en el subconjunto de validación. El entrenamiento continúa hasta que el error de validación de clasificación empieza a elevarse.

Codificación de vectores de entrada y salida

Es necesario determinar cuidadosamente la codificación o representación de los vectores de entrada y de los vectores de salida.

Para los vectores de entrada, cada componente puede ser normalizado en un intervalo $[0, 1]$. Se cree que con la normalización es posible reducir el efecto adverso del ruido.

Para los vectores de salida, muchos esquemas de codificación pueden ser aplicados, como por ejemplo, el enfoque binario. Tomando en cuenta, una clasificación de datos en 4 tipos distintos, sólo se necesitan dos neuronas en la capa de salida si se utiliza la codificación binaria, etiquetando las clases 1, 2, 3 y 4 de la siguiente manera 00, 01, 10, 11.

Otro enfoque empleado es el de propagación. Este tiene la ventaja de que el resultado de la clasificación es directamente mapeado en el vector de salida. Por lo tanto es ampliamente utilizado.

4.3.5. Ejemplo de decisión de bordes: XOR

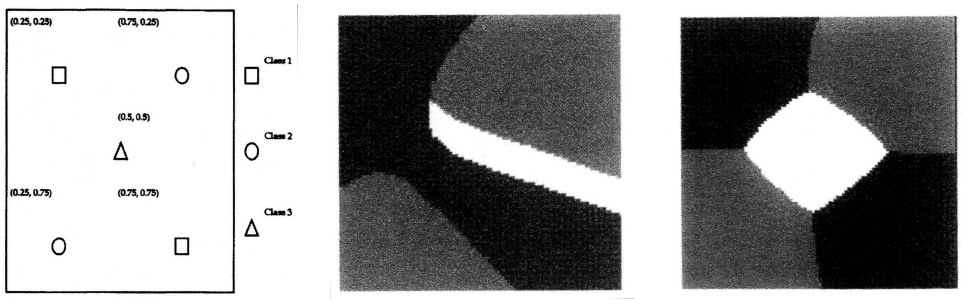
Un ejemplo clásico que muestra la relación entre la estructura de la red perceptrón multicapa y la decisión de la ubicación y bordes del espacio de clasificación, es el problema de clasificar un simple operador 'XOR' [48].

El problema 'XOR' se puede resolver únicamente en redes que contengan neuronas en capas ocultas. Más aún, tamaños diferentes de redes neuronales producirán distintas decisiones de bordes de clasificación: las grandes redes tienen el potencial de crear decisiones de bordes más complejas que las pequeñas.

Para demostrar lo anterior, se utiliza una variación del problema 'XOR' que clasifica los datos en 3 tipos distintos. Los datos están distribuidos en un espacio bidimensional particionado en 100×100 subcuadrados en el intervalo $[0, 1]$. Sólo se toman 5 patrones de entrenamiento cuya ubicación se expone en la Figura 4.11.a.

Dos redes son utilizadas para este experimento. Una tiene estructura 2|3|3 con 15 pesos de interconexión ($2 \times 3 = 6$ pesos conectando la capa de entrada con la intermedia, y $3 \times 3 = 9$ pesos conectando la capa intermedia con la de salida). La otra estructura es 2|40|60|3, con 2600 pesos. El vector de salida es codificado utilizando el enfoque de propagación.

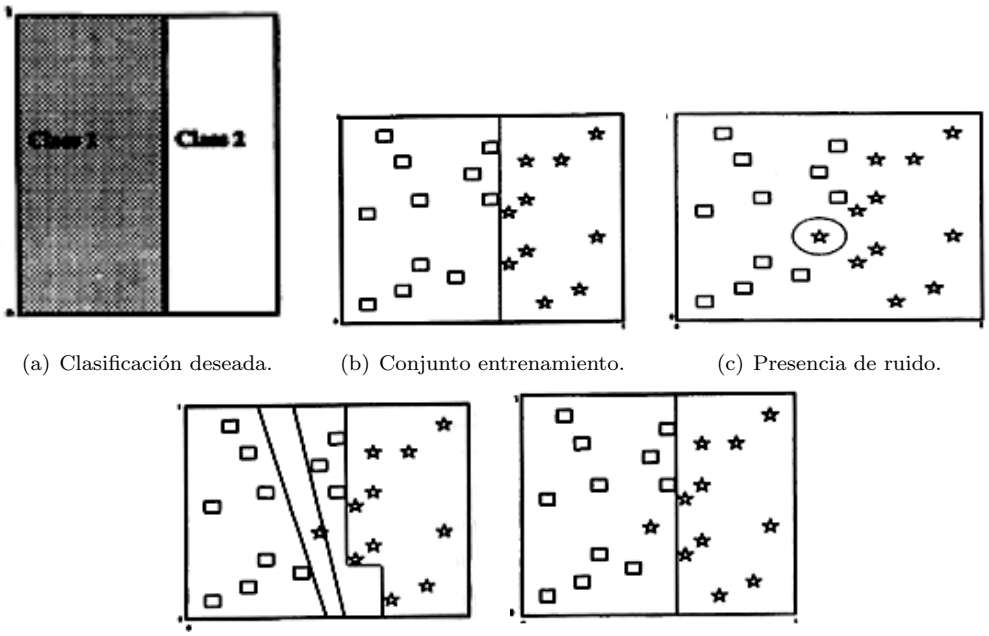
Los resultado de las clasificaciones con las distintas redes propuestas se muestran en la Figura 4.11.b y en la Figura 4.11.c respectivamente.



(a) Conjunto entrenamiento. (b) Red perceptrón 2|3|3. (c) Red perceptrón 2|40|60|3.

Figura 4.11: Ejemplo de clasificación con red perceptrón multicapas de una variación de XOR.

Como se observar, la decisión de bordes obtenida por la red más extensa es más satisfactoria que la producida por la red más chica. Esto lleva a preguntarse si redes neuronales más complejas clasifican mejor que las sencillas. La respuesta es negativa. Esto se puede probar con el siguiente ejemplo de la Figura 4.12.



(a) Clasificación deseada. (b) Conjunto entrenamiento. (c) Presencia de ruido.

(d) Red perceptrón 2|60|10|2. (e) Red perceptrón 2|3|2|2.

Figura 4.12: Ejemplo de clasificación con red perceptrón multicapas de un problema con ruido.

En la Figura 4.12.a se representa la clasificación deseada. El dominio utilizado en este problema es $[0, 1]$ en un espacio bidimensional particionado en $100x100$ subcuadrados iguales.

En la Figura 4.12.b se observa la ubicación de los patrones de entrenamiento.

En la Figura 4.12.c se agrega un parámetro de la 'clase estrella' en el espacio de la otra clase con el fin de testear la eficiencia de una red ante la presencia de ruido.

Las 22 muestras que se observan en la Figura 4.12.c se utilizan para entrenar dos redes. La primera red tiene una estructura 2|60|10|2 con 740 aristas, mientras que la segunda tiene una estructura 2|3|2|2 con sólo 16 conexiones. La Figura 4.12.d y la Figura 4.12.e muestran la decisión de bordes formada por cada una de las redes anteriores. La decisión de la Figura 4.12.e es la mejor.

Por lo tanto, se puede concluir que la estructura de red adecuada para lograr una buena clasificación depende del caso a analizar.

CAPÍTULO 5

Red neuronal artificial de Kohonen

5.1. Introducción

La idea de un mapa principal de auto organización, también llamado *SOM* (del inglés, *'Self Organizing Map'*) fue desarrollada por Teuvo Kohonen en la década de 1980 [49, 50]. Se basa en ciertas evidencias descubiertas a nivel cerebral.

Contrariamente a las redes perceptrón multicapas, estas no contienen capas intermedias, sólo la capa de entrada y la de salida. Tienen una propiedad de interés: detectan automáticamente relaciones dentro del conjunto de patrones de entrada a través de un aprendizaje no supervisado.

La red de auto organización descubre rasgos comunes, regularidades, correlaciones o categorías en los datos de entrada, y los incorpora a su estructura interna de conexiones. Se dice entonces, que las neuronas se auto organizan en función de los estímulos procedentes del exterior.

Para realizar esta tarea se emplea la técnica de 'aprendizaje competitivo', donde cada neurona de la capa de salida disputa con las otras la posibilidad de poseer mayor similitud al impulso recibido. Así, cuando se presenta un patrón de entrada, sólo la neurona vencedora (o la neurona vencedora y sus vecinas) se activa, quedando el resto de las neuronas anuladas.

El objetivo de este aprendizaje es categorizar los datos que se introducen en la red. Se clasifican los estímulos similares dentro de la misma categoría; por lo tanto, activan la misma neurona de salida.

5.2. Aprendizaje competitivo

El aprendizaje competitivo es un tipo de aprendizaje no supervisado que sirve de base para varios modelos de redes neuronales artificiales. El objetivo de las redes basadas en este aprendizaje es llevar a cabo una categorización de los datos de entrada. Impulsos parecidos deben ser clasificados como pertenecientes a una misma clase mediante un proceso de búsqueda de categorías que la red lleva a cabo de forma independiente.

La arquitectura básica de estas redes consiste en dos capas:

- La capa de entrada: Recibe los estímulos procedentes del entorno, y
- La capa de competición: Es la que produce la salida de la red.

Cada neurona de la capa de entrada está conectada con todas las neuronas de la capa de competición a través de pesos sinápticos adaptativos, es decir, que modifican su valor en base a una regla de aprendizaje definida.

Las neuronas de la capa de competición, además de recibir los datos ponderados procedente de las neuronas de la capa de entrada, tienen conexiones laterales inhibitorias con el resto de las neuronas de la capa y una conexión excitatoria consigo misma. (Figura 5.1)

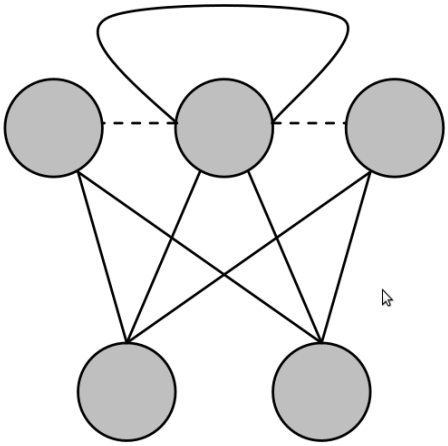


Figura 5.1: Esquema de estructura competitiva.

Las conexiones existentes entre las neuronas de la capa de competición son fijas. Permiten que la neurona con mayor valor de excitación se refuerce más a sí misma (vía autoconexión excitatoria) e inhiba con mayor fuerza a las demás neuronas de la capa.

Esta dinámica conduce a un proceso competitivo en el que todas las neuronas intentan aumentar su excitación al mismo tiempo que tratan de reducir la activación de las restantes. El proceso continúa hasta que la red se estabiliza. En ese momento existe una neurona ganadora que se considera la salida deseada.

En resumen, el algoritmo de aprendizaje se puede sintetizar en los siguientes pasos:

1. Se presenta un estímulo a la capa de entrada de la red.
2. La señal se propaga hasta la capa de competición a través de las conexiones adaptativas procedentes de la capa de entrada.
3. Se calcula el valor de las excitaciones de cada una de las neuronas de la capa de competición.
4. Se efectúa el proceso de competición mediante la inhibición lateral y el autorefuerzo.
5. Finalmente se modifican los pesos adaptativos asociados a la neurona vencedora.

Tras la competición, la neurona ganadora es la que mejor relaciona con el estímulo de entrada. El aprendizaje pretende reforzar la correspondencia modificando las conexiones entre la capa de entrada con la neurona vencedora. Esto produce que para esta neurona sea más sencillo reconocer el mismo estímulo (o estímulos parecidos) en las siguientes presentaciones o iteraciones.

5.3. Descripción general de los mapas de auto-organizativos

El modelo de red neuronal auto-organizativa *SOM* toma como base el agrupamiento de tareas similares que tiene lugar en las diferentes zonas del cerebro. Es un modelo competitivo donde la capa de entrada o sensorial consiste en m neuronas, una por cada variable de entrada. El procesamiento se realiza en la capa de competición, donde se forma un mapa de rasgos. (Figura 5.2)

Cada una de las neuronas de entrada está conectada con todas las neuronas de la segunda capa mediante pesos sinápticos. La capa de entrada tiene la misma dimensión que el dato de entrada, y la actividad de las neuronas de esta capa es proporcional a dicho patrón. A su vez, a toda neurona de la segunda capa se le asigna un vector de pesos que tiene la misma dimensión que los vectores de entrada, es decir, para la neurona 'ij' de la capa de salida se tiene:

$$w_{ij} = [w_{ij}^1, w_{ij}^2, ..., w_{ij}^k]$$

donde k es la longitud del vector de entrada.

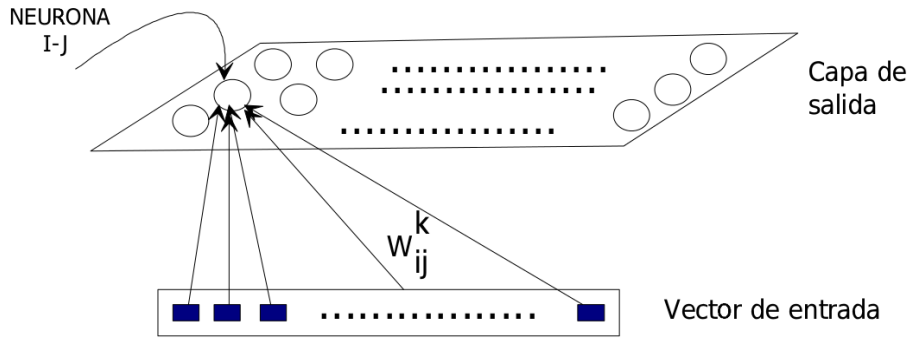


Figura 5.2: Arquitectura de un mapa auto-organizativo.

El objetivo de esta red neuronal es que patrones similares de entrada aumenten la actividad de neuronas próximas en la capa de salida del mapa auto-organizativo. Una manera de lograr esto es estableciendo una función de semejanza entre el vector de entrada y los vectores de los pesos asociados a cada una de las neuronas de salida. Existen diferentes formas de medir el grado de similitud entre ambos. Los métodos más utilizados son la función de distancia cuadrática, la función de distancia de Manhattan y la función de distancia de Minkowski (Tabla 5.1).

Función de distancia	**Expresión matemática**
Cuadrática	$\left(\sum_{s=1}^{k}(w_{ij}^{s}-x^{s})^{2}\right)^{\frac{1}{2}}$
Manhattan	$\sum_{s=1}^{k} \mid w_{ij}^{s}-x^{s} \mid$
Minkowski	$\left(\sum_{s=1}^{k}(w_{ij}^{s}-x^{s})^{\lambda}\right)^{\frac{1}{\lambda}}$

Tabla 5.1: Diferentes métricas para comparar vectores.

Generalmente se emplea una variantes de la función de distancia cuadrática: la función 'norma cuadrática'.

Si dos vectores tiene la misma norma, la semejanza entre ellos puede ser definida por el ángulo que forman entre sí. Esta conclusión se obtiene desarrollando la expresión de la distancia cuadrática como producto escalar de un vector consigo mismo y aplicando la normalización de los vectores y la definición escalar:

$$\left(\sum_{s=1}^{k}(w_{ij}^{s}-x^{s})^{2}\right)^{\frac{1}{2}} = ((w_{ij}-x)\cdot(w_{ij}-x))^{\frac{1}{2}} = \left(\left\|w_{ij}\right\|^{2}+\left\|x\right\|^{2}-2\cdot w_{ij}\cdot x\right)^{\frac{1}{2}}$$

$$\left(\sum_{s=1}^{k}(w_{ij}^{s}-x^{s})^{2}\right)^{\frac{1}{2}} = (2-2*\cos(\theta))^{\frac{1}{2}}$$

donde $\cos(\theta)$ es el ángulo que forman los dos vectores.

Luego, minimizar la distancia entre dos vectores con la misma norma es equivalente a maximizar el coseno del ángulo entre los vectores.

Una vez determinado el criterio de semejanza del sistema, es importarte tratar el aprendizaje de la red, es decir, el mecanismos de actualización de sus pesos.

En un principio se supone un desconocimiento del problema, por lo que los pesos se inicializan aleatoriamente. Después se ingresa en la red un vector de entrada y se aplica el criterio de similitud escogido a este vector y a cada uno de los vectores de la capa de salida. En dicha capa se lleva a

cabo el proceso de competición entre las distintas neuronas. La vencedora es aquella cuyo vector de peso posee mayor grado de semejanza con el vector de entrada.

Como el objetivo es que este vector sea el vencedor si el patrón de entrada aparece nuevamente, se debe modificar el vector de pesos de la neurona ganadora para que se parezca más al vector de entrada. La forma de efectuar esto, cuando se usa como función de similitud la distancia cuadrática, es aplicando la siguiente actualización de los pesos sinápticos:

$$w_{ks} = w_{ks} + \beta \cdot (x - w_{ks})$$

donde:

- ks es la neurona vencedora, y
- β es una valor de ponderación.

Si este proceso se repite con numerosos estímulos, la red auto-organizativa especializa cada neurona de la capa de salida en la representación de una de las clases a la que pertenece la información de entrada. Sin embargo, esto no garantiza que los representantes de clases parecidas se dispongan conjuntamente en la capa de salida. Para ello, el mapa de Kohonen incorpora una interacción lateral entre las neuronas adyacentes con el fin de conseguir que neuronas próximas en la capas de salida representen patrones similares de entrada. El alcance de esta interacción es consecuencia de la definición de una 'función de vecindad'.

La *función de vecindad* determina un grupo de neuronas próximas a la neurona ganadora en el proceso de competición y la intensidad con que éstas deben modificar sus pesos sinápticos. Por lo tanto, el cambio en los pesos sinápticos no sólo se aplica a la neurona vencedora sino también a aquellas que comprenda el alcance de las interacciones laterales definidas por la función de vecindad.

De este modo se consigue que el mapeo generado cumpla con el objetivo de que patrones similares en la entrada se correspondan con la activación de neuronas próximas en la capa de salida.

5.4. Algoritmo de aprendizaje

El algoritmo de entrenamiento del mapa auto-organizativo de Kohonen se resume en los pasos que se numeran a continuación:

1. Inicializar los pesos de la red.
2. Presentar un patrón de entrada $x(t)$.
3. Determinar la similitud entre los pesos de cada neurona de salida y la de entrada.

   Si se considera la distancia euclídea como medida de comparación:

   $$d(w_{ij}, x) = \sum_{s=1}^{k} (w_{ij}^s - x^s)^2$$

   donde:

   - ij es cada una de las neuronas de la capa de competición.
4. Determinar la neurona ganadora, es decir, aquella con menor distancia al vector de entrada.
5. Actualizar los pesos sinápticos.

   Si se utilizar la función cuadrática como función de similitud:

   $$w_{ij}(t+1) = w_{ij}(t) + \alpha(t)h(t)(x - w_{ij})$$

   donde:

   - $\alpha(t)$ es la velocidad de aprendizaje en el tiempo t, y

- $h(t)$ es la función de vecindad con pico de amplitud en la neurona ganadora en el tiempo t.

6. Si se alcanza el número máximo de iteraciones (debidamente estableciendo ante de comenzar el algoritmo), terminar la ejecución. En caso contrario, volver al Paso 2.

5.4.1. Métodos implicados

Para poder concretar el algoritmo de aprendizaje descripto es necesario definir los siguientes métodos:

- La velocidad de aprendizaje, y
- La función de vecindad.

Velocidad de aprendizaje

Como su nombre lo indica, la velocidad o tasa de aprendizaje fija la velocidad de cambio de los pesos de la red. Esto puede ser constante o dependiente del número de iteraciones, pero se recomienda que sea una función decreciente en el tiempo, es decir, que reduzca su magnitud conforme se incrementa el número de ciclos de entrenamiento.

La elección más usual es considerar la siguiente variación exponencial:

$$\alpha^t = \alpha_{max} \left(\frac{\alpha_{min}}{\alpha_{max}} \right)^{\frac{t}{t_{max}}}$$

donde:

- $1 \geq \alpha$,
- $\alpha_{max}, \alpha_{min} \geq 0$,
- t representa a la iteración actual, y
- t_{max} es el número máximo de iteraciones.

Función de vecindad

La función de vecindad tiene una forma definida pero su relación suele variar con respecto al tiempo de tal manera que inicialmente es un radio grande, con el fin de obtener una ordenación global del mapa, y se reduce hasta finalmente actualizar sólo los pesos de la neurona ganadora y de aquellas muy próximas.

Las dos funciones de vecindad más empleadas son la 'función rectangular' y la 'función Gaussiana'. (Figura 5.3)

Como se ha mencionado, el número de vecinos que se actualizan decrece conforme avanza el aprendizaje. A modo de ejemplo, para la función Gaussiana, el rango de vecindad centrado en la neurona vencedora ks se calcula de la siguiente manera:

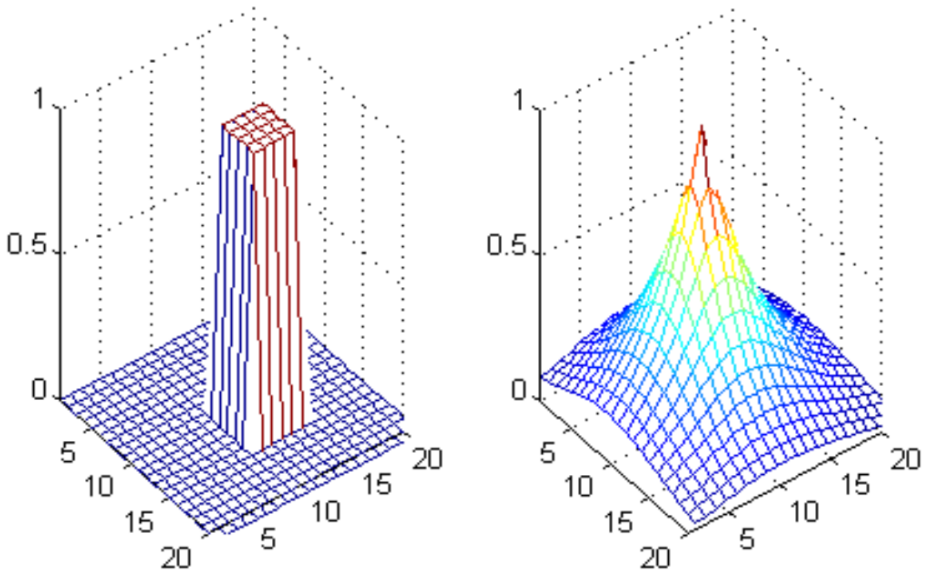
$$h_{ks'} = exp \left(\frac{- \left\| ks - ks' \right\|^2}{2\sigma^2} \right)$$

donde:

- ks' denota los vecinos de ks incluido ks mismo.
- El término σ es una función decreciente a medida que aumenta el número de iteraciones. Se define generalmente como:

$$\sigma(t) = \sigma_{max} \left(\frac{\sigma_{min}}{\sigma_{max}} \right)^{\frac{t}{t_{max}}}$$

donde:



(a) Función rectangular. (b) Función Gaussiana.

Figura 5.3: Funciones de vecindad más utilizadas[14].

- t es es el índice temporal o número de iteraciones,
- σ_{max} es el valor inicial de σ y debe ser lo suficientemente grande para que $h(t)$ cubra toda la capa de salida en la primera iteración, y
- σ_{min} es el valor final de σ. Generalmente se elige cercano a 1.

Es preciso observar que la ecuación $h_{ks'}$ indica que para la neurona ganadora ($ks' = ks$) la magnitud actualizada para el peso w_{ks} es proporcional a la taza de aprendizaje α ya que $h_{ks'} = h_{ks} = 1$.

La modificación de los pesos de aquellas neuronas cercanas a la neurona vencedora comienza abarcando toda la red y decrece en tamaño en las sucesivas iteraciones. Es decir, decrece la cantidad de ks's que forman parte de la vecindad de la neurona ganadora. Esto se ilustra en la Figura 5.4.

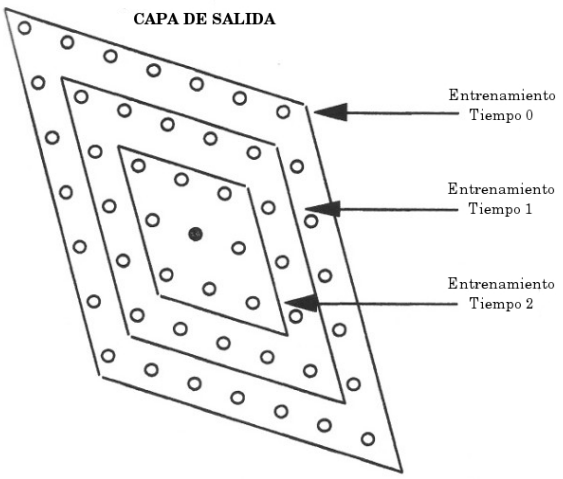


Figura 5.4: Reducción de la vecindad conforme avanza el algoritmo[48].

Selección de parámetros

Diferentes valores en los parámetros α y σ producen distintos efectos en el mecanismo de aprendizaje. Muchos estudios han investigado la diversidad de consecuencias ocasionadas al variar estos parámetros. La conclusión general es que si se elije un valor adecuado σ_{max}, tal que la función de vecindad cubra toda la corteza de mapeo inicialmente, es probable que la red *SOM* termine en un estado bien ordenado. También se observa que la tasa de aprendizaje α debe ser grande (en el orden de 1) en el comienzo del entrenamiento y debe decrecer durante la continuación del proceso. Finalizado el entrenamiento, el valor de α debe ser tan chico como 0,01.

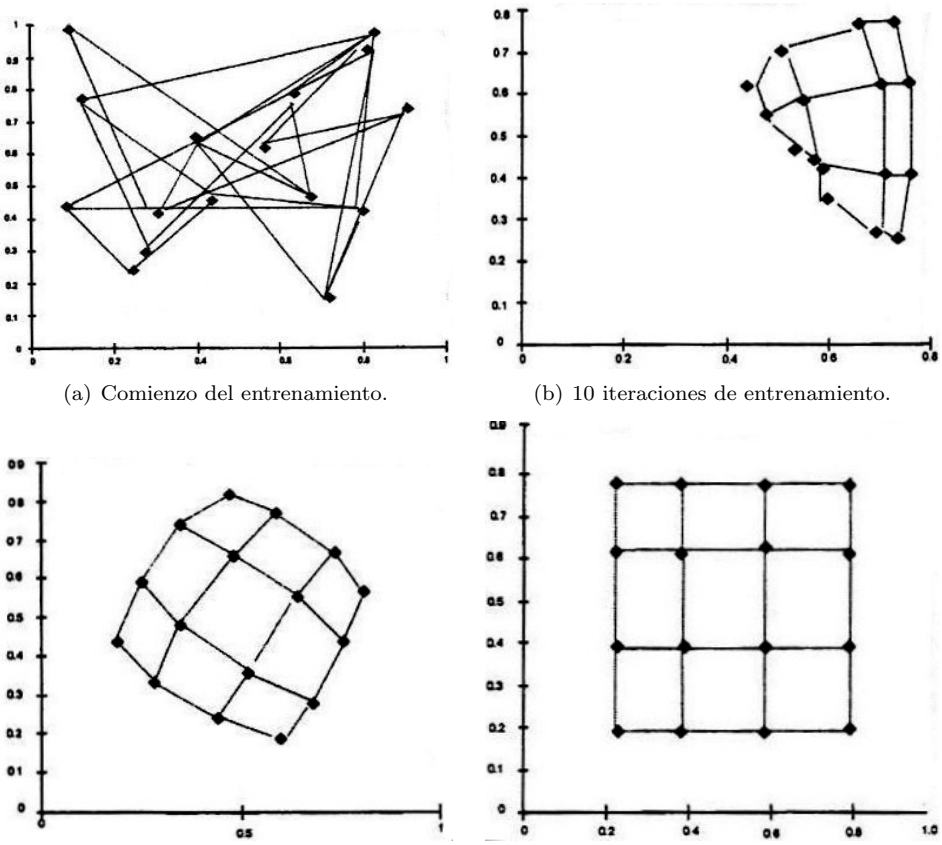
5.4.2. Aplicación del modelo SOM

Para explicar mejor el comportamiento de las redes neuronales *SOM* se presentan a continuación dos ejemplos [48].

En el primer ejemplo, 65,000 muestras distribuidas uniformemente en un espacio de dos dimensiones en el rango $[0, 1]$ en cada eje de coordenadas, se ingresan a la red *SOM* para efectuar el entrenamiento no supervisado. Cada dato es unidimensional.

La red *SOM* se construye usando 2 neuronas en la capa de entrada y 4×4 neuronas en la capa de salida. En la Figura 5.5 se puede observar la distribución de los pesos de la red. Los rombos indican la localización de las neuronas de salida en término de los pesos asociados. Las líneas de la Figura 5.5.a muestran los enlaces entre las distintas neuronas adyacentes en la corteza de mapeo.

Al comienzo del entrenamiento, los pesos son inicializados con valores aleatorios en el rango $[0, 1]$ (Figura 5.5.a). Después de 10 iteraciones, los pesos de la red *SOM* comienzan a expandirse (Figura 5.5.b). Tras 1000 iteraciones, el orden entre los patrones de entrada se puede ver por la topología creada por las neuronas de salida y sus pesos en la Figura 5.5.c. Concluidas 6000 iteraciones los pesos se disponen de manera aproximadamente uniforme como se puede ver en la Figura 5.5.d.



(a) Comienzo del entrenamiento.

(b) 10 iteraciones de entrenamiento.

(c) 1000 iteraciones de entrenamiento.

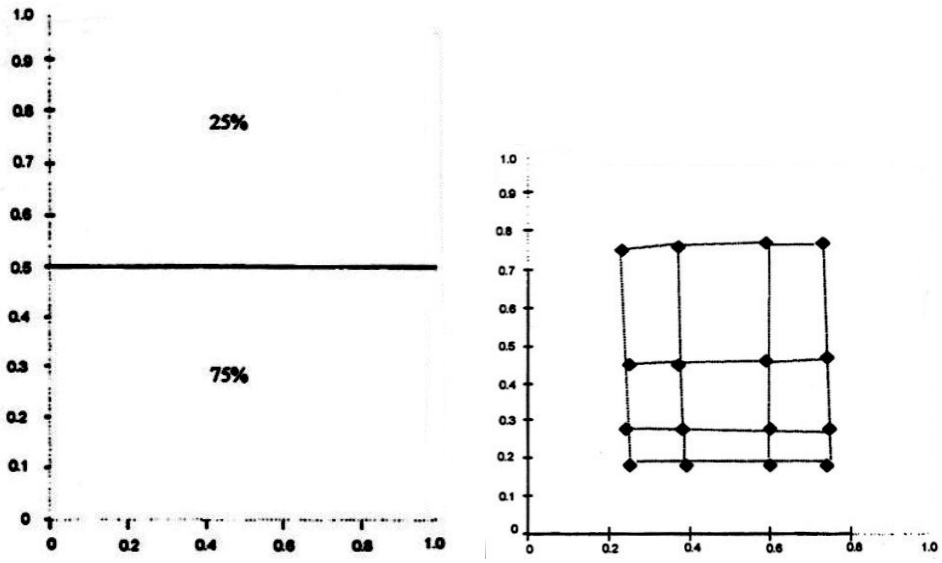
(d) 6000 iteraciones de entrenamiento.

Figura 5.5: Ejemplo de entrenamiento SOM con muestras uniformemente distribuidas

El segundo ejemplo se puede observar en la Figura 5.6. El número de datos de entrenamiento y sus rasgos característicos son los mismos que en el ejemplo anterior. Sin embargo, se realiza un pequeño cambio: el 75 % de las muestras aleatorias caen en el área inferior de la figura Figura 5.6.a.

La distribución de pesos resultantes después de 6000 iteraciones se observa en la figura Figura 5.6.b. Claramente el resultado es distinto del mostrado en la figura Figura 5.5.d del caso anterior.

La mayoría de los pesos de la Figura 5.6.b se distribuyen en el área inferior. Sólo 4 de las 16 neuronas se localizan en la parte superior. Este ejemplo muestra que variar la distribución de las muestras repercute en el resultado de la red *SOM*.



(a) Distribución de las muestras. (b) 6000 iteraciones de entrenamiento.

Figura 5.6: Ejemplo de entrenamiento SOM con muestras distribuidas con probabilidad 0.75 de ser menor a 0.5.

CAPÍTULO 6

Ingeniería de Software

6.1. Introducción

La *ingeniería de software* es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software [52].

Un estándar es un conjunto de criterios aprobados, documentados y disponibles para determinar la adecuación de una acción (estándar de proceso) o de un objeto (estándar de producto).

Esta capítulo se focaliza en los estándares de proceso de la ingeniería de software. Estos se caracterizan por:

- Garantizar una práctica responsable durante todo el ciclo de vida del software.
- Orientar la administración de la configuración, el aseguramiento de la calidad y la verificación del producto.
- Consolidar la tecnología existente en una base firme para introducir nuevas tecnologías.
- Incrementar la disciplina profesional.
- Proteger a los negocios.
- Proteger al comprador o cliente.
- Mejorar al producto.

6.2. Ciclo de vida del software

El ciclo de vida del software es el proceso a seguir para construir, entregar y hacer evolucionar un sistema computacional. Por lo tanto, comprende etapas desde la concepción de una idea hasta la entrega, el mantenimiento y el retiro de un producto asociado a ésta.

Las actividades involucradas durante el desarrollo deberán ser sistemáticamente planeadas y llevadas a cabo para garantizar que la aplicación cumpla todos sus requisitos.

Un modelo de ciclo de vida de un sistema estructura las tareas del proyecto en 'fases', las cuales poseen un alcance y un resultado bien definido.

Las fases que incluye el ciclo de vida del desarrollo de un software son [53]:

- Fase RU: Definición de los Requerimientos de Usuario.
- Fase RS: Definición de los Requerimientos de Software.
- Fase DA: Diseño Arquitectónico.

- Fase DD: Diseño Detallado y producción del código.
- Fase TR: Transferencia de software a operaciones.
- Fase OM: Operaciones y Mantenimiento.

6.2.1. Fase RU: Definición de los Requerimientos de Usuario

La *fase RU* es la 'fase de definición del problema' en el ciclo de vida de un proyecto de software. El objetivo de esta etapa es pulir una idea o una necesidad a ser cubierta, para determinar el alcance del sistema informático.

En esta etapa es responsabilidad del usuario la definición correcta de los requerimientos deseados.

El producto de esta fase es un *Documento de Requerimientos de Usuario* o *DRU* (en inglés, *User Requirement Document* o *URD*), en el que se exponen formalmente todos los requerimientos de usuario. Éste posee las siguientes características:

- Se produce antes de comenzar el proyecto de software.
- Incluye un identificador para cada requisito de usuario.
- Asegura que cada requisito de usuario sea verificable.
- Señala como esenciales los requisitos así considerados.
- Define una prioridad a cada requisito con el objetivo de facilitar entregas incrementales del producto.
- Identifica claramente los requisitos de usuario no aplicables.
- Proporciona una descripción general de lo que el usuario espera del producto.
- Es completo, es decir, incluye todos los requisitos de usuario.
- Detalla las operaciones que el usuario pretende realizar con el sistema.
- Define todas las restricciones a las que el usuario desea imponer una solución.
- Describir las interfaces externas del software.

La salida de esta fase se controla durante una reunión de revisión de requerimientos de usuario.

6.2.2. Fase RS: Definición de los Requerimientos de Software

Esta fase se puede llamar 'fase de análisis del problema' del ciclo de vida de un sistema computacional. En ella se examinan los requisitos de usuario y se producen los requerimientos de software relacionados.

Los requerimientos de software se determinan examinando el *DRU* y construyendo un modelo lógico, es decir, una descripción abstracta de lo que el sistema debe realizar. Se describe 'qué' hacer y no 'cómo' hacer, omitiendo, por lo tanto, todo detalle de implementación.

El modelo lógico se usará para producir un conjunto estructurado, consistente y completo de requisitos de software. Este se registra en el *Documento de Requerimientos de Software* o *DRS* (en inglés, *Software Requirement Document* o *SRD*).

Esta documentación determina el problema desde el punto de vista del desarrollador y no del usuario. Se caracteriza por:

- Incluir un identificador para cada requisito de software.
- Asegurar que cada requisito de software sea verificable.
- Señalar como esenciales los requisitos así considerados.

- Definir una prioridad a cada requisito con el objetivo de facilitar entregas incrementales del producto.

- Determinar una correspondencia entre cada requisito de software con al menos un requisito de usuario.

- Cubrir todos los requerimientos establecidos en el *DRU*.

La salida de esta fase se debe controlar formalmente durante una revisión de los requerimientos de software.

6.2.3. Fase DA: Diseño Arquitectónico

El propósito de la *fase DA* es definir la estructura del software. Para lograr esto, se transforma el modelo lógico descripto en la *fase RS* en un modelo físico o diseño arquitectónico.

Un diseño arquitectónico determina todos los componentes que formarán parte del sistema, y el control y flujo de datos entre ellos. A cada entidad se le asocia una funcionalidad específica y una interfaz de uso bien definida.

En esta etapa es posible identificar dificultades técnicas. Diseños alternativos pueden ser propuestos, pero sólo se elige y documenta uno de ellos.

La salida formal de esta fase, exigida para todo proyecto de software, es el *Documento de Diseño Arquitectónico* o *DDA* (en inglés, *Architectural Design Document* o *ADD*).

El *DDA* posee las siguientes particularidades:

- Refleja el diseño arquitectónico seleccionado para el desarrollo del sistema, es decir, los principales componentes del software y las interfaces de comunicación entre ellos.

- Para cada componente detalla los datos de entrada, las operaciones a llevar a cabo y la salida esperada.

- Para cada interfaz define una estructura de datos que la conforma.

- Para cada estructura de datos provee:

  - La descripción de cada elemento que la constituye,
  - La relación entre sus elementos,
  - El rango de valores posible para cada elemento, y
  - El valor inicial de cada elemento.

- Identifica las interfaces externas a utilizarse por el sistema.

- Evalúa los recursos del computador necesarios en el ambiente de desarrollo y en el ambiente operacional.

- Determinar una correspondencia entre las distintas entidades del diseño arquitectónico con al menos un requisito de software.

- Cubre todos los requerimientos establecidos en el *DRS*.

- Es consistente.

Durante la revisión de diseño arquitectónico se examina y controla el documento anteriormente descripto.

6.2.4. Fase DD: Diseño Detallado y producción del código

El objetivo de la *fase DD* es refinar el diseño del software, codificarlo y testearlo. Para ello se deben considerar los siguientes tres principios:

- Debe ser posible una descomposición *'top-down'*, es decir, de lo general a lo específico,
- Debe ser posible una programación estructurada u orientada a objetos, y
- Debe ser posible la realización concurrente de la producción del sistema y de la documentación.

En base al último ítem expuesto, se llevan a cabo simultáneamente actividades relacionadas a la codificación y testeo de unidades, y tareas vinculadas a la elaboración de la documentación (*Documento de Diseño Detallado* (o *DDD*) y *Manual de Usuario del Software* (o *MUS*)).

Inicialmente, el *DDD* y el *MUS* contienen las secciones correspondiente a los niveles más altos del sistema. Mientras el diseño progresa a niveles más bajos, las subsecciones de mayor detalle son agregadas. Finalmente, se completan los documentos y, junto al código, constituyen las salidas de esta fase.

Durante la etapa de diseño detallado, se efectúan distintos tipos de pruebas de acuerdo a los planes de verificación establecidos en las fases *RS* y *DA*. Estas son:

- Pruebas de unidad o *'unit tests'*: Deben avalar la calidad del código. Es recomendable establecer un objetivo de prueba como, por ejemplo, nivel o porcentaje de cobertura mínimo de requisitos.
- Pruebas de integración o *'integration tests'*: Verifican que toda la información intercambiada a través de una interfaz, coincida con la estructura de datos establecida en el *DDA* y confirman que el flujo de control implementado también corresponda con el definido en el mismo documento.
- Pruebas de sistema o *'system tests'*: Aseguran la correspondencia entre las operaciones implementadas y los objetivos del sistemas instaurados en el *DRS*.

Para finalizar el proceso, se realiza una revisión del diseño detallado. En este momento, el software está preparado para ser verificado mediante las pruebas de aceptación (o *'acceptance tests'*) durante la *fase TR*.

6.2.5. Fase TR: Transferencia de software a operaciones

La finalidad de la *fase TR* es confirmar que el software cumple con cada requisito expuesto en el *DRU*. Para ello se debe instalar el producto y verificar que todas las pruebas de aceptación se satisfacen.

Los planes esta fase son establecidos inicialmente en la *fase RU* y actualizados cuando corresponda. En particular, las actividades a realizarse en la *fase TR* se determinan con mayor precisión en la *fase DD*.

Durante la ejecución de las pruebas de aceptación deben participar tanto personal representativo de los usuarios como personal de operaciones.

Cuando el software ha demostrado que posee las capacidades requeridas, puede ser provisoriamente aceptado y comienzan las operaciones.

Para que un software sea provisoriamente aceptado debe cumplir con el criterio de test determinado en el documento *SVVP* (*Software Verification & Validation Plan*). Este documento asegura que las actividades de validación son apropiadas para el grado de criticidad del sistema y suficientes para asegurar la calidad del producto.

La salida de esta etapa es el *Documento de Transferencia de Software* (o *DTS*). En él se incluye un informe de las pruebas de aceptación ejecutadas, y la documentación sobre los cambios del software, realizados durante la *fase TR*.

6.2.6. Fase OM: Operaciones y Mantenimiento

Cuando el software cumple todos las pruebas de aceptación está preparado para llegar al usuario, es decir, para ser admitido finalmente.

En este momento comienza un proceso de monitoreo del producto para confirmar que satisfacen todos los requisitos enunciados en el *DRU*. Esta es la etapa de *operación y mantenimiento*.

Por *'mantenimiento'* se entiende toda modificación que el sistema experimente después de haber llegado al usuario, ya sea para corregir errores no detectados durante las fases anteriores o para agregar funcionalidad correspondiente a nuevos requerimientos.

El período de *'operación y matenimiento'* exige preservar la documentación actualizada y consistente con el código, y generar informes respecto a fallas encontradas.

6.3. Modelos del Ciclo de Vida del software

Los modelos de ciclo de vida del software describen las fases de un proyecto de software y el orden en que se ejecutan.

Un modelo de ciclo de vida de software es una vista de las actividades que ocurren durante el desarrollo de una aplicación. Tiene por finalidad determinar el orden de las etapas involucradas y los criterios de transición asociados.

Entre los modelos más conocidos e implementados en la ingeniería se encuentran los siguientes:

- Modelo Cascada,
- Modelo en V,
- Modelo espiral, y
- Modelo de prototipos.

6.3.1. Modelo Cascada

El primer modelo del ciclo de vida del software fue concebido por *Winston W. Royce* en 1970, comúnmente conocido como *'cascada'* o *'lineal secuencial'*.

El *modelo cascada* es un proceso de desarrollo sistemático y secuencial que comienza con la ingeniería del sistema y progresa a través del análisis, diseño, codificación, integración (pruebas) y mantenimiento. Antes de poder avanzar a la siguiente etapa, es necesario haber finalizado completamente la anterior. (Figura 6.1)

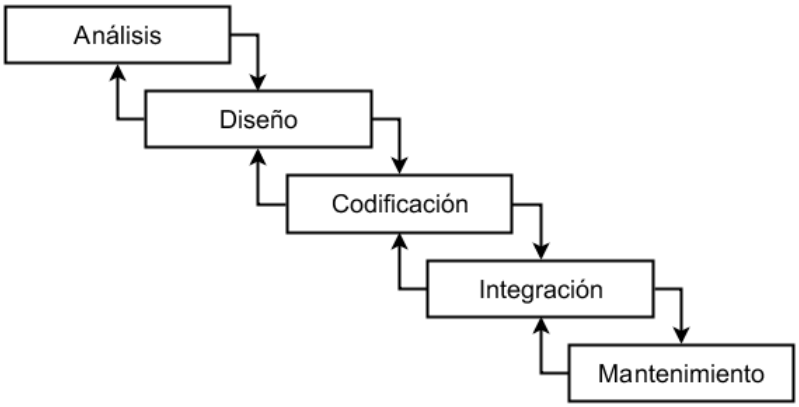


Figura 6.1: Ciclo de vida del modelo cascada [54].

Existen hitos y documentos asociados con cada etapa del proceso, de forma tal que es posible utilizar el modelo para comprobar los avances del proyecto y para estimar cuánto falta para su culminación.

Una modificación sobre este modelo consiste en introducir de una revisión y retroceso, con el fin de corregir las deficiencias detectadas durante las distintas etapas, o para completar o aumentar las funcionalidades del sistema en desarrollo. De esta manera, durante cualquiera de las fases se puede regresar momentáneamente a una fase previa para solucionar el problema que se ha encontrado. Es preciso destacar que cambios en los requisitos en etapas tardías del ciclo de vida del software pueden invalidar gran parte del esfuerzo empleado.

Este modelo es apropiado para proyectos estables (especialmente los proyectos con requerimientos no cambiantes) y donde es posible y probable que los diseñadores predigan totalmente las áreas del problema a resolver con el sistema y produzcan un diseño correcto antes comenzar la implementación. Se adapta perfectamente a proyectos pequeños donde los requerimientos están bien entendidos.

Sin embargo, muchas veces se considera un modelo pobre para proyectos complejos, largos, orientados a objetos y en aquellos en los que los requisitos cambian constantemente. Genera gran cantidad de riesgos ya que los resultados y/o mejoras no son expuestos progresivamente. El producto se exhibe cuando está finalizado, lo cual provoca inseguridad por parte del cliente al no poder percibir los avances en el software requerido.

6.3.2. Modelo en V

Fue desarrollado para solucionar problemas presentes en el enfoque cascada. Los defectos se encontraban demasiado tarde en el ciclo de vida del software, ya que las pruebas no se ejecutaban hasta el final del proyecto. El *modelo en V* (o *modelo de 'Validación y Verificación'*) propone comenzar las pruebas tan pronto como sea posible.

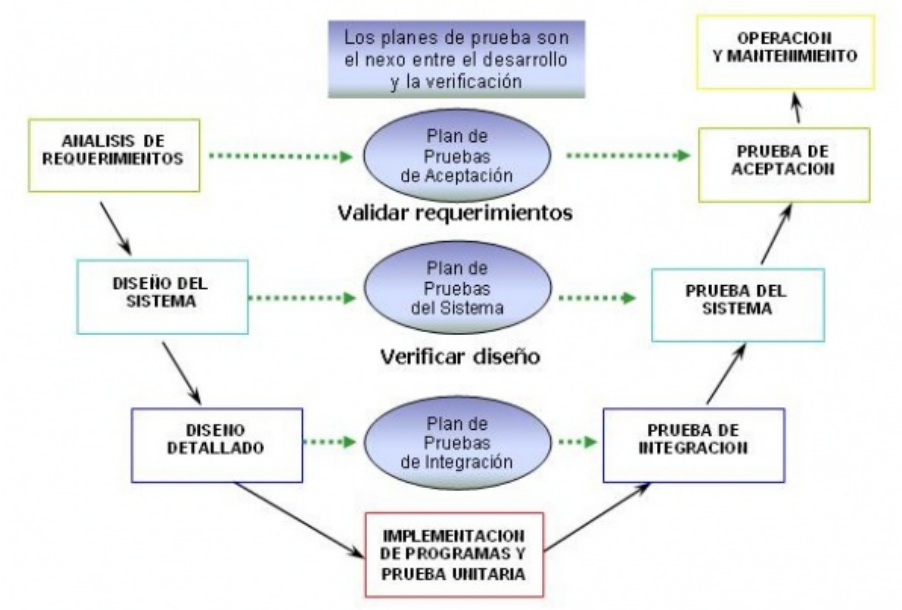


Figura 6.2: Ciclo de vida del modelo en V [55].

Este modelo evidencia que las pruebas no son sólo un proceso basado en la ejecución. Hay una variedad de actividades que deben realizarse antes de concluir la fase de codificación. Por lo tanto, los técnicos de pruebas deben trabajar paralelamente con los desarrolladores y analistas de negocio de tal forma que sea posible producir simultáneamente una serie de documentación de pruebas.

El *modelo en V* describe las actividades y resultados que deben ser producidos durante el progreso del producto. La parte izquierda de la 'V' representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho de la 'V' simboliza la integración de las partes y su verificación. (Figura 6.2)

Es un modelo simple y fácil de implementar. Presenta mayor oportunidad de éxito respecto al modelo en cascada. Se adapta especialmente a proyectos pequeños donde los requisitos son entendidos fácilmente.

6.3.3. Modelo espiral

En la década de los 80 Barry Bochm propuso un *modelo de ciclo de vida en espiral* que sustituye a la solución en fases del modelo cascada con ciclos de experimentación y aprendizaje. El modelo incorpora un nuevo elemento en el desarrollo de software: el *'análisis de riesgos'*.

Es un modelo de proceso de software evolutivo que conjuga la naturaleza evolutiva de la construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial.

Proporciona el potencial para el desarrollo rápido de versiones incrementales de software. En el modelo espiral, el software se produce en una serie de iteraciones incrementales. Durante las primeras iteraciones, la versión incremental podría ser un modelo en papel o un prototipo. Durante las últimas iteraciones, se producen versiones cada vez más complejas del sistema diseñado. (Figura 6.3)

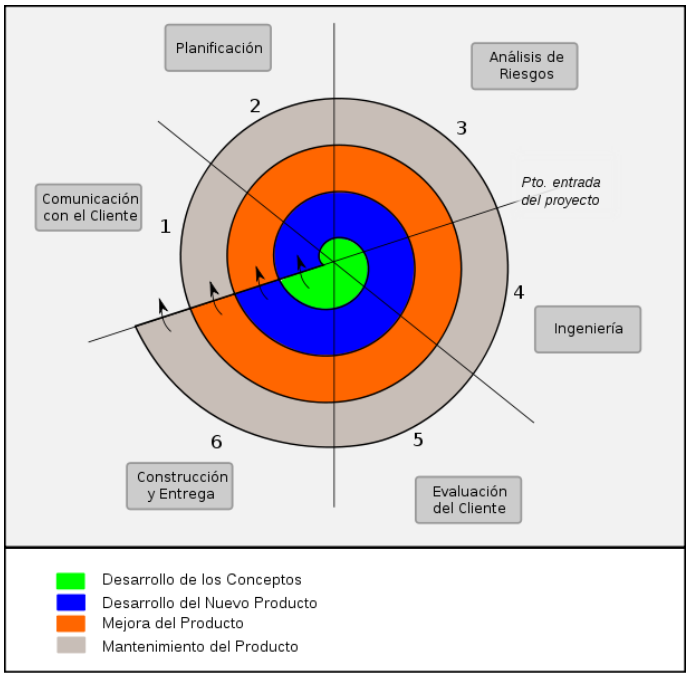


Figura 6.3: Ciclo de vida del modelo espiral [56].

Este modelo se divide en un número de actividades de marco de trabajo llamados 'región de tareas'. Generalmente existen entre tres y seis regiones de tareas:

- Comunicación con el cliente: Comprende las actividades necesarias para establecer acuerdos entre el desarrollador y el cliente.
- Planificación: Comprende las actividades necesarias para definir recursos, tiempo y toda otra información relacionada con el proyecto.
- Análisis de riesgos: Comprende las actividades necesarias para evaluar riesgos técnicos y de gestión.
- Ingeniería: Comprende las actividades necesarias para construir una o más representaciones de aplicación.
- Construcción y acción: Comprende las actividades necesarias para producir, probar, instalar y proporcionar soporte al usuario.

- Evaluación del cliente: Comprende las actividades necesarias para obtener la reacción del cliente según la evaluación de las representaciones de software creadas durante la etapa de ingeniería e implementadas durante la etapa de instalación.

El modelo espiral se caracteriza por generar mucho trabajo adicional al ser el análisis de riesgos una de las tareas principales. Esto lo puede convertir en un modelo costoso y no aplicable proyectos de pequeña envergadura.

6.3.4. Modelo de prototipos

En contraste con la Ingeniería de Software de la década de los 70, que dio respuesta a proyectos grandes pero con requisitos estables, la Ingeniería de Software de los 80 reaccionó ante las complicaciones resultantes de encontrarse con requerimientos poco claros e inestables. Se dio lugar así al *'modelo de prototipos'* propuesto por Gomaa en 1984.

El paradigma de construcción de prototipos comienza con la recolección de requisitos. El desarrollador y el cliente encuentran y definen los objetivos globales para el software, identifican los requerimientos conocidos y las áreas del esquema en donde es obligatorio refinar la definición. Entonces aparece un diseño rápido que se centra en una representación de los aspectos del software que serán visibles para el usuario/cliente. El diseño rápido lleva a la construcción de un prototipo. El prototipo es evaluado por el cliente/usuario y se utiliza para detallar los requisitos del software a desarrollar. La iteración ocurre cuando el prototipo está listo para satisfacer las necesidades del cliente, permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que se necesita hacer. (Figura 6.4)

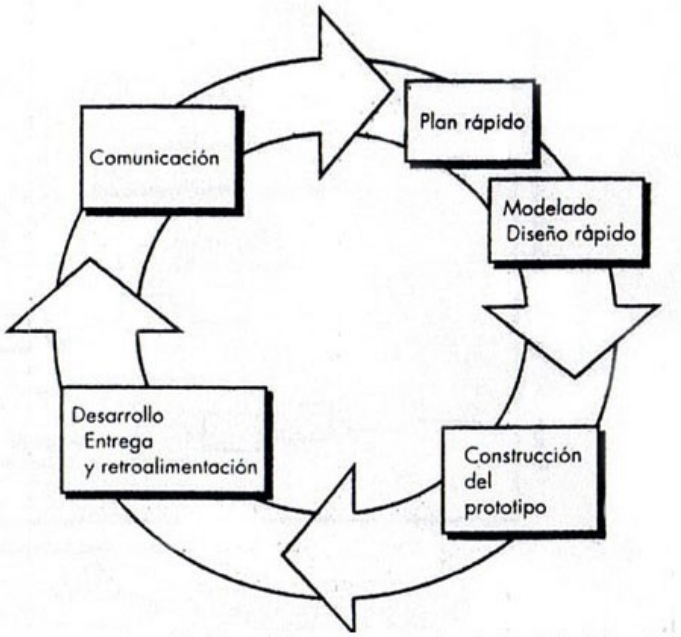


Figura 6.4: Ciclo de vida del modelo de prototipo [55].

Este modelo ofrece visibilidad del producto desde el inicio del ciclo de vida con el primer prototipo. Esto ayuda al cliente a definir mejor los requerimientos y a visualizar las necesidades reales del producto. Permite introducir cambios en las iteraciones siguientes del ciclo y la realimentación continua del cliente.

Además reduce el riesgo de construir productos que no satisfagan las necesidades de los usuarios, pero esto puede llevar tener un desarrollo más lento.

CAPÍTULO 7

Software de clasificación ANNIC

7.1. Introducción

El sistema ***ANNIC***, del inglés *Artificial Neural Network Image Classification*, es el software de clasificación desarrollado para exponer los distintos métodos de categorización planteados a lo largo de este trabajo.

La finalidad de este producto es explorar algoritmos de clasificación no tradicionales que involucren la utilización de las redes neuronales artificiales *Perceptrón* y *SOM*. A su vez brinda la posibilidad de comparar su efectividad y eficiencia con respecto a un método tradicional de categorización: *K-means*.

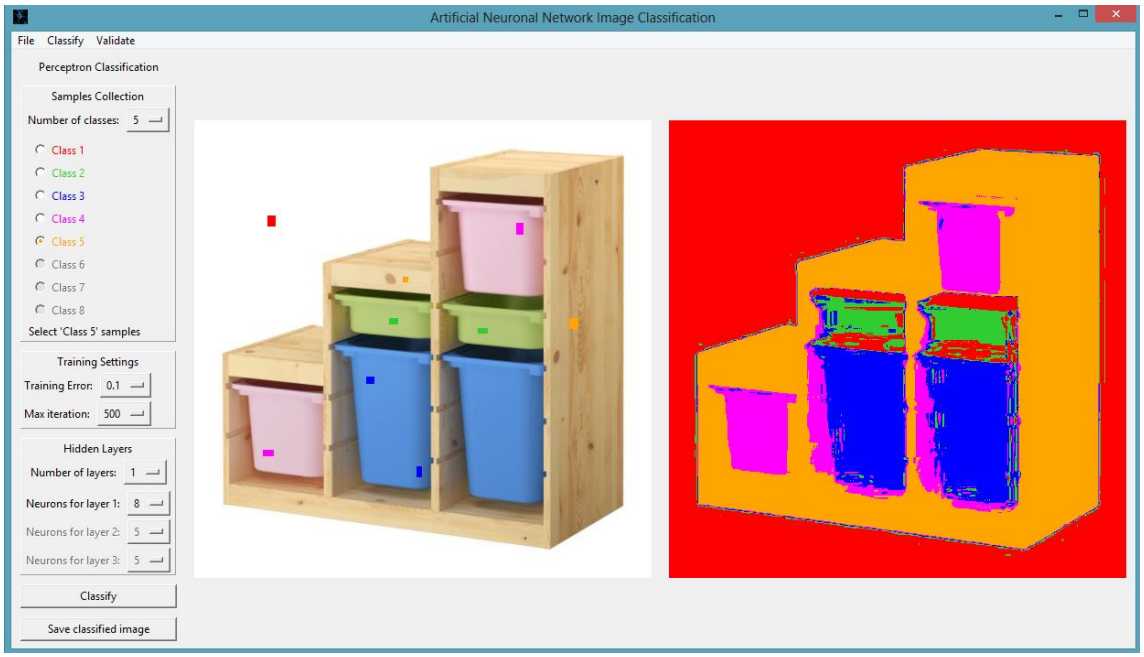


Figura 7.1: Clasificación de imagen basada en una *RNA Perceptrón* del sistema *ANNIC*.

Para cumplir los objetivos descriptos, el sistema *ANNIC* expone las siguientes funcionalidades principales:

- Clasificación de imágenes digitales basada en una *RNA Perceptrón*,

- Clasificación de imágenes digitales basada en una *RNA SOM*,
- Clasificación de imágenes digitales basada en el algoritmo *K-means*, y
- Evaluación de la clasificación mediante el cálculo de una *matriz de confusión* y su *coeficiente Kappa* relacionado.

En la Figura 7.1 se puede observar un ejemplo de clasificación basada en una *RNA Perceptrón* del sistema *ANNIC*. Se muestra además, el resultado de la ejecución de este algoritmo: una imagen con cinco categorías definidas, las cuales tiene correspondencia con las muestras de entrenamiento seleccionadas por el usuario.

Cabe aclarar que los textos visualizados en la interfaz gráfica son en *inglés* ya que este idioma es considerado la lengua de comunicación global.

7.2. Estándar de software empleado: PSS-05

Todo el proceso del desarrollo del sistema *ANNIC* es realizado en base al estándar **PSS-05-0** de la *ESA* (*European Space Agency*) [57]. Este estándar es recomendado para proyectos de pequeña envergadura ya que avala un enfoque simplificado de los estándares de la ingeniería de software.

Un proyecto de software puede ser considerado pequeño si se satisface alguno de los siguientes criterios:

- Son necesarios menos de dos años de desarrollo,
- El sistema completo puede ser elaborado por menos de 5 personas,
- La cantidad de líneas de código que conforma la aplicación es menor a 10,000.

Las estrategias recomendadas por *ESA* para la producción de pequeños proyectos de software abarcan los siguientes aspectos:

- Combinar las fases de requerimientos de software (*fase RS*) y de diseño de la arquitectura (*fase DA*),
- Simplificar la documentación,
- Reducir la formalidad de los requisitos, y
- Usar la especificación de los pruebas de sistema para las pruebas de aceptación.

7.2.1. Combinación las fases RS y DA

Los estándares indican que la definición de requerimientos de software y el diseño de la arquitectura deben ser realizados en fases separadas. Estas fases terminan con una revisión formal del Documento de Requerimientos de Software y del Documento de Diseño Arquitectónico.

Cada revisión normalmente incluye al usuario, y puede durar entre dos semanas y un mes. Para un proyecto de software pequeño, las revisiones del *DRS* y el *DDA* por separado pueden alargar los tiempos significativamente.

Por lo tanto, una manera eficiente de organizar estas etapas es:

- Combinar las fases *RS* y *DA* en una sola *fase RS/DA*, y
- Combinar las revisiones *RS/R* y *DA/R* en una sola revisión formal al finalizar la *fase RS/DA*.

7.2.2. Simplificación la documentación

Las normas *PSS-05-0* de la *ESA* proveen modelos de documentos basados en los estándares *ANSI/IEEE*, y diseñados para cubrir toda la documentación requerida por los proyectos de software.

Para el caso de los sistemas pequeños las normas establecen el uso de modelos reducidos donde:

- Los desarrolladores deben combinar los documentos *DRS* y *DDA* en un mismo documento denominado *Documento de Especificación de Software* (o *DES*), cuando las fases *RS* y *DA* se combinen.
- Los desarrolladores deben documentar el diseño detallado en el código fuente y extender el *DES* para contener cualquier información que no pueda estar contenida en el código.
- La producción del Documento de Historial del Proyecto es opcional.

Adjuntos al sistema de clasificación de imágenes *ANNIC* se entregan los documentos *'User Requirement Document'*, *'Software Specification Document'* y *'User Manual Document'* (Apéndice A, Apéndice B y Apéndice C respectivamente).

7.2.3. Reducción la formalidad de los requisitos

La solvencia de un producto de software se logra diseñando formalmente el sistema, revisando rigurosamente el código y la documentación, y probando incrementalmente las distintas funcionalidades.

La formalidad de los requisitos de software siempre debe ajustarse de acuerdo al costo de corrección de los defectos durante la fase de desarrollo frente al costo de reparar errores en etapas tardías.

Hay evidencia que la cantidad de defectos disminuye significativamente cuando la cobertura de test sobrepasa el 90 % de los requisitos. Sin embargo, lograr una alta protección en las pruebas puede ser costoso en términos de esfuerzo, e inclusive exceder el costo de reparación de los errores durante la etapa de operaciones.

Por ello, se recomienda:

- Establecer un objetivo de prueba de requisitos con cobertura del 80 %, y
- Revisar cada requisito no cubierto en las pruebas.

Existen herramientas software disponibles para medir la cobertura de test. Deben usarse siempre que sea posible.

7.2.4. Uso de especificaciones de pruebas de sistema para pruebas de aceptación

Cuando el desarrollador es responsable de definir las pruebas de aceptación, a menudo repite procedimientos detallados en las pruebas de sistema.

Por lo tanto, un método recomendable para la documentación de pruebas de aceptación es indicar en la especificación de pruebas de sistema cuales escenarios y procedimientos pueden ser reutilizados para cubrir las pruebas de aceptación.

7.3. Tecnología utilizada en el diseño: UML

El diseño del software *ANNIC* se basa en el estándar **UML**, del inglés *Unified Modeling Language*: Lenguaje de Modelado Unificado [58]. Esta elección es consecuencia de la expresividad que provee el lenguaje para representar gráficamente, analizar, entender y reflejar la solución propuesta.

7.3.1. Concepto

UML es un lenguaje, por lo tanto proporciona un vocabulario y conjunto de reglas para combinar sus palabras con el objetivo de posibilitar la comunicación. Al ser un lenguaje de modelado su vocabulario y reglas se centran en la representación conceptual y física de un sistema.

Nunca es suficiente un único modelo para comprender un producto de software; se requieren múltiples modelos conectados entre sí. Esto genera la necesidad, a la cual UML da respuesta, de un lenguaje que abarque las diferentes vistas de la arquitectura de un sistema conforme evoluciona a través del ciclo de vida del desarrollo de software.

El vocabulario y las reglas del lenguaje UML indican cómo crear y leer modelos bien formados. Sin embargo, no determina qué modelos se deben diseñar ni cuándo hacerlos. Esta es la tarea del proceso de desarrollo de software y se adapta según lo requieran de los distintos proyectos. En particular, en el sistema *ANNIC* es necesario definir un *diagrama de componentes* y los *diagrama de secuencias* relacionados a las funcionalidades principales del producto, para su correcto entendimiento. Éstos se encuentran detallados en el *Documento de Especificación de Software* (Apéndice B).

7.3.2. Funcionalidades

Las funciones principales del lenguaje UML son las siguientes:

- Visualizar: UML permite expresar de forma gráfica un sistema de manera que otro programador lo pueda entender.
- Especificar: UML permite construir modelos precisos, completos y no ambiguos, con el objetivo de detallar un sistema antes de su producción.
- Construir: A partir de los modelos UML diseñados se pueden elaborar el software, es decir, es posible la generación del código en base a los prototipos expuestos (ingeniería directa).
- Documentar: Los elementos gráficos UML sirven como documentación del sistema desarrollado.

Para lograr el cumplimiento de estas funcionalidades, UML provee tres clases de bloques:

- Elementos: Abstracciones de cosas reales o físicas, como por ejemplo, abstracciones de objetos,
- Relaciones: Vínculos establecidos entre los elementos de un sistema, y
- Diagramas: Colecciones de elementos con sus relaciones.

7.3.3. Diagramas UML

Un diagrama ofrece una vista del sistema a modelar. UML provee una amplia variedad de diagramas para poder representar correctamente un producto de software desde distintas perspectivas.

Entre los diagramas UML se incluyen los siguientes:

- Diagrama de casos de uso: Representa gráficamente los casos de uso de un producto. Se define como *caso de uso* cada interacción supuesta con el sistema donde se representen los requisitos funcionales.
- Diagrama de clases: Muestra un conjunto de clases, interfaces y sus relaciones.
- Diagrama de secuencia: Exhibe la interacción de los objetos que componen un sistema de forma temporal. Incluye los mensajes que pueden ser enviados entre las distintas partes.
- Diagrama de objetos: Presenta un conjunto de objetos y sus relaciones.
- Diagrama de colaboración: Especifican las relaciones entre los roles. También son llamados *diagramas de comunicación.*

- Diagrama de estados: Expone una máquina de estados que consta de estados, transiciones, eventos y actividades.
- Diagrama de actividades: Revela la estructura de un proceso y detalla el flujo de control y de datos paso a paso en la ejecución su algoritmo interno.
- Diagrama de componentes: Define la encapsulación de una clase, junto con sus interfaces, puertos y estructura interna.
- Diagrama de despliegue: Describe la configuración de nodos de procesamiento en tiempo de ejecución y los artefactos que residen en ellos.

Los diagramas más utilizados son los de casos de uso, clases y secuencia ya que con éstos es posible tanto resumir y comunicar el funcionamiento completo de un producto de software, como especificar en detalle los componentes de un sistema y sus relaciones para su posterior implementación.

Los demás esquemas muestran otros aspectos a modelar. Para detallar el comportamiento dinámico de la aplicación se usan generalmente los diagramas de colaboración, de estados y de actividades. Los diagramas de componentes y de despliegue están enfocados a la implementación del software.

En la Figura 7.2 se pueden observar el diagrama de componentes correspondiente al sistema de categorización *ANNIC*. El *subsistema de clasificación*, el *subsistema de control* y el *subsistema de validación* hacen posible proveer cada una de las funcionalidades requeridas para este software.

También determina que la interacción con el usuario es posible únicamente a través del *subsistema de control*, unidad cuya responsabilidad es la comunicación y coordinación de los procesos a concretarse por los demás elementos del sistema para realizar la acción solicitada por el usuario.

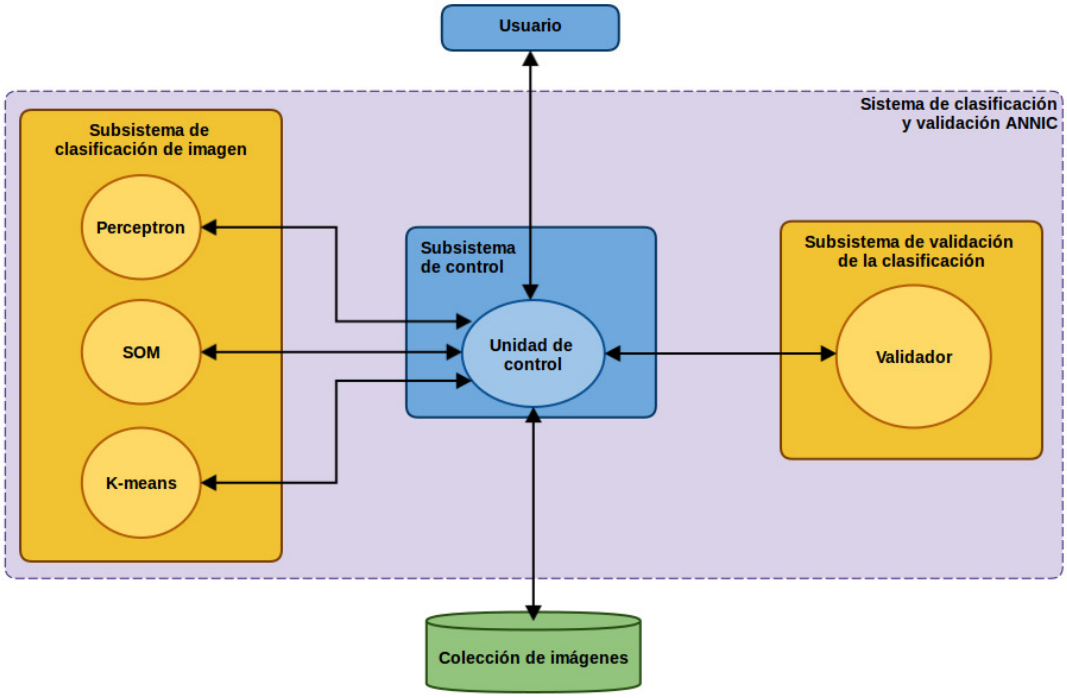


Figura 7.2: Diagrama de componentes del sistema *ANNIC*.

Un ejemplo de *diagrama de secuencia* que explica el proceso completo de clasificación basada en una *RNA SOM* del sistema *ANNIC* se muestra en la Figura 7.3.

Los *diagramas de secuencia* asociados a otros métodos de categorización provistos por el producto y al método de verificación se exponen en el *Documento de Especificación de Software* (Apéndice B).

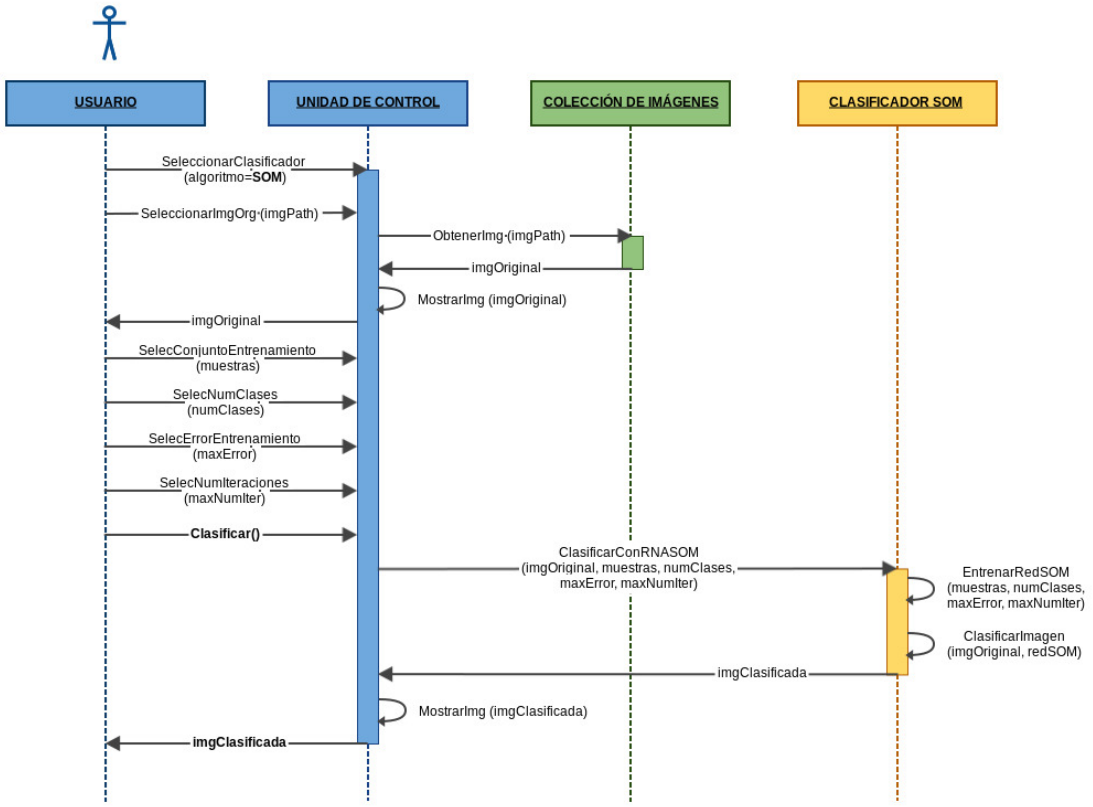


Figura 7.3: Diagrama de secuencia de clasificación basada en una *RNA SOM* del sistema *ANNIC*.

7.4. Tecnología usada en la implementación: Python

El lenguaje de programación seleccionado para el desarrollo de la aplicación *ANNIC* es ***Python*** [62] debido a su gran potencial.

Python provee una sintaxis sencilla y clara, tipado dinámico, gestor de memoria y un conjunto de bibliotecas con alto alcance en cuanto a funcionalidades cubiertas. Cuenta además, con estructuras de datos eficientes y de alto nivel, y un enfoque simple pero efectivo de la programación orientada a objetos.

7.4.1. Características del lenguaje

A continuación se exponen las principales características que brinda el lenguaje *Python*. Éstas hacen que sea elegido y recomendado por gran parte de la industria (como por ejemplo, *Google*):

- **Python es software libre**: Todos sus usuarios tienen permitido ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software.
- **Python es un lenguaje activo y en crecimiento**: La comunidad de desarrolladores que eligen *Python* es muy grande y se incrementa continuamente. Esta corporación se caracteriza fuertemente por una buena aceptación de nuevos miembros y por una gran predisposición a la ayuda mutua a través de medios como foros y lista de correos. En Argentina la comunidad *PyAr* (o *Python Argentina*) es responsable de nuclear a los usuarios de *Python* y centralizar la comunicación a nivel nacional.
- **Python es flexible**: Gracias a su sintaxis clara y simple, este lenguaje es fácil de aprender y entender. Esto repercute positivamente en obtener resultados en forma temprana y hace factible, de manera activa, la mantención, extensión, modificación y mejora de las aplicaciones desarrolladas utilizando esta tecnología.

- **Python es eficiente**: Este lenguaje cuenta con una enorme cantidad de bibliotecas implementadas en *C* y *C++* que permiten realizar operaciones complejas de manera rápida y eficiente. Incluso permite implementar en *C* funcionalidades críticas en cuanto a tiempo de procesamiento y luego utilizarlas desde porciones de código *Python*.

- **Python tiene gran soporte para la computación científica**: Este lenguaje provee librerías sólidas, fuertemente mantenidas, bien documentadas y eficientes para el desarrollo y aplicación en la ciencia. En particular, en el sistema *ANNIC* se utilizaron las siguientes librerías existentes para el soporte científico: *'neurolab'*, *'numpy'*, *'PIL'* y *'csipy'*.

- **Python es portable**: Este lenguaje es multiplataforma, por lo tanto permite desarrollar fácilmente aplicaciones para ser ejecutadas en *Linux*, *Windows* o *MAC*. Esto es importante ya que no limita al usuario a utilizar un determinado sistema operativo. Por lo tanto, el software de clasificación *ANNIC* puede emplearse en cualquiera de los sistemas operativos mencionados.

7.5. Paradigma de programación aplicado: POO

El desarrollo del sistema *ANNIC* se basa en el paradigma de **'programación orientada a objetos'** (o *POO*) debido a sus grandes capacidades y ventajas respecto a otros métodos de programación.

La aplicación se diseña a partir de un conjunto de objetos que interactúan entre sí a través de diferentes técnicas, entre las que se incluyen la herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Como su nombre lo indica, la *POO* utiliza objetos como elementos fundamentales en la construcción de la solución. Un objeto es una abstracción de algún hecho o ente del mundo real, con atributos que representan sus características o propiedades, y métodos que emulan su comportamiento o actividad. Todas las propiedades y métodos comunes a los objetos se encapsulan o agrupan en clases. Una clase es una plantilla, es decir, un prototipo para crear objetos. En general, se dice que cada objeto es una instancia o ejemplar de una clase.

7.5.1. Conceptos fundamentales

La programación orientada a objetos es una forma de desarrollo que introduce nuevos conceptos respecto a conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- **Clase**: Definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

- **Objeto**: Instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de un grupo de comportamientos o funcionalidades (métodos).

- **Método**: Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un 'mensaje'. Un método puede producir un cambio en las propiedades del objeto, o la generación de un 'evento' con un mensaje asociado para otro objeto del sistema.

- **Evento**: Es un suceso en el sistema. Incluye tanto la interacción del usuario con la aplicación como reacción que puede desencadenar un objeto implementado. El software maneja un evento enviando el mensaje adecuado al objeto pertinente.

- **Atributo**: Característica que tiene una clase.

- **Mensaje**: Comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

- **Propiedad**: Característica 'visible' de un objeto o clase. El valor de ésta pueden alterado por la ejecución de algún método interno o externo del objeto o clase a la cual pertenece.

7.5.2. Características de la POO

Las características más importantes que contempla la programación orientad a objetos se detallan a continuación:

- **Abstracción**: Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un agente abstracto que puede realizar un determinado trabajo, informar, cambiar su estado y comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos. La técnica de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella es posible armar un conjunto de clases que permitan modelar la realidad o el problema a resolver.
- **Encapsulamiento**: Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema.
- **Modularidad**: Es la propiedad que permite subdividir una aplicación en partes más pequeñas (o módulos), cada una de las cuales debe ser tan independiente como sea posible.
- **Principio de ocultación**: Cada clase expone una interfaz que especifica cómo pueden interactuar con los objetos que la instancien. El aislamiento protege a las propiedades de un objeto contra su modificación por otros elementos. Solamente los propios métodos del objeto pueden acceder a su estado interno y modificarlo.
- **Polimorfismo**: Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre. Al invocar un método por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. Cuando esto ocurre en 'tiempo de ejecución', esta característica se denomina *asignación dinámica*.
- **Herencia**: Las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía de clases. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Éstos pueden compartir y extender su comportamiento sin tener que volver a implementarlo. Habitualmente se agrupan los objetos en clases y estas en árboles que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.
- **Recolección de basura**: (Conocida por su nombre en inglés, *garbage collection*). Es la técnica por la cual el entorno se encarga de destruir automáticamente los objetos que hayan quedado sin ninguna referencia a ellos (y por tanto desvincular la memoria asociada). Esto significa que el programador no es responsable de la asignación o liberación de memoria, ya que el entorno es quién la asigna al crear un nuevo objeto y la libera cuando éste no es usado.

7.5.3. Aplicación en el sistema ANNIC

Esta sección tiene por finalidad destacar las principales características de la programación orientada a objetos aplicadas en el desarrollo del sistema de clasificación *ANNIC*.

- **Abstracción**: Esta herramienta se emplea fundamentalmente durante el diseño del sistema *ANNIC*, donde se pretende reflejar *'qué'* hacer y no *'cómo'* hacer. Es imprescindible identificar y dividir el software en partes, y determinar la interacción entre cada una de ellas para ofrecer respuesta a todas las funcionalidades requeridas para este producto. En particular, se establecieron las siguientes entidades: *'unidad de control'*, *'clasificador Perceptrón'*, *'clasificador SOM'*, *'clasificador K-means'* y *'validador'*. A su vez, la *'unidad de control'* requiere una descomposición en nuevas abstracciones (*menú principal'*, *'menú de clasificación'* y *'menu de validación'*) para poder concretar su objetivo.

- **Modularidad y Encapsulamiento**: Estás características son claramente visibles en el *diagrama de componentes* del software *ANNIC* expuesto anteriormente. Es fácil distinguir tres grandes módulos dentro del producto: *'subsistema de clasificación de imágenes'*, *'subsistema de control'* y *'subsistema de validación de la clasificación'*. A su vez se puede observar que el *'clasificador Perceptrón'*, el *'clasificador SOM'* y el *'clasificador K-means'* están encapsulados dentro del mismo nivel de abstracción

- **Principio de ocultación**: Pensar en un módulo como entidad abstracta no resulta valioso si su utilización implica conocerlo en detalle. Una solución es crear puntos de acceso de alto nivel a los módulos definiendo una interfaz de uso para cada uno de ellos y encubriendo el mecanismo de ejecución. En el sistema *ANNIC* cada componente posee una interfaz de interacción. Éstas se encuentran descriptas en la sección *'Descripción de los componentes'* en el *Documento de Especificación de Software* (Apéndice B).

- **Polimorfismo**: Un ejemplo donde se evidencia este concepto es en la implementación de los distintos tipos de clasificación provistos por la aplicación *ANNIC*. Para concretar la funcionalidad de categorización se utilizan tres clasificadores: *'clasificador Perceptrón'*, *'clasificador SOM'* y *'clasificador K-means'*. Cada uno de ellos tiene definido un método *'run'* que ejecutará el algoritmo correspondiente. De acuerdo al método de clasificación seleccionado por el usuario, se instancia el clasificador adecuado y en 'tiempo de ejecución' se determina cuál definición de *'run'* efectuar.

- **Herencia**: El principal componente donde manifiesta esta concepto es en el *'menú de clasificación'* expuesto en la interfaz de usuario. Existe un componente *'Classification Menu'* que permite seleccionar los parámetros de clasificación: el número de categorías deseadas, un conjunto de ejemplares representativos, el error máximo permitido y el número máximo iteraciones posibles. También provee un botón para iniciar el proceso y otro para guardar el resultado (la imagen de categorías). De este componente heredan las clases *'Perceptrón Classification Menu'*, *'SOM Classification Menu'* y *'K-means Classification Menu'*. La primera agrega la posibilidad de seleccionar muestras conocidas por clase y la estructura de capas ocultas para construir la red neuronal Perceptrón, mientras que las otras dos conservan sin modificación la estructura de la clase padre. (Figura 7.4)

- **Recolección de basura**: Dado que el sistema *ANNIC* está desarrollado en *Python*, se utiliza el módulo *'gc'* provisto por el lenguaje para gestionar el recolector de basura. En *Python* los objetos nunca son destruidos de forma explícita, son recolectados cuando ya no son alcanzables. Para objetos que contienen referencias a recursos externos como archivos se recomienda utilizar el método *close()* siempre que sea posible, dado que el recolector no garantiza la liberación de recursos.

7.6. Principal patrón de diseño explorado: 'Observer'

Una estrategia clave empleada principalmente la implementación de la interfaz de usuario del sistema *ANNIC* es el **patrón 'observer'**. Este patrón define una dependencia del tipo *uno-a-muchos* entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica esta variación a todos los dependientes.

Se trata de un patrón de comportamiento, es decir, está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a clases y objetos. Los patrones de comportamiento describen no solamente estructuras de relación entre objetos o clases sino también esquemas de comunicación entre ellos.

7.6.1. Participantes

En el patrón *Observer*, también denominado *'Publicación-Inscripción'*, existen sujetos concretos cuyos cambios pueden resultar de interés a otros objetos, y observadores que necesitan estar pendientes de al menos un elemento de un sujeto concreto para reaccionar ante sus modificaciones.

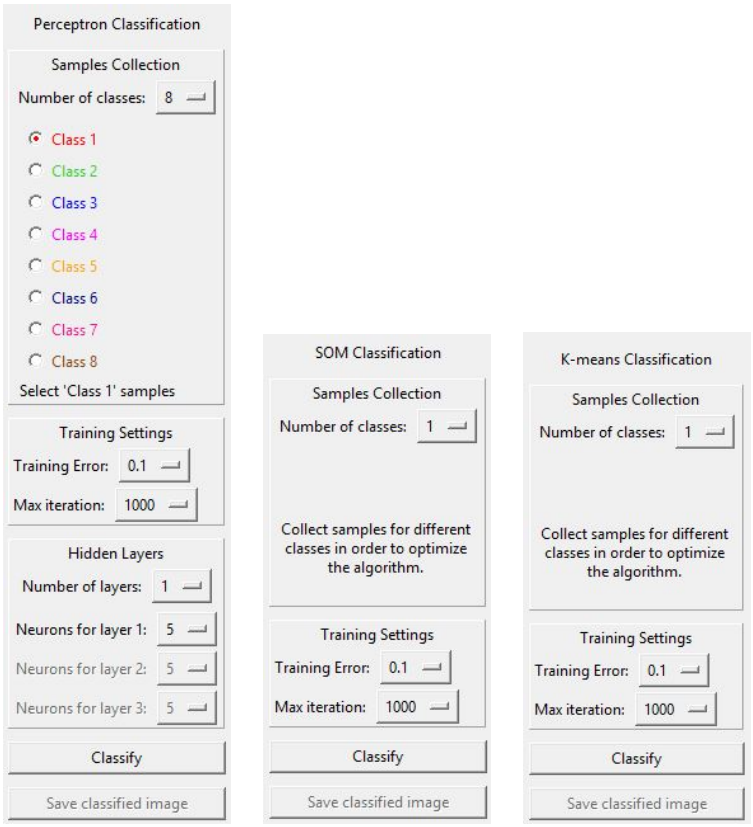


Figura 7.4: Instancias de las clases heredadas de 'Classification Menu': 'Perceptrón Menu', 'SOM Menu' y 'K-means Menu'.

Todos los sujetos tienen en común que un conjunto de observadores quieren estar pendientes de alguno de sus elementos. Cualquier elemento que pueda ser observado debe permitir 'inscribir' observadores, 'desuscribirlos' y poseer un mecanismo de aviso de cambio de estado a los interesados.

A continuación se listan los participantes de forma desglosada:

- **Sujeto** *(Subject)*: Proporciona una interfaz para agregar (*attach*) y eliminar (*detach*) objetos observadores. El *sujeto* conoce a todos sus observadores.
- **Observador** *(Observer)*: Define el método que usa el sujeto para notificar cambios en su estado.
- **Sujeto Concreto** *(Concrete Subject)*: Almacena el estado de interés para sus observadores y les envía notificaciones cuando éste cambia.
- **Observador Concreto** *(Concrete Observer)*: Mantiene una referencia al *Sujeto Concreto* e implementa una interfaz de actualización. En caso de ser notificado de algún cambio, reaccionar ante éste.

La colaboración más importante en este patrón se evidencia entre el *sujeto* y sus *observadores*, ya que cuando el *sujeto* experimenta un cambio, este notifica el nuevo estado a sus *observadores* para que respondan con la funcionalidad adecuada según su responsabilidad.

7.6.2. Consecuencias

El empleo el patrón *Observer* tiene aparejadas numerosas consecuencias. Las más importantes son las siguientes:

- Permite modificar *sujetos* y *observadores* de manera independiente,

- Permite reutilizar un *sujeto* sin reusar sus *observadores*, y viceversa,
- Permite añadir *observadores* sin tener que cambiar el *sujeto* ni los demás *observadores*,
- Abstrae el acoplamiento entre el *sujeto* y cada *observador*. El *sujeto* no 'conoce' la clase concreta de sus *observadores*,
- Soporta la difusión: El *sujeto* envía la notificación a todos los *observadores* suscritos. Se pueden añadir/quitar *observadores*, y
- Puede ejecutar actualizaciones inesperadas: Una operación en el *sujeto* puede desencadenar cambios no deseados en sus *observadores*. El protocolo no ofrece detalle sobre lo que ha cambiado.

7.6.3. Aplicación en el sistema ANNIC

Dado los beneficios listados previamente, el sistema *ANNIC* utiliza el *patrón observer* como herramienta para alcanzar el comportamiento esperado de este software. Es preciso aclarar que el empleo de este patrón es cuidadosamente testeado debido a que su consecuencia negativa puede tener un alto impacto en la experiencia de usuario.

En la siguiente imagen (Figura 7.5) se puede observar ejemplos de uso del *patrón observer* en el diseño y codificación del producto.

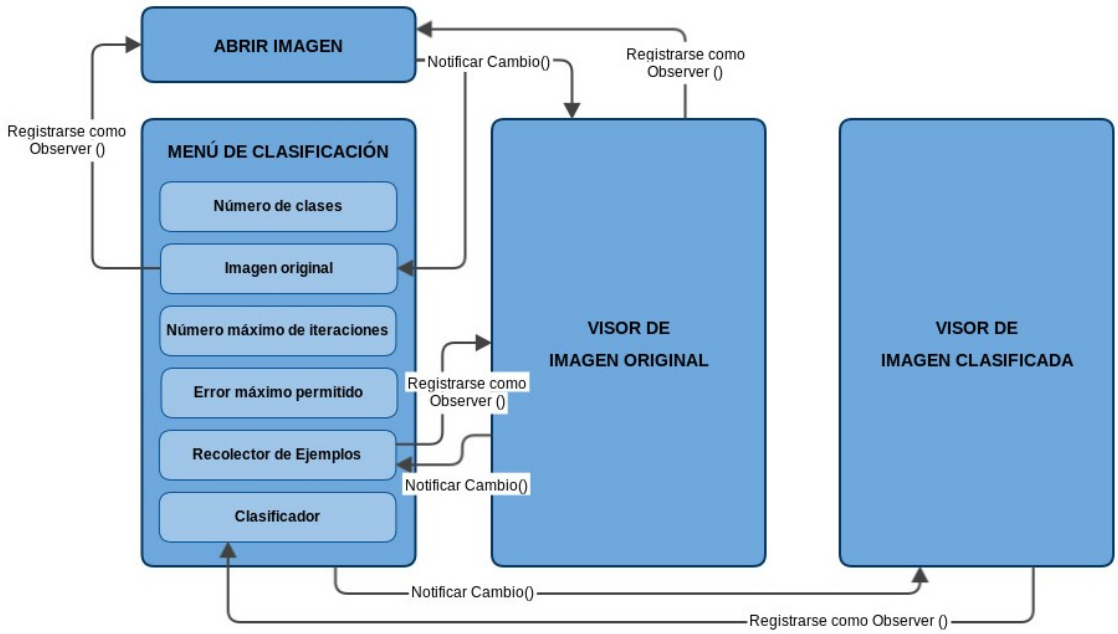


Figura 7.5: Ejemplo de uso del patrón observer en la interfaz de clasificación del sistema *ANNIC*.

La clase *'Abrir imagen'*:

- Provee un método para registrar *observadores*.
  - Las clases *'Visor de imagen original'* e *'Imagen original'* del menú de clasificación se suscriben para informarse cada vez que el usuario seleccione una nueva imagen a categorizar.
- Expone un método de notificación de *observadores*: *'establecer imagen original'*.
  - Este método es invocado cuando el usuario abre una imagen.

- Este método es implementado por las dos clases 'observadoras' de tal manera que al ser invocado la clase *'Visor de imagen original'* muestra en pantalla la imagen a clasificar y la clase *'Imagen original'* del menú de clasificación guarda internamente una referencia a la imagen para emplearla luego en el proceso de clasificación.

La clase *'Visor de imagen original'*:

- Provee un método para registrar *observadores*.
  - La clase *'Recolector de ejemplos'* (del menú de clasificación) se suscribe para informarse cuando que el usuario seleccione nuevas muestras de entrenamiento.
- Expone un método de notificación de *observadores*: *'agregar ejemplos'*.
  - Este método es invocado cada vez que el usuario seleccione un nuevo conjunto de muestras de entrenamiento (rectángulos dibujados sobre la imagen original).
  - Este método es implementado por la clase 'observadora' tal que al ser invocado agrega a una colección interna de muestras de entrenamiento los nuevos ejemplares.

La clase *'Clasificador'*:

- Provee un método para registrar *observadores*.
  - La clase *'Visor de imagen clasificada'* se suscribe para informarse cuando termina el algoritmo de clasificación.
- Expone un método de notificación de *observadores*: *'establecer imagen clasificada'*.
  - Este método es invocado cuando el algoritmo de categorización seleccionado por el usuario finaliza.
  - Este método es implementado por la clase 'observadora' tal que al ser invocado muestra en pantalla la imagen categorizada.

CAPÍTULO 8

Resultados y conclusiones

8.1. Resultados y conclusiones: Pruebas de estrés

8.1.1. Pruebas de estrés ejecutadas

El sistema de clasificación de imágenes *ANNIC* provee la posibilidad de ejecutar pruebas de estrés sobre los algoritmos disponibles en el software.

Para realizar estas pruebas se utilizan dos imágenes de ejemplo, una considerada como 'caso base' (Figura 8.1.a) y otra que representa una situación de categorización más compleja (Figura 8.1.b).

(a) Ejemplo caso base. (b) Ejemplo caso complejo.

Figura 8.1: Imágenes de ejemplo utilizadas en las pruebas de estrés del sistema ANNIC.

Para cada prueba de estrés se considera el mismo conjunto de entrenamiento y el mismo conjunto de validación. En base a ellos, se ejecutan los tres métodos de clasificación, *Perceptrón*, *SOM* y *K-means*, 1000 veces para la imagen 'caso base' y 100 veces para la imagen 'caso complejo'. En toda iteración se calcula el tiempo empleado para la categorización y se verifica la calidad de la clasificación, almacenando el coeficiente *Kappa* obtenido.

8.1.2. Resultados

Cuando se utiliza la imagen 'caso base' para ejecutar las pruebas de estrés (Figura 8.1.a), el coeficiente *Kappa* obtenido en todas las iteraciones de cada uno algoritmos de clasificación es siempre **Kappa = 1**.

Sin embargo es preciso resaltar que el tiempo de ejecución de los distintos métodos varía considerablemente, obteniendo mejor rendimiento el tradicional algoritmo *K-means* y peor la red neuronal *SOM* (Figura 8.2):

- **Algoritmo K-means**:
  - El **99.9 %** de las iteraciones se efectúan en un tiempo **menor a un segundo**.
  - **El tiempo promedio de ejecución de la clasificación es 0.42 segundos.**
- **Algoritmo Perceptrón**:
  - El **69 %** de las iteraciones se efectúan en un tiempo **entre 2 y 3 segundos**, mientras que el **31 %** restante completan su ciclo invirtiendo **entre 3 y 4 segundos**.
  - **El tiempo promedio de ejecución de la clasificación es 3 segundos.**
- **Algoritmo SOM**:
  - El **91 %** de las iteraciones se efectúan en un tiempo **entre 21 y 22 segundos**.
  - **El tiempo promedio de ejecución de la clasificación es 21.66 segundos.**

Figura 8.2: Rendimiento de los algoritmos de clasificación en un caso base.

Respecto a las pruebas de estrés realizadas con la imagen de mayor complejidad (Figura 8.1.b) se puede concluir que el método que expone mejores resultados de clasificación es el algoritmo basado en una red neuronal *Perceptrón*. En este punto, el método *K-means* es el que finaliza con una calidad de clasificación más baja.

Es preciso mencionar que la unidad de medida para evaluar la calidad de la categorización es el coeficiente *Kappa*. Éste indica una mejor clasificación cuando su valor tiende a uno. Valores negativos o cercanos a cero sugieren que el acuerdo observado es puramente debido al azar.

Los resultados referidos a la calidad del reconocimientos de patrones se pueden sintetizar en los siguientes:

- **Algoritmo K-means**:
  - Todas las iteraciones evalúan la clasificación con un **valor Kappa entre 0.4 y 0.6**.
  - **El coeficiente Kappa promedio de las pruebas de estrés es 0.51**

- **Algoritmo Perceptrón**:
  - Todas las iteraciones evalúan la clasificación con un **valor Kappa entre 0.8 y 1**.
  - **El coeficiente Kappa promedio de las pruebas de estrés es 0.87**
- **Algoritmo SOM**:
  - El **67 %** de las iteraciones evalúan la clasificación con un **valor Kappa entre 0.8 y 1**, mientras que el **33 %** restante la evalúan con un **valor entre 0.6 y 0.8**.
  - **El coeficiente Kappa promedio de las pruebas de estrés es 0.76**

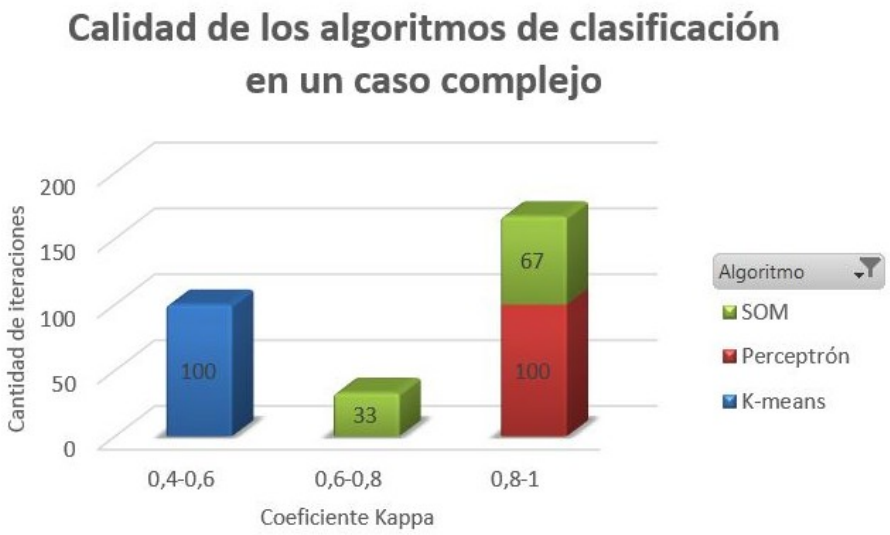


Figura 8.3: Calidad de los algoritmos de clasificación en un caso complejo.

En cuanto a los tiempos de clasificación para la imagen de mayor complejidad, el método *K-means* continúa siendo mucho más eficiente que los dos algoritmos que involucran redes neuronales (Figura 8.4):

- **Algoritmo K-means**:
  - El **100 %** de las iteraciones se efectúan en un tiempo **menor a 5 segundos**.
  - **El tiempo promedio de ejecución de la clasificación es 1.08 segundos.**
- **Algoritmo Perceptrón**:
  - El **95 %** de las iteraciones se efectúan en un tiempo **entre 15 y 20 segundos**
  - **El tiempo promedio de ejecución de la clasificación es 18.74 segundos**, sin considerar en este cálculo dos casos anómalos en los que el ciclo se completa en más de 100 segundos.
- **Algoritmo SOM**:
  - La distribución de la mayoría de tiempos de ejecución se concentra en un rango **mayor a 1 minuto y menor a 1 minuto y 40 segundos**.
  - **El tiempo promedio de ejecución de la clasificación es 74.36 segundos**, sin considerar en este cálculo el caso atípico en el que la iteración se completa en más de 100 segundos.

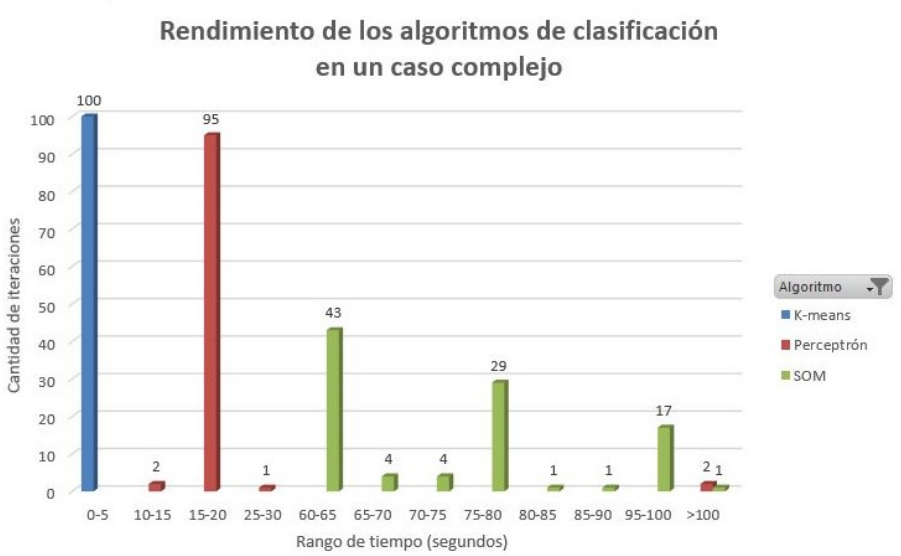


Figura 8.4: Rendimiento de los algoritmos de clasificación en un caso complejo.

8.1.3. Conclusiones

A pesar del eficiente rendimiento, en cuanto a tiempo de ejecución, demostrado por el algoritmo *K-means* en los resultandos anteriores, es necesario destacar que su calidad de clasificación para casos más complejos y donde los parámetros de entrada no aseguran tener ninguna distribución probabilística, se aleja mucho de una categorización deseada. El coeficiente *Kappa* es cercano a cero.

El método de clasificación basado en los mapas auto organización de Kohonen (redes *SOM*) evidencia mejorar la calidad obtenida en la imagen clasificada, pero su rendimiento es muy lento (supera el minuto para obtener una categorización de bondad 0.76 en promedio).

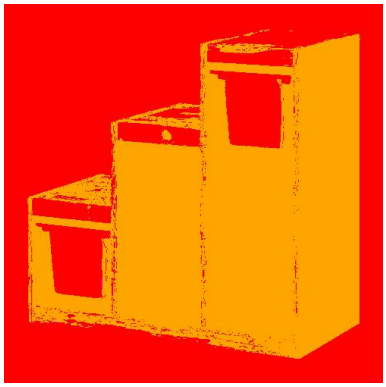
El algoritmo de clasificación que involucra una red neuronal *Perceptrón* arroja los mejores resultados en cuanto a calidad, mostrando un coeficiente *Kappa* cercano a 0.9 en promedio. Si bien su tiempo de ejecución es mayor que el del algoritmo *K-means*, el beneficio obtenido en la calidad de la imagen de categorías justifica su utilización. Además, la duración de cada iteración es aproximadamente la cuarta parte del costo del algoritmo basado en la red neuronal *SOM*.

Es importante resaltar que las redes neuronales ofrecen la gran ventaja de ser algoritmos que no se ajustan a ningún supuesto. Sin embargo, es necesario definir ciertos parámetros para su funcionamiento, como la arquitectura de la red y la tasa de aprendizaje. Estos parámetros afectan directamente el tiempo de entrenamiento, el rendimiento y la tasa de convergencia de la red neuronal.

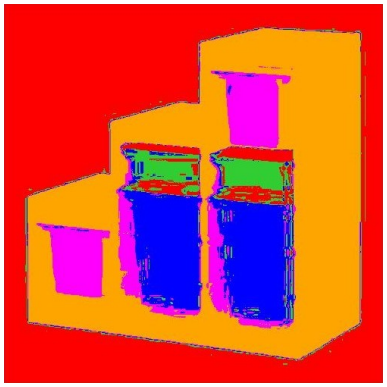
No existen reglas para asistir al diseño de la red ni para elegir una tasa de aprendizaje adecuada, sólo se utilizan heurísticas.

Una mala elección de estos parámetros puede llevar a clasificaciones muy pobres, como se muestra en el ejemplo de la Figura 8.5.a, donde se ejecuta el método de clasificación basado en una red neuronal *Perceptrón* sobre la imagen ejemplo 'caso complejo' (Figura 8.1.b) con una estructura de dos capas ocultas con 5 y 3 neuronas respectivamente. Sólo dos clases de las cinco categorías esperadas son halladas por el algoritmo.

En contrapartida, la Figura 8.5.b muestra la imagen de categorías resultante de ejecutar el algoritmo de clasificación que involucra una red neuronal *Perceptrón* con una única capa oculta de 8 neuronas sobre la misma imagen. La clasificación es de excelente calidad.



(a) Imagen clasificada con una red Perceptrón de dos capas ocultas.



(b) Imagen clasificada con una red Perceptrón de una capa ocultas.

Figura 8.5: Imágenes clasificadas empleando redes neuronales Perceptrón con distintas estructuras.

8.2. Resultados y conclusiones: Proceso de desarrollo del software ANNIC

Para poder obtener los resultados anteriormente expuestos y cumplir con el objetivo de este trabajo de investigación fue necesario elaborar el sistema computacional de clasificación *ANNIC* siguiendo los lineamientos de desarrollo de software propuestos para pequeños proyectos por el estándar *PSS-05* de la *ESA* (*European Space Agency*).

Se ejecutaron cada una de las etapas definidas por estas normas para completar el ciclo de vida del software, generando los documentos establecidos para cada fase.

La aplicación de las normas fue satisfactoria, ya que el producto final asegura poseer la calidad deseada, complacer cada funcionalidad requerida por el usuario y cumplir todo requisito de software asociado a éstas.

El desarrollo del sistema *ANNIC* ha demostrado que el estudio detallado del problema, a partir de una definición rigurosa de los requerimientos de usuario y el diseño de una arquitectura modular, permite ejecutar sin dificultades y concluir exitosamente las etapas de diseño e implementación de software.

A su vez, la detección temprana de defectos (incluyendo defectos de requerimientos, defectos de diseño y defectos de implementación) repercute positivamente en el costo del desarrollo de un producto computacional.

Es preciso destacar que el seguimiento de un estándar de software durante el desarrollo del sistema *ANNIC* permite a cualquier persona con cocimientos en ingeniería de software comprender fácilmente el trabajo y ampliar el alcance del producto. Además, la documentación entregada hace posible una comunicación satisfactoria con diseñadores, desarrolladores y potenciales nuevos usuarios.

8.3. Trabajos a futuro

El campo de reconocimiento de patrones empleando redes neuronales artificiales resulta prometedor y de gran interés debido a que simulaciones del cerebro humano, de modo computacional, repercuten positivamente en la eficiencia de algoritmos de categorización.

Considerando la solución explorada respecto al problema de clasificar imágenes digitales durante el desarrollo de este trabajo, se ha demostrado que la red neuronal *Perceptrón* expone resultados satisfactorios. Sin embargo, si bien la calidad de las imágenes categorizadas por este algoritmo es alta, el elevado tiempo de procesamiento y la dependencia de una elección de estructura de red adecuada para lograr la calidad deseada son dos aspectos a mejorar.

Como trabajos futuros, se pueden plantear:

- Mejorar la implementación de la red neuronal *Perceptrón* y su método de entrenamiento, asegurando la paralelización absoluta del procesamiento de cada neurona a través del uso completo de los recursos de hardware disponibles en cada computadora.
- Explorar algoritmos que permitan la elección adecuada de las estructuras internas de las redes neuronales y otros parámetros involucrados en los procesos desempeñados por éstas, con el objetivo de eliminar la posibilidad de obtener resultados pobres de clasificación ante una elección inadecuada de los parámetros.

En cuanto al software entregado, su construcción modular hace posible, de manera sencilla y ágil, agregar nuevos algoritmos de clasificación, poder verificar su calidad y compararlos con otros métodos. Esto resulta de crucial utilidad para aplicaciones académicas que deseen explorar otros métodos de reconocimiento de patrones sobre imágenes digitales en el futuro.

Bibliografía

[1] Chuvieco, Emilio. *Fundamentos de teledetección espacial*. Tercera edición. Madrid: Ediciones Rialp S.A., 1996.

[2] Bustos, Oscar H.; Frery, Alejandro C.; Lamfri Mario A.; Scavuzzo Carlos M. Técnicas Estadísticas en Teledetección Espacial. Noviembre, 2004.

[3] Pitas, Ioannis. *Digital Image Processing Algorithms and Applications*. New York: John Wiley & Sons, 2000.

[4] Wikipedia. RGB — Wikipedia, La enciclopedia libre, 2014. `http://es.wikipedia.org/wiki/RGB`.

[5] Wikipedia. Modelo de color CMYK — Wikipedia, La enciclopedia libre, 2014. `http://es.wikipedia.org/wiki/Modelo_de_color_CMYK`.

[6] Wikipedia. HSL and HSV — Wikipedia, The free encyclopedia, 2014. `http://en.wikipedia.org/wiki/HSI_color_space`.

[7] Warren, Mike. This post is saturated with the brightest colors 'hue' have ever seen, 2013. `http://warrenperception.wordpress.com/2008/04/13/this-post-is-saturated-with-the-brightest-colors-hue-have-ever-seen`

[8] Hudson, W.D.; Ramm, C.W. Correct formulation of the kappa coefficient of agreement, *Photogrammetric Engineering & Remote Sensing*, vol. 53, pp. 421-422. 1987.

[9] Skidmore, A.K. An expert system classifies eucalypt forest types using thematic mapper data and a digital terrain model, *Photogrammetric Engineering & Remote Sensing*, vol. 55, pp. 1149-1164. 1989.

[10] Congalton, R.G. A comparison of five sampling schemes used in assessing the accuracy of land cover-land use maps derived from remotely sensed data. Tesis doctoral, Virginia Polytechnic Institute. Blacksburg, 1989.

[11] Flóres López, Raquel; Fernández, José Miguel. *Las redes neuronales artificiales*. España: Netbiblo, 2008.

[12] De la Fuente Aparicio, María Jesús; Calonge Cano, Teodoro *Aplicaciones de las redes de neuronas en supervisión, diagnosis y control de proceso*. Venezuela: Equinoccio, 1999.

[13] Herzt, John; Krogh, Anders; Palmer, Richard. *Introduction to the theory of neural computation*. Nueva York: The advanced book program, 1991.

[14] Patterson, Dan W. *Artificial Neural Networks, theory and applications*. New Jersey: Prentice Hall, 1996.

[15] Micro Respuestas. ¿Cuáles son las partes de una neurona?, 2014. http://microrespuestas.com/partes-de-una-neurona

[16] Neurofisiología. La neurona, 2014. http://neurofisiologia1.jimdo.com/la-neurona/

[17] Azoff, E. M. *Neural network time series forecasting of financial markets*. New York: John Wiley & Sons, 1994.

[18] Rosenblatt, F. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 1958.

[19] Widrow, B; Hoff, M. E. Adaptive switching circuits. *Ire Wescon*. New York, 1960.

[20] Hebb, D. *The Organization of Behavior*. New York: Wiley, 1949.

[21] Coehn M. A.; Grossberg, S. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*. USA: 1983.

[22] Simpson, P. K. *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations*. New York: Pergamon Press, 1996.

[23] Kosko, B. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Englewood Cliffs, New Jersey: Prentice-Hall International, 1992.

[24] McNelis, P. D. *Neural Networks in Finance: Gaining Predictive Edge in the Market*. USA: Academic Press, 2005.

[25] Hannan E. J.; Quinn B. G. The Determination of the Order of an Autoregression. *Journal of the Royal Statistical Society*. London, 1979.

[26] Akaike H. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*. United States: 1974.

[27] Schwarz, G. Estimating the dimension of a model. *The annals of statistics*. Israel, 1978.

[28] Golub, G.; Heath, H.; and Wahba, G. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*. Estados Unidos, 1979.

[29] Moody, J.; Utans, J. Selecting neural network architectures via the prediction risk: application to corporate bond rating prediction. *Neural Networks in the Capital Markets*. New York: John Wiley & Sons, 1994.

[30] : Moody, J. Note on generalization, regularization and architecture selection in nonlinear learning systems. *Neural Networks for Signal Processing*. Princeton, New Jersey, 1991.

[31] Lachenbruch P.A.; Mickey M.R. Estimation of error rates in discriminant analysis. *Technometrics*. Estados Unidos, 1979.

[32] Kohavi R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. San Francisco, California, 1995.

[33] . Tukey J. W. Bias and confidence in not quite large samples. *Ann. Math. Statist.*. 1958.

[34] Quenouille, M. H. Approximate tests of correlation in time-series. *J. R. Statist Soc*. 1949.

[35] . Efron B. *The jackknife, the bootstrap, and other resampling plans*. Regional Conference Series in Applied Mathematics. Philadelphia: SIAM, 1982.

[36] Efron B.; Tibshirani, J. R. *An introduction to the Bootstrap*. New York: Chapman & Hall, 1993.

[37] Merler S.; Furlanello C. Selection of tree-based classifiers with the bootstrap 632+ rule. *Biometrical Journal*. 1997.

[38] Minsky M.; Seymour P. *Perceptrons: An Introduction to Computational Geometry*. Cambridge: MIT Press, 1969.

[39] Kung S. Y. *Digital Neural Networks*. New Jersey: Prentice Hall, 1993.

[40] Gallant S. *Neural Networks Learning and Expert Systems*. Cambridge: MIT Press, 1993.

[41] Haykin, S. *Neural Networks: A Comprehensive Foundation*. New Jersey: Prentice Hall, 1998.

[42] Haykin, S. *Neural Networks and Learning Machines*. Third edition. New Jersey: Prentice Hall, 2009.

[43] Rumelhart, D. E.; Hinton, G. E.; Williams, J. R.*Learning internal representations by error propagation*. Cambridge: MIT Press, 1986.

[44] Kavzoglu, T. An Investigation of the Design and Use of Feed-forward Artificial Neural Networks in the Classification of Remotely Sensed Images. *PhD Thesis*. Nottingham, 2001

[45] Kanellopoulos, I.; Wilkinson, G.G.; Roli, F.; Austin, J. *Stock Image Neurocomputation in Remote Sensing Data Analysis*. Berlin: Springer, 1997.

[46] Garson G. D. *Neural Networks: An Introductory Guide for Social Scientists*. London: Sage Publications, 1998.

[47] Ardo, J.; Pilesjo, P.; Skidmore, A. Neural network, multi-temporal thematic mapper data and topographic data to classify forest damage in the Czech republic. *Canadian Journal of Remote Sensing*. 1997.

[48] Tso, B.; Mather P. M. *Classification Methods for Remotely Sensed Data*. First Edition. New York: Taylor & Francis, 2001.

[49] Kohonen, T. Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*. 1982.

[50] Kohonen, T. *Self-Organization and Associative Memory*. Thrid Edition. Berlin: Springer-Verlag, 1989.

[51] Jalote, Pankaj. *An Integrated Approach to Software Engineering*. Tercera edición. Nueva York: Springer, 1997.

[52] Abran A.; Moore J. W.; Bourque, P.; Dupuis, R. Tripp, L. L. *Guide to the Software Engineering Body of Knowledge*. California: Computer Society, 2004.

[53] Guide to applying the ESA software engineering standars to small software projects. *ESA Board for Software Standardisation and Control*. Francia, 1996.

[54] Dart, S.A.; Ellison, R.J.; Feiler, P.H.; Habermann; A.N. *Software Development Environments*. IEEE Computer, 1987.

[55] Wiki de la Universidad de Oriente, Núcleo Monagas. Ciclo de vida de la Ingeniería del Software en comparación con los sistemas clásicos, 2014. `http://wiki.monagas.udo.edu.ve/index.php/Ciclo_de_vida_de_la_Ingenier\%C3\%ADa_del_Software_en_comparaci\%C3\%B3n_con_los_sistemas_cl\%C3\%A1sicos`.

[56] Pressman, R. S. *Ingeniería del Software, Un Enfoque Práctico*. Sexta Edición. España: McGraw-Hill, 2005.

[57] ESA Software Engineering Standards. *ESA PSS-05-0*. 1991.

[58] Rumbaugh J. E.; Jacobson, I.; Booch, G. *The unified modeling language reference manual*. Massachusetts: Addison-Wesley-Longman, 1999.

[59] Wikipedia Commons. Unified Modeling Language — Wikipedia Commons, the free media repository, 2014. http://commons.wikimedia.org/wiki/Unified_Modeling_Language.

[60] Hernández Orallo, Enrique. El Lenguaje Unificado de Modelado (UML), 2014. http://www.disca.upv.es/enheror/pdf/ActaUML.PDF.

[61] Freeman, Eric; Freeman, Elisabeth. *Head first dessign patterns*. Estados Unidos: O'Reilly, 2004.

[62] Python Software Foundation. Python Programming Language, 2014. http://www.python.org/.

[63] Python Software Foundation. PyPI, the Python Package Index, 2014. http://pypi.python.org/pypi.

[64] Wikipedia. Python (programming language) — Wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/wiki/Python_\%28programming_language\%29.

[65] Wikipedia. Object-oriented programming — Wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/wiki/Object-oriented_programming.

[66] Observer (patrón de diseño) — Wikipedia, La enciclopedia libre, 2014. http://es.wikipedia.org/wiki/Observer_\%28patr\%C3\%B3n_de_dise\%C3\%B1o\%29.

APÉNDICE A

Documento de Requerimientos de Usuario

User Requirement Document
OF
ANNIC

Florencia Mihaich

(Spanish Version)
Revisión: 0.1

9 de abril de 2014

I. Historial de revisiones

Versión	Fecha	Autor	Resumen de cambios
1.0	09/04/2014	Mihaich, Florencia	Primera versión del documento.

Tabla 1: Historial de revisiones

II. Documentos relacionados

ID	Nombre	Fecha	Autor
BSSC96	Guía para la aplicación de estándares de Ingeniería de Software ESA (Agencia Espacial Europea) para proyectos de software pequeños.	1996	ESA Comité de Estandarización y Control de Software (BSSC)

Tabla 2: Documentos relacionados

III. Tabla de contenidos

1. Introducción

1.1. Propósito de este documento

Este documento tiene por propósito definir correctamente y describir formalmente los requerimientos de usuario del sistema de clasificación de imágenes ***ANNIC***.

1.2. Definiciones, acrónimos y abreviaciones

Acrónimo	Definición
ANNIC	Del inglés, *Artificial Neural Network Image Classification*: 'Clasificación de imágenes utilizando redes neuronales artificiales'.
UR	Del inglés, *User Requirements*: Requerimientos de usuario.
UI	Del inglés, *User Interface*: Interfaz de usuario.
RNA	Red Neuronal Artificial.
ENV	Del inglés, *Environment*: Entorno.
GEN	Del inglés, *General*: General.
PER	Del inglés, *Perceptron*: RNA Perceptrón.
SOM	Del inglés, *Self Organizing Map*: Mapa autoorganizado (de Kohonen).
KMA	Del inglés, *K-Means-Algorithm*: Algoritmo 'K-medias'.
VAL	Del inglés, *Validation*: Validación.
RES	Del inglés, *Restriction*: Restricción.

Tabla 3: Definiciones, acrónimos y abreviaciones

1.3. Referencias

- Documento de definición de requerimientos de usuario según el estándar PSS-05-0.

1.4. Visión general del documento

El presente documento pretende establecer los requerimientos de los usuarios del sistema de clasificación *ANNIC* y definir las restricciones pertinentes en caso de considerarse necesario.

Su estructura se puede resumir de acuerdo a las siguientes secciones:

- Sección 1: Tiene por finalidad dar una primera aproximación de este documento. Menciona las referencias y abreviaciones a utilizar.
- Sección 2: Pretende profundizar tanto el objetivo del sistema como cuestiones acerca de su funcionamiento. Especifica características del ambiente operacional, capacidades de los usuarios, suposiciones pactadas para el desarrollo de las distintas funcionalidades y dependencias externas, entre otros contenidos.
- Sección 3: Determina detalladamente los requisitos expresados por los usuarios para este sistema de clasificación.

2. Descripción general

2.1. Perspectiva del producto

Se pretende desarrollar un sistema que sea capaz de explorar algoritmos de clasificación no tradicionales que involucren la utilización de redes neuronales artificiales, en particular, la *RNA Perceptrón* y la *RNA SOM*.

El objetivo es poder comprar la efectividad y eficiencia de estos métodos que no están sujetos a ningún supuesto, con respecto a procedimientos ampliamente usados para categorizar imágenes, donde es necesario garantizar determinadas precondiciones como la independencia de los datos de entrada.

2.2. Capacidades generales

El sistema ***ANNIC*** (*Artificial Neural Network Image Classification*) permitirá clasificar imágenes digitales utilizando alguno de los siguientes métodos supervisados: red neuronal Perceptrón, mapa autoorganizado de Kohonen (redes SOM) o el tradicional algoritmo k-meas.

A su vez, proporcionará métodos de validación aplicables sobre la imagen clasificada tales como cálculo de una matriz de confusión y determinación del coeficiente *Kappa*.

2.3. Restricciones generales

El software deberá ser desarrollado de acuerdo a los estándares ESA PSS-05 para pequeños proyectos.

2.4. Características de los usuarios

El software *ANNIC* deberá estar dirigido a todo usuario con conocimiento en métodos de clasificación de imágenes digitales.

El usuario no sólo será considerado el maestro o moderador de la clasificación, sino también su intervención será fundamental para validar los resultados de los distintos algoritmos de categorización.

No deberán existir distintas jerarquías de usuarios distinguibles para el uso de este producto.

2.5. Entorno operacional

El sistema deberá ser ejecutado en cualquier computadora con sistema operativo *Windows*. También se desea, con menor prioridad, que sea portable a los sistema operativo *Linux* y *MAC*.

3. Requerimientos específicos

Esta sección describe todos los requerimientos de usuario del sistema de clasificación *ANNIC*. Cada requerimiento es priorizado considerando la siguiente nomenclatura:

- **M**: Requerimiento obligatorio (en inglés, *Mandatory Requirement*). Las características deben estar incluidas en el sistema final.

- **D**: Requerimiento deseable (en inglés, *Desirable Requirement*). Las características deberían estar incluidas en el sistema final a menos que su costo sea realmente alto.
- **O**: Requerimiento opcional (en inglés, *Optional Requirement*). Las características podrían ser incluidas en el sistema final dependiendo de la voluntad del líder del proyecto.
- **E**: Mejoramientos posibles (en inglés, *Posible Requirement Enhancement*). Las características serán descriptas en este documento con la finalidad de dejar asentadas tales ideas. La decisión de cuándo incluirlas en el sistema dependerá del avance de los requerimientos obligatorios.

3.1. Capacidades

Teniendo en cuenta las distintas formas de clasificación basadas en los algoritmos requeridos y las funcionalidades adicionales, a continuación se detallan las capacidades del sistema *ANNIC*.

3.1.1. Requerimientos referidos al entorno de ejecución

ID	Descripción	Prioridad
UR-ENV-01	El sistema deberá ejecutarse en el sistema operativo *Linux*.	D
UR-ENV-02	El sistema deberá ejecutarse en el sistema operativo *Windows*.	M
UR-ENV-03	El sistema deberá ejecutarse en el sistema operativo *MAC*.	D
UR-ENV-04	Todos los textos en la interfaz de usuario (UI) deberán visualizarse en inglés.	M
UR-ENV-05	El idioma de la interfaz de usuario (UI) será consistente con el idioma del sistema operativo.	E

Tabla 4: Requerimientos referidos al entorno de ejecución

3.1.2. Requerimientos de funcionalidad general

ID	Descripción	Prioridad
UR-GEN-01	El usuario deberá seleccionar el algoritmo de clasificación de imagen a ejecutarse entre las siguientes opciones: *RNA Perceptrón*, *RNA SOM* o *K-means*.	M
UR-GEN-02	El usuario deberá poder seleccionar el método de validación: *matriz de confusión*.	M

Tabla 5: Requerimientos de funcionalidad general

3.1.3. Requerimientos de clasificación con red neuronal Perceptrón

ID	Descripción	Prioridad
UR-PER-01	El usuario deberá seleccionar la imagen de entrada a la cual aplicar el algoritmo de clasificación basado en una *RNA Perceptrón*.	M
UR-PER-02	El usuario deberá visualizar la imagen seleccionada para clasificar con el algoritmo basado en una *RNA Perceptrón*.	M
UR-PER-03	El usuario deberá seleccionar la cantidad de clases o grupos a diferenciar por el algoritmo de clasificación basado en una *RNA Perceptrón*.	M
UR-PER-04	El usuario deberá tomar muestras representativas de cada clase sobre la imagen a clasificar utilizando una *RNA Perceptrón*.	M
UR-PER-05	El usuario deberá seleccionar la cantidad máxima de iteraciones y el error máximo permitido en el entrenamiento de la *RNA Perceptrón*.	M
UR-PER-06	El usuario deberá seleccionar el número de capas ocultas y la cantidad de neuronas por capa a utilizar en el entrenamiento de la *RNA Perceptrón*.	M
UR-PER-07	El usuario deberá decidir cuándo comenzar a la ejecución del algoritmo de clasificación basado en una *RNA Perceptrón* considerando los parámetros previamente seleccionados: la cantidad de clases, el conjunto de píxeles de entrenamiento, la cantidad máxima de iteraciones, el error máximo permitido y la arquitectura de capas ocultas.	M
UR-PER-08	El usuario deberá visualizar la imagen clasificada tras finalizar la ejecución del algoritmo basado en una *RNA Perceptrón*.	M
UR-PER-09	El usuario podrá guardar en disco la imagen clasificada por el algoritmo basado en una *RNA Perceptrón*.	M

Tabla 6: Requerimientos de clasificación con red neuronal Perceptrón

3.1.4. Requerimientos de clasificación con red neuronal SOM

ID	Descripción	Prioridad
UR-SOM-01	El usuario deberá seleccionar la imagen de entrada a la cual aplicar el algoritmo de clasificación basado en una *RNA SOM*.	M
UR-SOM-02	El usuario deberá visualizar la imagen seleccionada para clasificar con el algoritmo basado en una *RNA SOM*.	M

UR-SOM-03	El usuario deberá seleccionar la cantidad de clases o grupos a diferenciar por el algoritmo de clasificación basado en una *RNA SOM*.	M
UR-SOM-04	El usuario deberá tomar muestras representativas sobre la imagen de entrada con el objetivo de hacer más eficiente el entrenamiento de la *RNA SOM*.	M
UR-SOM-05	El usuario deberá seleccionar la cantidad máxima de iteraciones y el error máximo permitido en el entrenamiento de la *RNA SOM*.	M
UR-SOM-06	El usuario deberá decidir cuándo comenzar a la ejecución del algoritmo de clasificación basado en una *RNA SOM* considerando los parámetros previamente seleccionados: la cantidad de clases, el conjunto de píxeles de entrenamiento, la cantidad máxima de iteraciones y el error máximo permitido.	M
UR-SOM-07	El usuario deberá visualizar la imagen clasificada tras finalizar la ejecución del algoritmo basado en una *RNA SOM*.	M
UR-SOM-08	El usuario podrá guardar en disco la imagen clasificada por el algoritmo basado en una *RNA SOM*.	M

Tabla 7: Requerimientos de clasificación con red neuronal SOM

3.1.5. Requerimientos de clasificación basada en el algoritmo K-means

ID	Descripción	Prioridad
UR-KMA-01	El usuario deberá seleccionar la imagen de entrada a la cual aplicar el algoritmo de clasificación *K-means*.	M
UR-KMA-02	El usuario deberá visualizar la imagen seleccionada para clasificar con el algoritmo *K-means*.	M
UR-KMA-03	El usuario deberá seleccionar la cantidad de clases o grupos a diferenciar por el algoritmo de clasificación *K-means*.	M
UR-KMA-04	El usuario deberá tomar muestras representativas sobre la imagen de entrada con el objetivo de hacer más eficiente el algoritmo de clasificación *K-means*.	M
UR-KMA-05	El usuario deberá seleccionar la cantidad máxima de iteraciones y el error máximo permitido en la ejecución del entrenamiento del algoritmo de clasificación *K-means*.	M

UR-KMA-06	El usuario deberá decidir cuándo comenzar a la ejecución del algoritmo de clasificación *K-means* considerando los parámetros previamente seleccionados: la cantidad de clases, el conjunto de píxeles de entrenamiento, la cantidad máxima de iteraciones y el error máximo permitido.	M
UR-KMA-07	El usuario deberá visualizar la imagen clasificada tras finalizar la ejecución del algoritmo *K-means*.	M
UR-KMA-08	El usuario podrá guardar en disco la imagen clasificada por el algoritmo *K-means*.	M

Tabla 8: Requerimientos de clasificación basada en el algoritmo K-means

3.1.6. Requerimientos referidos a la validación de la clasificación

ID	Descripción	Prioridad
UR-VAL-01	El usuario deberá seleccionar la cantidad de clases esperadas en la clasificación.	M
UR-VAL-02	El usuario deberá tomar muestras conocidas de cada clase sobre la imagen clasificada.	M
UR-VAL-03	El usuario deberá decidir cuándo comenzar el cálculo la matriz de confusión y el coeficiente *Kappa* considerando las muestras seleccionadas previamente.	M
UR-VAL-04	El usuario deberá visualizar la matriz de confusión y el coeficiente *Kappa* que evalúan la clasificación.	M
UR-VAL-05	El usuario podrá guardar en disco la matriz de confusión y el coeficiente *Kappa* que evalúan la clasificación.	D

Tabla 9: Requerimientos referidos a la validación de la clasificación

3.2. Restricciones

ID	Descripción	Prioridad
UR-RES-01	El desarrollo completo del sistema deberá seguir los estándares ESA para pequeños proyectos.	D

Tabla 10: Requerimientos de restricción de usuario

APÉNDICE B

Documento de Especificación de Software

Software Specification Document

OF

ANNIC

Florencia Mihaich

(Spanish Version)
Revisión: 0.1

3 de mayo de 2014

I. Historial de revisiones

Versión	Fecha	Autor	Resumen de cambios
1.0	03/05/2014	Mihaich, Florencia	Primera versión del documento.

Tabla 1: Historial de revisiones

II. Documentos relacionados

ID	Nombre	Fecha	Autor
URD	*User Requirement Document of ANNIC*. (Documento de requerimientos de usuario del sistema de clasificación de imágenes *ANNIC*).	09/04/2013	Mihaich, Florencia
BSSC96	Guía para la aplicación de estándares de Ingeniería de Software ESA (Agencia Espacial Europea) para proyectos de software pequeños.	1996	ESA Comité de Estandarización y Control de Software (BSSC)

Tabla 2: Documentos relacionados

III. Tabla de contenidos

1. Introducción

1.1. Propósito

Este documento tiene por propósito definir correctamente y describir formalmente los requerimientos de software y el diseño general del sistema de clasificación de imágenes ***ANNIC***.

El público al cual está dirigido comprende tanto al equipo de desarrollo de software como a las personas que harán uso del producto.

1.2. Definiciones, acrónimos y abreviaciones

Acrónimo	Definición
ANNIC	Del inglés, *Artificial Neural Network Image Classification*: 'Clasificación de imágenes utilizando redes neuronales artificiales'.
UR	Del inglés, *User Requirements*: Requerimientos de usuario.
SR	Del inglés, *Software Requirements*: Requerimientos de software.
UI	Del inglés, *User Interface*: Interfaz de usuario.
RNA	Red Neuronal Artificial.
GEN	Del inglés, *General*: General.
PER	Del inglés, *Perceptron*: RNA Perceptrón.
SOM	Del inglés, *Self Organizing Map*: Mapa autoorganizado (de Kohonen).
KMA	Del inglés, *K-Means-Algorithm*: Algoritmo 'K-medias'.
VAL	Del inglés, *Validation*: Validación.
INT	Del inglés, *Interface*: Interfaz.
OP	Del inglés, *Operational*: Operacional.
DOC	Del inglés, *Documentation*: Documentación.
ENV	Del inglés, *Environment*: Entorno.

Tabla 3: Definiciones, acrónimos y abreviaciones

1.3. Referencias

- Documento de definición de requerimientos de usuario según el estándar PSS-05-0.

1.4. Visión general del documento

El presente documento pretende establecer los requerimientos de software del sistema de clasificación *ANNIC*, solución propuesta dada la necesidad de explorar algoritmos de categorización no tradicionales.

Su estructura se puede resumir de acuerdo a las siguientes secciones:

- Sección 1: Tiene por finalidad dar una primera aproximación de este documento. Menciona las referencias y abreviaciones a utilizar.

- Sección 2: Describe el modelo lógico del sistema, en el cual se refleja una interpretación general de los requerimientos de usuario definidos en el documento respectivo (*URD*).
- Sección 3: Determina los requisitos del software a un nivel de detalle suficiente para poder emplease en las siguientes fases del desarrollo del producto: diseño detallado y codificación. Incluye requerimientos funcionales, de interfaces, operacionales, de documentación y de portabilidad.
- Sección 4: Expone el diseño global del sistema y plantea la estrategia a seguir para resolver y construir la solución deseada. Indica los componentes necesarios y el control y flujo de datos entre ellos.
- Sección 5: Brinda una especificación minuciosa de cada uno de los componentes que constituyen el sistema, incluyendo la descripción de las interfaces expuestas, su función, las dependencias en caso de existir y el método de procesamiento del módulo.
- Sección 6: Muestra la correspondencia entre los requerimiento de usuario y los requerimiento de software que los resuelven.
- Sección 7: Muestra la correspondencia entre los requerimiento software y los componentes que los soportan.

2. Descripción del modelo lógico

En la Figura 1 se observa el modelo lógico del sistema de clasificación *ANNIC*, donde se destaca tanto su funcionalidad principal '**clasificar**', como así también su funcionalidad adicional de '**validación**'.

Para poder realizar la clasificación de una imagen digital, será necesario que el usuario provea las siguientes entradas al producto:

- Una imagen a clasificar, considerada como la *imagen original*, y
- Los *parámetros de clasificación* a utilizarse en el entrenamiento del método de categorización deseado.

Los posibles algoritmos de clasificación se basan en la construcción de una *RNA Perceptrón* o una *RNA SOM* o la aplicación del método *K-means*. El resultado de ejecutar cualquiera de estos clasificadores será una *imagen clasificada*.

Con el objetivo de realizar una validación del método utilizado, el sistema considerará las entradas que se detallan a continuación:

- La imagen clasificada obtenida tras aplicar el algoritmo deseado, y
- Los *Parámetros de validación* provistos por el usuario, quien es considerado, en este punto, como el agente externo con conocimiento de las instancias reales de cada clase.

El resultado de la verificación será una *matriz de confusión* y el *coeficiente Kappa* relacionado a esta.

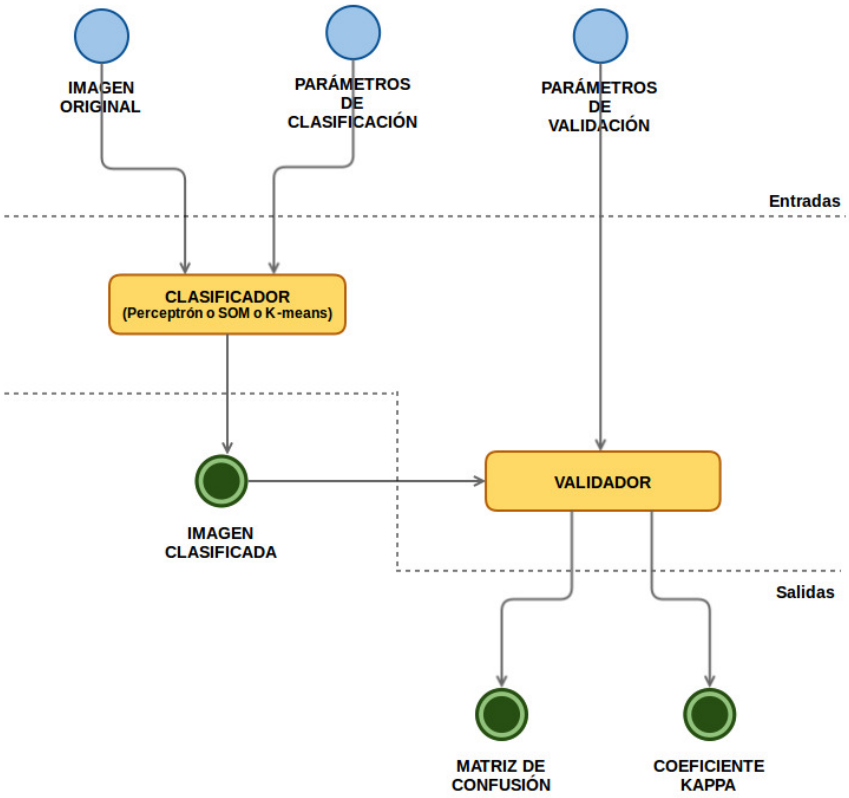


Figura 1: Modelo lógico

3. Requisitos específicos

Esta sección describe todos los requerimientos de software del sistema de clasificación *ANNIC*. Cada requerimiento es priorizado considerando la siguiente nomenclatura:

- **M**: Requerimiento obligatorio (en inglés, *Mandatory Requirement*). Las características deben estar incluidas en el sistema final.
- **D**: Requerimiento deseable (en inglés, *Desirable Requirement*). Las características deberían estar incluidas en el sistema final a menos que su costo sea realmente alto.
- **O**: Requerimiento opcional (en inglés, *Optional Requirement*). Las características podrían ser incluidas en el sistema final dependiendo de la voluntad del líder del proyecto.
- **E**: Mejoramientos posibles (en inglés, *Posible Requirement Enhancement*). Las características serán descriptas en este documento con la finalidad de dejar asentadas tales ideas. La decisión de cuándo incluirlas en el sistema dependerá del avance de los requerimientos obligatorios.

3.1. Requisitos funcionales

Teniendo en cuenta las distintas formas de clasificación basadas en los algoritmos requeridos y las funcionalidades adicionales, a continuación se detallan las los requerimientos funcionales del sistema *ANNIC*.

3.1.1. Requerimientos de funcionalidad general

ID	Descripción	Prioridad
SR-GEN-01	El sistema deberá permitir al usuario seleccionar el algoritmo de clasificación de imagen a ejecutarse.	M
SR-GEN-02	Los posibles algoritmos de clasificación deberán ser: *RNA Perceptrón*, *RNA SOM* o *K-means*.	M
SR-GEN-03	El sistema deberá permitir al usuario validar el resultado de la clasificación.	M

Tabla 4: Requerimientos de funcionalidad general

3.1.2. Requerimientos de clasificación con una RNA Perceptrón

ID	Descripción	Prioridad
SR-PER-01	El sistema deberá permitir al usuario seleccionar la imagen a clasificar con el algoritmo basado en una *RNA Perceptrón*.	M
SR-PER-02	La imagen de entrada, a la cual aplicar el algoritmo de clasificación basado en una *RNA Perceptrón*, deberá ser de formato *'.jpg'*, *'.png'* o *'.tif'*.	M
SR-PER-03	La imagen de entrada, a la cual aplicar el algoritmo de clasificación basado en una *RNA Perceptrón*, podrá ser una imagen en escala de grises o *RGB*.	M
SR-PER-04	El sistema deberá mostrar la imagen seleccionada para clasificar con el algoritmo basado en una *RNA Perceptrón*.	M
SR-PER-05	El sistema deberá permitir al usuario seleccionar la cantidad de clases o grupos a diferenciar por el algoritmo de clasificación basado en una *RNA Perceptrón*.	M
SR-PER-06	La cantidad de clases a diferenciar por el algoritmo de clasificación basado en una *RNA Perceptrón* deberá ser un número entero mayor o igual a 1 y menor o igual a 8.	M
SR-PER-07	La cantidad de clases a diferenciar por el algoritmo de clasificación basado en una *RNA Perceptrón* será por defecto 1.	M

ID	Descripción	Prioridad
SR-PER-08	El sistema deberá permitir tomar muestras representativas de cada clase sobre la imagen a clasificar utilizando una *RNA Perceptrón*.	M
SR-PER-09	El formato de la selección de muestras a emplear en el entrenamiento de la *RNA Perceptrón* será rectangular.	M
SR-PER-10	El sistema deberá asignar automáticamente un color diferente a las muestras de las distintas clases a utilizarse en el entrenamiento de la *RNA Perceptrón*.	M
SR-PER-11	El sistema deberá permitir al usuario seleccionar la cantidad máxima de iteraciones y el error máximo permitido en el entrenamiento de la *RNA Perceptrón*.	M
SR-PER-12	El error máximo permitido en el entrenamiento de la *RNA Perceptrón* deberá ser mayor o igual a 0,1, menor o igual a 2,5 y divisible por 0,1.	M
SR-PER-13	El error máximo permitido en el entrenamiento de la *RNA Perceptrón* será por defecto 0,1.	M
SR-PER-14	El número máximo de iteraciones permitido en el entrenamiento de la *RNA Perceptrón* deberá ser mayor o igual a 500, menor o igual a 7500 y divisible por 500.	M
SR-PER-15	El número máximo de iteraciones permitido en el entrenamiento de la *RNA Perceptrón* será por defecto 1000.	M
SR-PER-16	El sistema deberá permitir al usuario seleccionar el número de capas ocultas y la cantidad de neuronas por capa a utilizar en el entrenamiento de la *RNA Perceptrón*.	M
SR-PER-17	El número de capas ocultas a utilizar para construir la *RNA Perceptrón* deberá ser un número entero mayor o igual a 1 y menor o igual a 3.	M
SR-PER-18	El número de capas ocultas a utilizar para construir la *RNA Perceptrón* será por defecto 1.	M
SR-PER-19	La cantidad de neuronas por capa a utilizar para construir la *RNA Perceptrón* deberá ser un número entero mayor o igual a 1 y menor o igual a 8.	M
SR-PER-20	La cantidad de neuronas por capa a utilizar para construir la *RNA Perceptrón* será por defecto 5.	M
SR-PER-21	El sistema deberá permitir al usuario iniciar el algoritmo de clasificación basado en una *RNA Perceptrón*.	M
SR-PER-22	El sistema deberá tomar en cuenta los parámetros previamente seleccionados para crear y entrenar una *RNA Perceptrón* para la clasificación: la cantidad de clases, el conjunto de píxeles de entrenamiento, la cantidad máxima de iteraciones, el error máximo permitido y la arquitectura de capas ocultas.	M

ID	Descripción	Prioridad
SR-PER-23	El sistema deberá entrenar la *RNA Perceptrón* utilizando el conjunto de píxeles seleccionados para este fin.	M
SR-PER-24	El sistema deberá clasificar todos los píxeles de la imagen utilizando la *RNA Perceptrón* previamente entrenada.	M
SR-PER-25	La cantidad de clases encontradas por el algoritmo basado en una *RNA Perceptrón* será menor o igual al número de clases seleccionadas por el usuario antes de comenzar la clasificación.	M
SR-PER-26	El sistema deberá mostrar la imagen clasificada tras finalizar la ejecución del algoritmo basado en una *RNA Perceptrón*.	M
SR-PER-27	La imagen clasificada por el algoritmo basado en una *RNA Perceptrón* tendrá formato *RGB*.	M
SR-PER-28	El sistema deberá permitir al usuario guardar en disco la imagen clasificada por el algoritmo basado en una *RNA Perceptrón*.	M
SR-PER-29	El sistema deberá permitir al usuario seleccionar el nombre del archivo en el cual guardar la imagen clasificada por el algoritmo basado en una *RNA Perceptrón*.	M
SR-PER-30	La imagen clasificada por el algoritmo basado en una *RNA Perceptrón* podrá ser guardada con formato *'.jpg'*.	M

Tabla 5: Requerimientos de clasificación con una *RNA Perceptrón*

3.1.3. Requerimientos de clasificación con una RNA SOM

ID	Descripción	Prioridad
SR-SOM-01	El sistema deberá permitir al usuario seleccionar la imagen a clasificar con el algoritmo basado en una *RNA SOM*.	M
SR-SOM-02	La imagen de entrada, a la cual aplicar el algoritmo de clasificación basado en una *RNA SOM*, deberá ser de formato *'.jpg'*, *'.png'* o *'.tif'*.	M
SR-SOM-03	La imagen de entrada, a la cual aplicar el algoritmo de clasificación basado en una *RNA SOM*, podrá ser una imagen en escala de grises o *RGB*.	M
SR-SOM-04	El sistema deberá mostrar la imagen seleccionada para clasificar con el algoritmo basado en una *RNA SOM*.	M

ID	Descripción	Prioridad
SR-SOM-05	El sistema deberá permitir al usuario seleccionar la cantidad de clases o grupos a diferenciar por el algoritmo de clasificación basado en una *RNA SOM*.	M
SR-SOM-06	La cantidad de clases a diferenciar por el algoritmo de clasificación basado en una *RNA SOM* deberá ser un número entero mayor o igual a 1 y menor o igual a 8.	M
SR-SOM-07	La cantidad de clases a diferenciar por el algoritmo de clasificación basado en una *RNA SOM* será por defecto 1.	M
SR-SOM-08	El sistema deberá permitir tomar muestras representativas sobre la imagen de entrada con el objetivo de hacer más eficiente el entrenamiento del algoritmo de clasificación basado en una *RNA SOM*.	M
SR-SOM-09	El formato de la selección de muestras a emplear en el entrenamiento de la *RNA SOM* será rectangular.	M
SR-SOM-10	El sistema deberá asignar automáticamente un único color para tomar todas las muestras de las distintas clases a utilizar en el entrenamiento de la *RNA SOM*.	M
SR-SOM-11	El sistema deberá permitir al usuario seleccionar la cantidad máxima de iteraciones y el error máximo permitido en el entrenamiento de la *RNA SOM*.	M
SR-SOM-12	El error máximo permitido en el entrenamiento de la *RNA SOM* deberá ser mayor o igual a 0,1, menor o igual a 2,5 y divisible por 0,1.	M
SR-SOM-13	El error máximo permitido en el entrenamiento de la *RNA SOM* será por defecto 0,1.	M
SR-SOM-14	El número máximo de iteraciones permitido en el entrenamiento de la *RNA SOM* deberá ser mayor o igual a 500, menor o igual a 7500 y divisible por 500.	M
SR-SOM-15	El número máximo de iteraciones permitido en el entrenamiento de la *RNA SOM* será por defecto 1000.	M
SR-SOM-16	El sistema deberá permitir al usuario iniciar el algoritmo de clasificación basado en una *RNA SOM*.	M
SR-SOM-17	El sistema deberá tomar en cuenta los parámetros previamente seleccionados para crear y entrenar una *RNA SOM* para la clasificación: la cantidad de clases, el conjunto de píxeles de entrenamiento, la cantidad máxima de iteraciones y el error máximo permitido.	M
SR-SOM-18	El sistema deberá entrenar la *RNA SOM* utilizando el conjunto de píxeles seleccionados para este fin.	M
SR-SOM-19	El sistema deberá clasificar todos los píxeles de la imagen utilizando la *RNA SOM* previamente entrenada.	M

ID	Descripción	Prioridad
SR-SOM-20	La cantidad de clases encontradas por el algoritmo basado en una *RNA SOM* será menor o igual a al número de clases seleccionadas por el usuario antes de comenzar la clasificación.	M
SR-SOM-21	El sistema deberá mostrar la imagen clasificada tras finalizar la ejecución del algoritmo basado en una *RNA SOM*.	M
SR-SOM-22	La imagen clasificada por el algoritmo basado en una *RNA SOM* tendrá formato *RGB*.	M
SR-SOM-23	El sistema deberá permitir al usuario guardar en disco la imagen clasificada por el algoritmo basado en una *RNA SOM*.	M
SR-SOM-24	El sistema deberá permitir al usuario seleccionar el nombre del archivo en el cual guardar la imagen clasificada por el algoritmo basado en una *RNA SOM*.	M
SR-SOM-25	La imagen clasificada por el algoritmo basado en una *RNA SOM* podrá ser guardada con formato *'.jpg'*.	M

Tabla 6: Requerimientos de clasificación con una *RNA SOM*

3.1.4. Requerimientos de clasificación basada en el algoritmo k-means

ID	Descripción	Prioridad
SR-KMA-01	El sistema deberá permitir al usuario seleccionar la imagen a clasificar con el algoritmo *K-means*.	M
SR-KMA-02	La imagen de entrada, a la cual aplicar el algoritmo de clasificación *K-means*, deberá ser de formato *'.jpg'*, *'.png'* o *'.tif'*.	M
SR-KMA-03	La imagen de entrada, a la cual aplicar el algoritmo de clasificación *K-means*, podrá ser una imagen en escala de grises o *RGB*.	M
SR-KMA-04	El sistema deberá mostrar la imagen seleccionada para clasificar con el algoritmo *K-means*.	M
SR-KMA-05	El sistema deberá permitir al usuario seleccionar la cantidad de clases o grupos a diferenciar por el algoritmo de clasificación *K-means*, tomando en cuenta que se encontrará una cantidad de categorías menor o igual a este número.	M
SR-KMA-06	La cantidad de clases a diferenciar por el algoritmo de clasificación *K-means* deberá ser un número entero mayor o igual a 1 y menor o igual a 8.	M
SR-KMA-07	La cantidad de clases a diferenciar por el algoritmo de clasificación *K-means* será por defecto 1.	M

ID	Descripción	Prioridad
SR-KMA-08	El sistema deberá permitir tomar muestras representativas sobre la imagen de entrada con el objetivo de hacer más eficiente el entrenamiento del algoritmo de clasificación *K-means*.	M
SR-KMA-09	El formato de la selección de muestras de entrenamiento a emplear en el algoritmo *K-means* será rectangular.	M
SR-KMA-10	El sistema deberá asignar automáticamente un único color para tomar todas las muestras de entrenamiento de las distintas clases a utilizar por el algoritmo *K-means*.	M
SR-KMA-11	El sistema deberá permitir al usuario seleccionar la cantidad máxima de iteraciones y el error máximo permitido en la ejecución del entrenamiento del algoritmo de clasificación *K-means*.	M
SR-KMA-12	El error máximo permitido en el entrenamiento del algoritmo de clasificación *K-means* deberá ser mayor o igual a 0,1, menor o igual a 2,5 y divisible por 0,1.	M
SR-KMA-13	El error máximo permitido en el entrenamiento del algoritmo de clasificación *K-means* será por defecto 0,1.	M
SR-KMA-14	El número máximo de iteraciones permitido en el entrenamiento del algoritmo de clasificación *K-means* deberá ser mayor o igual a 500, menor o igual a 7500 y divisible por 500.	M
SR-KMA-15	El número máximo de iteraciones permitido en el entrenamiento del algoritmo de clasificación *K-means* será por defecto 1000.	M
SR-KMA-16	El sistema deberá permitir al usuario iniciar el algoritmo de clasificación *K-means*.	M
SR-KMA-17	El sistema deberá tomar en cuenta los parámetros previamente seleccionados al aplicar el algoritmo de clasificación *K-means*: la cantidad de clases, el conjunto de píxeles de entrenamiento, la cantidad máxima de iteraciones y el error máximo permitido.	M
SR-KMA-18	El sistema deberá aplicar el método de entrenamiento *K-means* sobre el conjunto de píxeles seleccionados para este fin, con el objetivo de encontrar a lo sumo k centroides.	M
SR-KMA-19	El sistema deberá clasificar todos los píxeles de la imagen considerando los centroides definidos tras la ejecución del método de entrenamiento *K-means*.	M
SR-KMA-20	La cantidad de clases encontradas por el algoritmo *K-means* será menor o igual al número de clases seleccionadas por el usuario antes de comenzar la clasificación.	M

ID	Descripción	Prioridad
SR-KMA-21	El sistema deberá mostrar la imagen clasificada tras finalizar la ejecución del algoritmo *K-means*.	M
SR-KMA-22	La imagen clasificada por el algoritmo *K-means* tendrá formato *RGB*.	M
SR-KMA-23	El sistema deberá permitir al usuario guardar en disco la imagen clasificada por el algoritmo *K-means*.	M
SR-KMA-24	El sistema deberá permitir al usuario seleccionar el nombre del archivo en el cual guardar la imagen clasificada por el algoritmo *K-means*.	M
SR-KMA-25	La imagen clasificada por el algoritmo *K-means* podrá ser guardada con formato *'.jpg'*.	M

Tabla 7: Requerimientos de clasificación basada en el algoritmo K-means

3.1.5. Requerimientos de validación de la clasificación

ID	Descripción	Prioridad
SR-VAL-01	El sistema deberá permitir al usuario seleccionar la cantidad de clases esperadas en la clasificación.	M
SR-VAL-02	La cantidad de clases esperadas en la clasificación deberá ser un número entero mayor o igual a 1 y menor o igual a 8.	M
SR-VAL-03	La cantidad de clases esperadas en la clasificación será por defecto 1.	M
SR-VAL-04	El usuario deberá asegurar que el número de clases a validar coincida con el número de clases seleccionado para la clasificación.	M
SR-VAL-05	El sistema deberá permitir tomar muestras conocidas de cada clase sobre la imagen clasificada.	M
SR-VAL-06	El formato de la selección de muestras conocidas será rectangular.	M
SR-VAL-07	El sistema deberá proveer al usuario los mismos colores presentes en la imagen clasificada para la selección de muestras conocidas por clases.	M
SR-VAL-08	El usuario deberá asegurar que el color de cada muestra de validación coincida con el color visualizado en la imagen clasificada.	M
SR-VAL-09	El sistema deberá permitir al usuario iniciar la verificación de la clasificación.	M
SR-VAL-10	El sistema deberá tomar en cuenta los parámetros previamente seleccionados para calcular la matriz de confusión: la cantidad de clases y el conjunto de muestras conocidas por clase.	M

ID	Descripción	Prioridad
SR-VAL-11	El sistema deberá calcular el coeficiente *Kappa* relacionado a la matriz de confusión previamente obtenida.	M
SR-VAL-12	El sistema deberá mostrar la matriz de confusión y el coeficiente *Kappa* obtenido tras procesar la verificación de la clasificación.	M
SR-VAL-13	El sistema deberá permitir al usuario guardar en disco los resultados de la verificación de la clasificación.	D
SR-VAL-14	Los resultados de la validación de la clasificación podrán ser guardados en un archivo de texto.	D

Tabla 8: Requerimientos de validación de la clasificación

3.2. Requisitos de interfaces

ID	Descripción	Prioridad
SR-INT-01	Todos los textos en la interfaz de usuario (UI) del sistema *ANNIC* deberán visualizarse en inglés.	M
SR-INT-02	El idioma de la interfaz de usuario del sistema *ANNIC* será consistente con el idioma del sistema operativo.	E
SR-INT-03	La interfaz de usuario del sistema *ANNIC* deberá proveer un menú para el manejo de archivos.	M
SR-INT-04	El menú para el manejo de archivos de la interfaz de usuario del sistema *ANNIC* deberá permitir al usuario abrir una imagen y guardar la imagen clasificada.	M
SR-INT-05	La interfaces relacionadas a las opciones de 'abrir' y 'guardar' una imagen dependerán de la interfaz de manejo de archivos nativa del sistema operativo donde se ejecute el producto *ANNIC*.	M
SR-INT-06	La interfaz de usuario del sistema *ANNIC* deberá proveer un marco donde visualizar la imagen original seleccionada por el usuario.	M
SR-INT-07	La interfaz de usuario del sistema *ANNIC* deberá proveer un menú de selección de clasificador para permitir al usuario seleccionar el algoritmo de clasificación deseado.	M
SR-INT-08	La interfaz de usuario del sistema *ANNIC* deberá proveer un menú de clasificación '*Perceptrón*' para permitir al usuario ingresar los datos de entrada necesarios para la ejecución del algoritmo de categorización basado en una *RNA Perceptrón*.	M

ID	Descripción	Prioridad
SR-INT-09	La interfaz de usuario del sistema *ANNIC* deberá proveer un botón 'clasificar' para permitir al usuario iniciar la ejecución del algoritmo de clasificación basado en una *RNA Perceptrón*.	M
SR-INT-10	La interfaz de usuario del sistema *ANNIC* deberá proveer un menú de clasificación '*SOM*' para permitir al usuario ingresar los datos de entrada necesarios para la ejecución del algoritmo de categorización basado en una *RNA SOM*.	M
SR-INT-11	La interfaz de usuario del sistema *ANNIC* deberá proveer un botón 'clasificar' para permitir al usuario iniciar la ejecución del algoritmo de clasificación basado en una *RNA SOM*.	M
SR-INT-12	La interfaz de usuario del sistema *ANNIC* deberá proveer un menú de clasificación '*K-means*' para permitir al usuario ingresar los datos de entrada necesarios para la ejecución del algoritmo de categorización *K-means*.	M
SR-INT-13	La interfaz de usuario del sistema *ANNIC* deberá proveer un botón 'clasificar' para permitir al usuario iniciar la ejecución del algoritmo de clasificación *K-means*.	M
SR-INT-14	La interfaz de usuario del sistema *ANNIC* deberá proveer un marco donde visualizar la imagen clasificada tras finalizar la ejecución del algoritmo de categorización seleccionado por el usuario.	M
SR-INT-15	La interfaz de usuario del sistema *ANNIC* deberá proveer un menú de validación para permitir al usuario ingresar los datos de entrada necesarios para la ejecución de la verificación de la calidad de la clasificación.	M
SR-INT-16	La interfaz de usuario del sistema *ANNIC* deberá proveer un botón 'validar' para permitir al usuario iniciar la ejecución de la verificación de clasificación.	M
SR-INT-17	La interfaz de usuario del sistema *ANNIC* deberá proveer ventana de resultados de validación donde se mostrarán la matriz de confusión y el coeficiente *Kappa* obtenidos tras finalizar la verificación de la clasificación.	M
SR-INT-18	La ventana de validación de resultados del sistema *ANNIC* deberá proveer un botón u otro mecanismo para permitir al usuario guardar los resultados de la verificación de la clasificación.	D

ID	Descripción	Prioridad
SR-INT-19	La interfaz relacionada a la operación de 'guardar' los resultados de la verificación de la clasificación dependerá de la interfaz de manejo de archivos nativa del sistema operativo donde se ejecute este producto *ANNIC*.	D

Tabla 9: Requerimientos de interfaces

3.3. Requisitos operacionales

ID	Descripción	Prioridad
SR-OP-01	El sistema operativo donde se ejecutará el producto *ANNIC* deberá tener pre-instalado *Python*.	M
SR-OP-02	El sistema operativo donde se ejecutará el producto *ANNIC* deberá tener pre-instaladas las librerías para necesarias para su correcto funcionamiento. A saber: 'neurolab', 'numpy', 'PIL' y 'csipy'.	M

Tabla 10: Requerimientos operacionales

3.4. Requisitos de documentación

ID	Descripción	Prioridad
SR-DOC-01	El desarrollo completo del sistema deberá seguir los estándares ESA para pequeños proyectos.	M

Tabla 11: Requerimientos de documentación

3.5. Requisitos de entorno-portabilidad

ID	Descripción	Prioridad
SR-ENV-01	El sistema deberá ejecutarse en el sistema operativo *Linux*.	D
SR-ENV-02	El sistema deberá ejecutarse en el sistema operativo *Windows*.	M
SR-ENV-03	El sistema deberá ejecutarse en el sistema operativo *MAC*.	D

Tabla 12: Requerimientos de entorno-portabilidad

4. Diseño del sistema

El **diseño** del sistema de clasificación *ANNIC* representa la estrategia para resolver y construir la solución deseada dada la necesidad de explorar algoritmos de clasificación de imágenes no tradicionales. Éste incluye tanto decisiones acerca de la organización del sistema en subsistemas, y de los subsistemas en módulos con funcionalidades específicas, como así también la selección de una aproximación para la administración de almacenamiento de datos y la interacción de todas sus partes.

4.1. Método de diseño

El diseño de este software se basará en el estándar **UML**, del inglés *Unified Modeling Language*: Lenguaje de Modelado Unificado. Esta elección se debe a la expresividad que provee el lenguaje para representar gráficamente, analizar, comprender y reflejar la solución propuesta.

UML proporciona un vocabulario y conjunto de reglas para combinar sus palabras con el objetivo de posibilitar la comunicación. Al ser un lenguaje de modelado su vocabulario y reglas se centran en la representación conceptual y física de un sistema.

El vocabulario y las reglas del lenguaje UML indican cómo crear y leer modelos bien formados. Sin embargo, no determina qué modelos se deben diseñar. Esta es la tarea del proceso de desarrollo de software y se adapta según la necesidad de los distintos proyectos. En particular, en el sistema *ANNIC* será necesario definir un *diagrama de componentes* y detallar los *diagrama de secuencias* relacionados al las funcionalidades principales del producto. Con ellos se logrará un correcto entendimiento de este software.

4.2. Descripción de la descomposición

En esta sección se describirá el modelo físico de la solución propuesta para el desarrollo del sistema de clasificación de imágenes *ANNIC*.

4.3. Componentes

La organización global del software se puede resumir en la figura 2, donde se distinguen 3 subsistemas principales:

- El **subsistema de control**: Es el subsistema principal del producto. Contiene la *unidad de control*, la cual hará efectiva la interacción con el *usuario*, le proveerá todas las posibles acciones a concretarse mediante la utilización del software, procesará las entradas provistas por él, determinará la ejecución de los distintos componentes en base a sus necesidades, proporcionará un nivel de abstracción adecuando para permitir la interacción con el almacenamiento de archivos (*colección de imágenes*) y visualizará las salidas generadas tras la ejecución de los distintos algoritmos.
- El **subsistema de clasificación de imagen**: Está integrado por los módulos *Perceptrón*, *SOM* y *K-means*. Su responsabilidad será clasificar una imagen de acuerdo al algoritmo deseado: *RNA Perceptrón*, *RNA SOM* o el método *K-means*.

- El **subsistema de validación de la clasificación**: Está compuesto por un único módulo denominado *validador*. Su funcionalidad principal será verificar la clasificación realizada por alguno de los componentes del subsistema de clasificación.

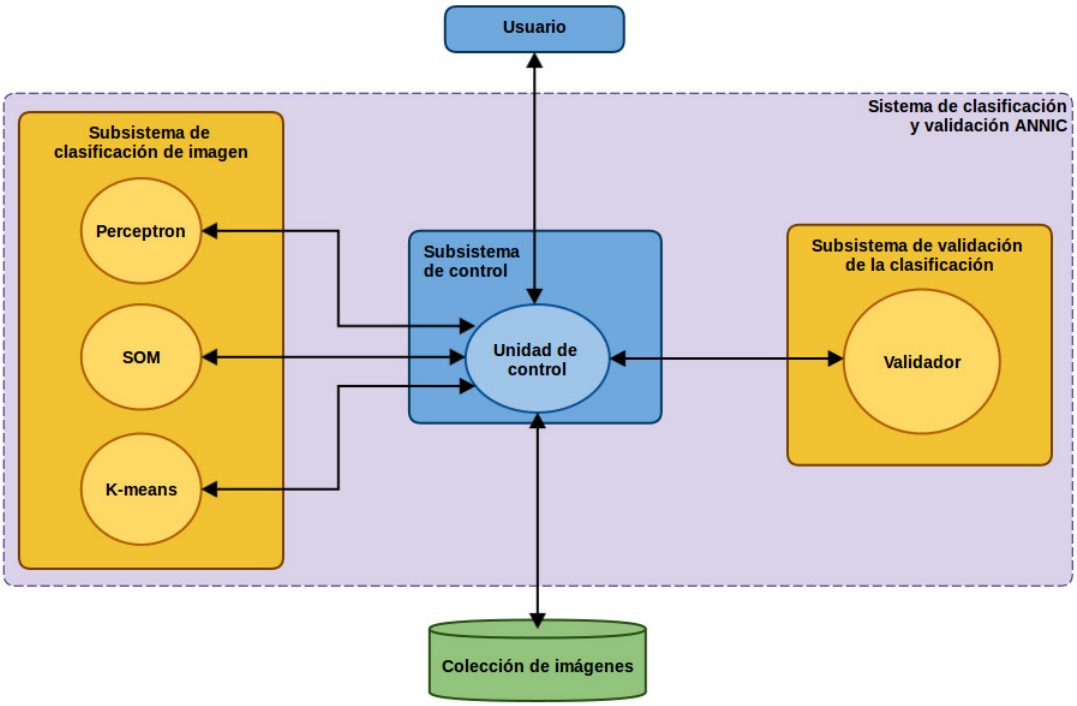


Figura 2: Modelo lógico

Cada subsistema y cada módulo proporcionará una interfaz bien definida, la cual especificará la forma de interacción y el flujo de información con las demás partes del sistema.

La relación entre el subsistema de control con cualquiera de los módulos del subsistema de clasificación será de tipo *cliente-servidor*: el subsistema de control (cliente) conocerá la interfaz de cada módulo del subsistema de clasificación (servidor), pero los componentes del subsistema de clasificación no necesitarán conocer las interfaces del subsistema de control. El subsistema de clasificación se limitará a responder a la comunicación iniciada por el subsistema de control, en particular, se limitará retornar una imagen clasificada cuando se solicite la ejecución de alguno de los algoritmos de categorización disponibles.

De igual modo, la relación entre el subsistema de control con el subsistema de validación será *cliente-servidor*. En este caso, el subsistema de validación se focalizará en retornar el resultado de la verificación la clasificación de una imagen cuando el subsistema de control lo requiera.

4.3.1. Control y flujo de datos

Los principales flujos de información del sistema de clasificación de imágenes *ANNIC* se exhibirán y describirán a través de cuatro diagramas de secuencia relacionados a las

funcionalidades primordiales que el software provee: clasificación basada en una *RNA Perceptrón*, clasificación basada en una *RNA SOM*, clasificación basada en el tradicional algoritmo *K-means* y validación de la clasificación.

4.3.2. Diagrama de secuencia de clasificación basada en una RNA Perceptrón

El diagrama de secuencia relacionado a la funcionalidad de clasificar una imagen mediante el empleo de una *RNA Perceptrón* se muestra en la figura 3.

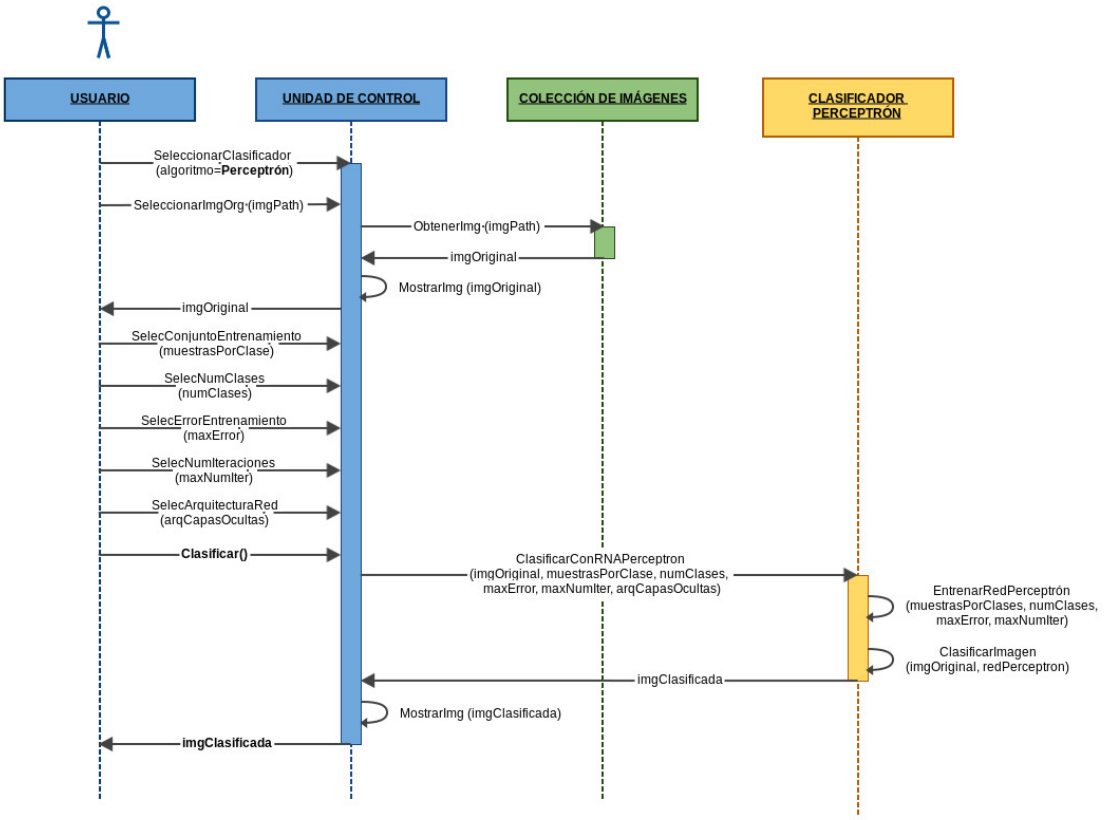


Figura 3: Diagrama de secuencia de clasificación basada en una RNA Perceptrón

Se puede observar que el *usuario* siempre y solamente interactuará con la *unidad de control*, es decir, con la interfaz gráfica del sistema. A su vez, este módulo será encargado de coordinar, con los demás subsistemas del producto, las distintas operaciones necesarias para satisfacer sus requerimientos.

El *usuario* deberá seleccionar el algoritmo de clasificación *Perceptrón* y proveer la ubicación física de la imagen a clasificar. Con este dato, la *unidad de control* obtendrá la imagen original de la base de datos '*colección de imágenes*' y la mostrará en pantalla.

Una vez que el *usuario* visualiza la imagen, seleccionará un conjunto de muestras de entrenamiento por clase, el número de categorías, el error máximo permitido, la cantidad máxima de iteraciones y la arquitectura de capas ocultas de la red neuronal.

Al terminar de identificar los parámetros necesarios para la clasificación, el *usuario* podrá iniciar el algoritmo. Así la unidad de control recibirá la orden *'clasificar'* y, en respuesta a ésta, invocará el método de categorización del *clasificador Perceptrón* proveyéndole los parámetros anteriormente elegidos.

Por su parte, el *clasificador Perceptrón* construirá y entrenará una *RNA Perceptrón* en base a las muestras de entrenamiento por clase, el número de categorías, el error máximo permitido y la cantidad máxima de iteraciones. Con esta *RNA* asignará una categoría a cada píxel de la imagen original, obteniendo así la imagen clasificada.

La imagen categorizada será el resultado que recibirá la *unidad de control*, módulo que finalmente se encargará de exponer esta solución al *usuario*.

4.3.3. Diagrama de secuencia de clasificación basada en una RNA SOM

El diagrama de secuencia relacionado a la funcionalidad de clasificar una imagen mediante el empleo de una *RNA SOM* se muestra en la figura 4.

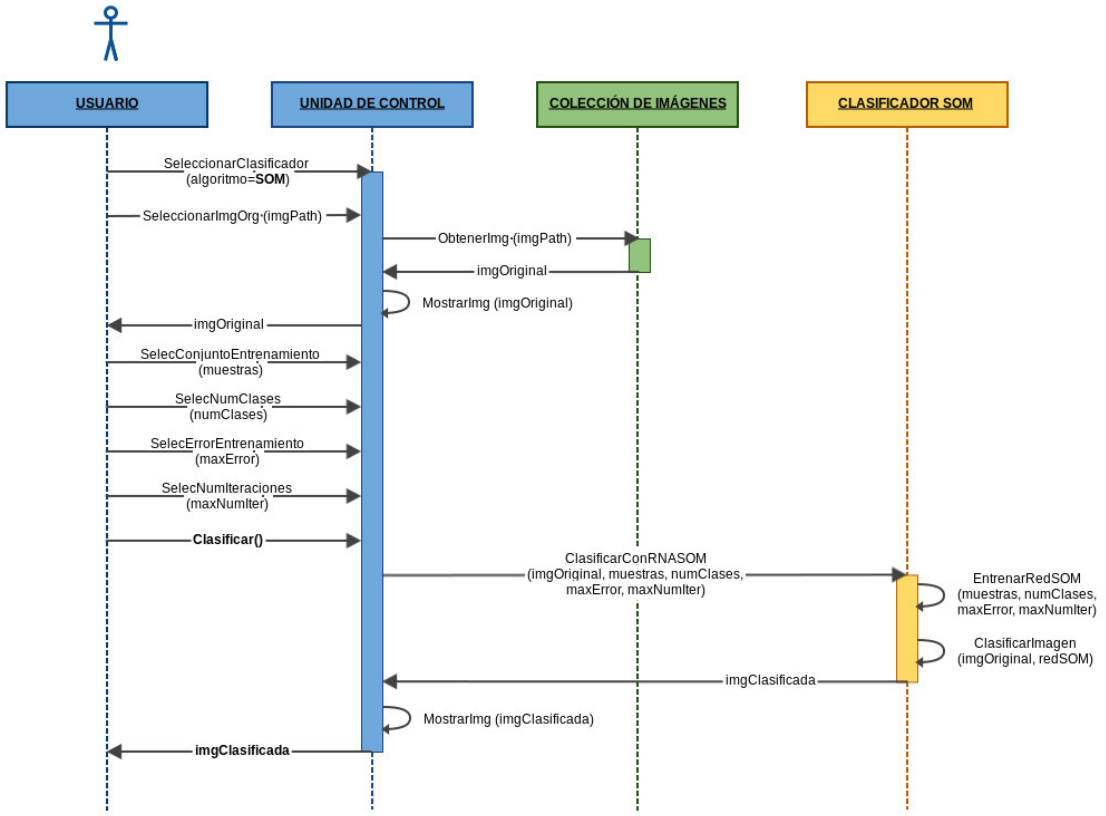


Figura 4: Diagrama de secuencia de clasificación basada en una RNA SOM

Se puede observar que el *usuario* siempre y solamente interactuará con la *unidad de control*, es decir, con la interfaz gráfica del sistema. A su vez, este módulo será encargado de coordinar, con los demás subsistemas del producto, las distintas operaciones necesarias para satisfacer sus requerimientos.

El *usuario* deberá seleccionar el algoritmo de clasificación *SOM* y proveer la ubicación física de la imagen a clasificar. Con este dato, la *unidad de control* obtendrá la imagen original de la base de datos '*colección de imágenes*' y la mostrará en pantalla.

Una vez que el *usuario* visualiza la imagen, seleccionará un conjunto de muestras de entrenamiento, el número de categorías, el error máximo de permitido y la cantidad máxima de iteraciones.

Al terminar de identificar los parámetros necesarios para la clasificación, el *usuario* podrá iniciar el algoritmo. Así la unidad de control recibirá la orden *'clasificar'* y, en respuesta a ésta, invocará el método de categorización del *clasificador SOM* proveyéndole los parámetros anteriormente elegidos.

Por su parte, el *clasificador SOM* construirá y entrenará una *RNA SOM* en base a las muestras de entrenamiento, el número de categorías, el error máximo permitido y la cantidad máxima de iteraciones. Con esta *RNA* (o *mapa auto-organizativo*) asignará una categoría a cada píxel de la imagen original, obteniendo así la imagen clasificada.

La imagen categorizada será el resultado que recibirá la *unidad de control*, módulo que finalmente se encargará de exponer esta solución al *usuario*.

4.3.4. Diagrama de secuencia de clasificación K-means

El diagrama de secuencia relacionado a la funcionalidad de clasificar una imagen mediante el algoritmo *K-means* se muestra en la figura 5.

Se puede observar que el *usuario* siempre y solamente interactuará con la *unidad de control*, es decir, con la interfaz gráfica del sistema. A su vez, este módulo será encargado de coordinar, con los demás subsistemas del producto, las distintas operaciones necesarias para satisfacer sus requerimientos.

El *usuario* deberá seleccionar el algoritmo de clasificación *K-means* y proveer la ubicación física de la imagen a clasificar. Con este dato, la *unidad de control* obtendrá la imagen original de la base de datos '*colección de imágenes*' y la mostrará en pantalla.

Una vez que el *usuario* visualiza la imagen, seleccionará un conjunto de muestras de entrenamiento, el número de categorías, el error máximo de permitido y la cantidad máxima de iteraciones.

Al terminar de identificar los parámetros necesarios para la clasificación, el *usuario* podrá iniciar el algoritmo. Así la unidad de control recibirá la orden *'clasificar'* y, en respuesta a ésta, invocará el método de categorización del *clasificador K-means* proveyéndole los parámetros anteriormente elegidos.

Por su parte, el *clasificador K-means* encontrará K centroides en base a las muestras de entrenamiento, el número de categorías, el error máximo permitido y la cantidad máxima de iteraciones. Con los centroides definidos, asignará una categoría a cada píxel de la imagen original, obteniendo así la imagen clasificada.

La imagen categorizada será el resultado que recibirá la *unidad de control*, módulo que finalmente se encargará de exponer esta solución al *usuario*.

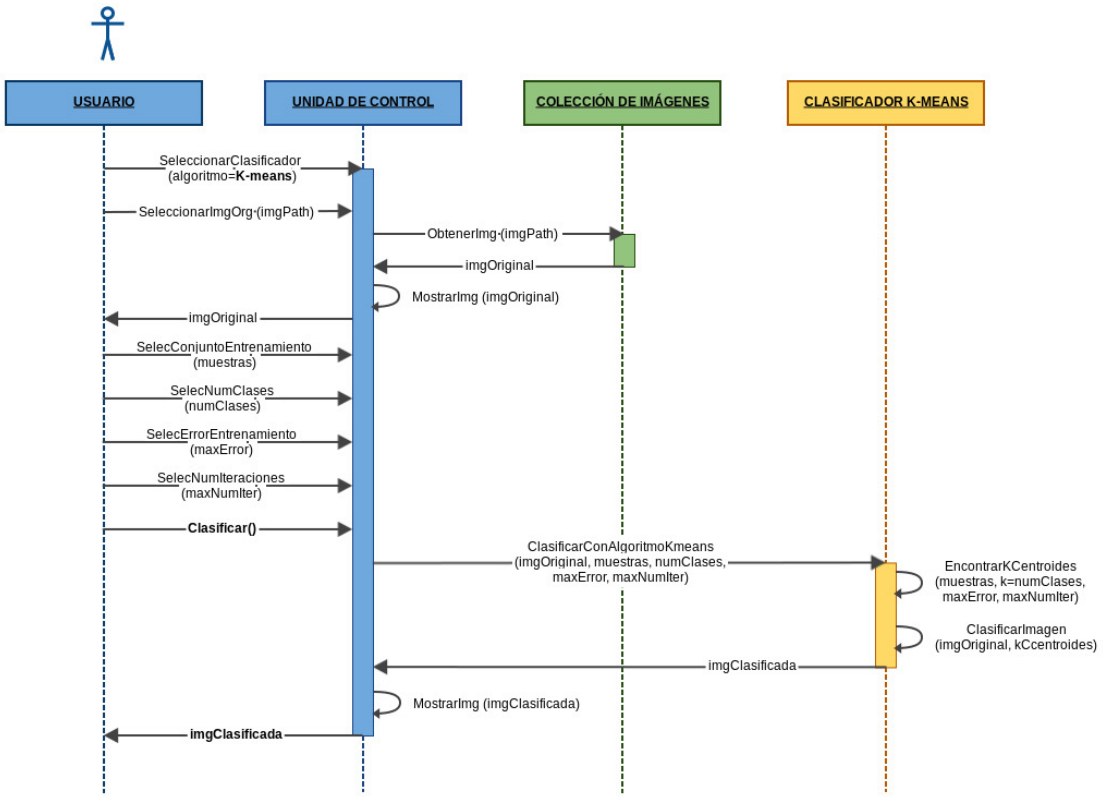


Figura 5: Diagrama de secuencia de clasificación K-means

4.3.5. Diagrama de secuencia de validación de la clasificación

El diagrama de secuencia relacionado a la funcionalidad validar la clasificación se muestra en la figura 6. Es preciso destacar que para iniciar este flujo de datos deberá existir una imagen previamente clasificada con alguno de los algoritmos provistos por el software.

Se puede observar que el *usuario* siempre y solamente interactuará con la *unidad de control*, es decir, con la interfaz gráfica del sistema. A su vez, este módulo será encargado de coordinar, con los demás subsistemas del producto, las distintas operaciones necesarias para satisfacer sus requerimientos.

El *usuario* deberá seleccionar el menú de validación. Este menú le permitirá elegir un conjunto de muestras conocidas por clase sobre la imagen original e iniciar el algoritmos de verificación de la clasificación.

Una vez iniciada la verificación, la *unidad de control* recibirá la orden *'validar'* y, en respuesta a ésta, invocará el método de verificación del *validador* proveyéndole las muestras seleccionadas y la imagen clasificada.

Por su parte, el *validador* calculará la *matriz de confusión* considerando los parámetros anteriormente mencionados. Con está matriz, luego computará el *coeficiente Kappa*.

Finalmente la *unidad de control* recibirá los resultandos de la verificación de la clasificación (*matriz de confusión* y *coeficiente Kappa*) y los mostrará en pantalla al *usuario*.

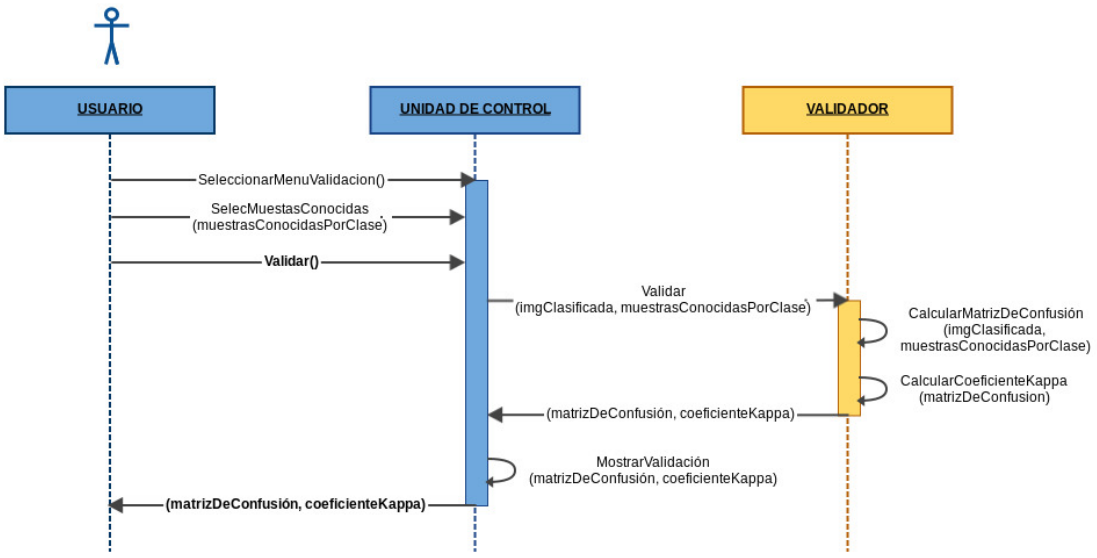


Figura 6: Diagrama de secuencia de validación de la clasificación

5. Descripción de los componentes

5.1. Componente 1: Unidad de control

5.1.1. Tipo

La *unidad de control* será un **módulo** del sistema *ANNIC* encargado de proveer la interacción entre el sistema y el usuario para permitir la administración sus distintas funcionalidades.

5.1.2. Función

La función principal de este componente será la comunicación con el usuario. Deberá permitirle utilizar todas las funcionalidades presentes en el sistema de clasificación *ANNIC* y proveer los datos necesarios para la procesamiento de éstas.

Otra responsabilidad de la *unidad de control* será iniciar la ejecución de los demás componentes, procesar y mostrar su salida de acuerdo a la acción solicitada por el usuario.

5.1.3. Interfaces

La interacción con el usuario se llevará a cabo mediante una interfaz gráfica, que a su vez proveerá interfaces específicas para las distintas funcionalidad del sistema de clasificación *ANNIC*. Estas se detallan a continuación:

- Menú de manejo de archivos: Permitirá al usuario abrir la imagen a clasificar y guardar la imagen clasificada.

- Menú de selección de clasificador: Permitirá al usuario seleccionar el algoritmo de clasificación deseado, para luego hacer posible la interacción mediante el menú de clasificación correspondiente: Perceptrón, SOM o K-means.
- Menú de clasificación Perceptrón: Permitirá al usuario seleccionar el número de clases, tomar muestras por clase sobre la imagen original, seleccionar el error máximo y el número máximo de iteraciones permitidas en el entrenamiento de la *RNA Perceptrón*, iniciar el algoritmo de clasificación y guardar la imagen categorizada tras finalizar el proceso.
- Menú de clasificación SOM: Permitirá al usuario seleccionar el número de clases, tomar un conjunto de muestras representativas sobre la imagen original, seleccionar el error máximo y el número máximo de iteraciones permitidas en el entrenamiento de la *RNA SOM*, iniciar el algoritmo de clasificación y guardar la imagen categorizada tras finalizar el proceso.
- Menú de clasificación K-means: Permitirá al usuario seleccionar el número de clases, tomar un conjunto de muestras representativas sobre la imagen original, seleccionar el error máximo y el número máximo de iteraciones permitidas en el algoritmo *K-menas*, iniciar el algoritmo de clasificación y guardar la imagen categorizada tras finalizar el proceso.
- Menú de validación: Permitirá al usuario seleccionar el número de clases, tomar muestras conocidas por clase sobre la imagen original e iniciar el proceso de verificación de la clasificación.
- Marco de imagen original: Permitirá al usuario visualizar la imagen a clasificar o imagen original.
- Marco de imagen clasificada: Permitirá al usuario visualizar la imagen clasificada.
- Ventana de resultados de validación: Permitirá al usuario visualizar la matriz de confusión y el coeficiente *Kappa*

5.1.4. Dependencias

La elaboración de la *unidad de control* podrá ser independiente de la elaboración de los demás módulos del sistema.

Sin embargo, para el funcionamiento global del sistema de clasificación *ANNIC* se observan las siguientes dependencias:

- La interfaces relacionadas a las opciones del *menú de manejo de archivos* dependerán de la interfaz de manejo de archivos nativa del sistema operativo donde se ejecute este producto,
- La imagen mostrada en el *marco de imagen clasificada* dependerá del resultado de la ejecución del clasificador seleccionado por el usuario, y
- Los datos mostrados en la *ventana de resultado* dependerán de la salida de la ejecución del módulo de validación.

5.1.5. Procesamiento

El procesamiento principal que llevará a cabo la *unidad de control* se puede especificar en base las dos funcionalidades primordiales del sistema *ANNIC*:

- Clasificación: La *unidad de control* permitirá al usuario seleccionar el clasificador deseado a través del *menú de selección de clasificador*.

  En base a esta opción mostrará el *menú de clasificación* correspondiente, el cual facilitará la recolección de los datos necesarios para la categorización: número de clases, muestras representativas, y error máximo y número máximo de iteraciones permitidas en la fase de entrenamiento. En el caso de una clasificación en base a una *RNA Perceptrón* también aceptará el ingreso de la arquitectura de capas ocultas de la red.

  Con los datos anteriormente especificados, la *unidad de control* dará comienzo al *componente de clasificación* adecuado (*clasificador Perceptrón*, *clasificador SOM* o *clasificador K-means*) y visualizará su resultado en el *marco de imagen clasificada*.

- Validación: La *unidad de control* permitirá la recolección de los datos necesarios para verificar la calidad de la clasificación: número de clases, muestras conocidas por clase, e imagen clasificada. Luego ejecutará el *componente de validación* y visualizará sus resultados en la *ventana de resultados de validación*.

En cuanto las funcionalidades secundarias de este software, a través del *menú de manejo de archivos*, será posible:

- Abrir una imagen: La *unidad de control* permitirá al usuario seleccionar la imagen a clasificar a través de la interfaz de manejo de archivos nativa del sistema operativo donde se ejecute este producto y visualizará esta selección en el *marco de imagen original*.

- Guardar resultados: La *unidad de control* permitirá al usuario seleccionar el archivo de destino en el cual guardar el resultado de la verificación de la clasificación a través de la interfaz de manejo de archivos nativa del sistema operativo donde se ejecute este producto.

5.2. Componente 2: Clasificador Perceptrón

5.2.1. Tipo

El *clasificador Perceptrón* será un **módulo** del sistema *ANNIC* encargado de la clasificación de imágenes.

5.2.2. Función

La función principal de este componente será la clasificación de una imagen digital utilizando una *RNA Perceptrón* durante la etapa de entrenamiento del algoritmo de clasificación.

5.2.3. Interfaces

Este módulo expondrá una función con las siguientes características:

- **Entradas**:
  - Imagen original,
  - Conjunto de píxeles de entrenamiento por clase,
  - Número de clases a diferenciar,
  - Error máximo permitido en el entrenamiento de la *RNA Perceptrón*,
  - Número máximo de iteraciones permitidas en el entrenamiento de la *RNA Perceptrón*, y
  - Arquitectura de capas ocultas para construir la *RNA Perceptrón*
- **Salida**:
  - Imagen clasificada tras la ejecución del algoritmo de clasificación basado en una *RNA Perceptrón*.

5.2.4. Dependencias

El comportamiento del componente *clasificador Perceptrón* será independiente, porque su diseño permitirá reutilizar este módulo en otros productos de software.

Sin embargo, en el sistema de clasificación *ANNIC*, el momento en el cual comenzar la ejecución del componente dependerá de la *unidad de control* ya que esta proveerá al usuario el mecanismo para decidir el inicio la clasificación basada en una *RNA Perceptrón*.

5.2.5. Procesamiento

El *clasificador Perceptrón* construirá una *RNA Perceptrón* tal qué:

- La cantidad de neuronas de la capa de entrada será igual a la cantidad de niveles de grises de la imagen original,
- La cantidad de capas ocultas y su estructura corresponderán con la arquitectura de capas ocultas provistas por el usuario, y
- La capa de salida tendrá una única neurona cuyos posibles valores serán decimales pertenecientes intervalo $[0, 1]$.

Luego esta red será entrenada en base al conjunto de píxeles de entrenamiento, considerando el error máximo permitido y la cantidad máxima de iteraciones. A cada subconjunto por clase de entrenamiento se le asociará un número decimal mayor o igual a 0 y menor o igual a 1, tal que corresponde con una única representación numérica de la clase asociada al conjunto.

La representación numérica de las clases se calcularán en base a la siguiente fórmula, para asegurar equidistancia entre estas:

$$Clase(i) = i/(numeroDeClases - 1)$$

donde:

- $0 <= i <= numeroDeClases$

Concluida la fase de entrenamiento, se clasificará cada píxel de la imagen original 'introduciéndolo' a la *RNA*, obteniendo así un número de salida asociado a ese píxel.

La clase del píxel será aquella cuya representación numérica sea más cercana al decimal obtenido en la capa de salida de la red.

5.3. Componente 3: Clasificador SOM

5.3.1. Tipo

El *clasificador SOM* será un **módulo** del sistema *ANNIC* encargado de la clasificación de imágenes.

5.3.2. Función

La función principal de este componente será la clasificación de una imagen digital utilizando una *RNA SOM* durante la etapa de entrenamiento del algoritmo de clasificación.

5.3.3. Interfaces

Este módulo expondrá una función con las siguientes características:

- **Entradas**:
  - Imagen original,
  - Conjunto de píxeles de entrenamiento,
  - Número de clases a diferenciar,
  - Error máximo permitido en el entrenamiento de la *RNA SOM*, y
  - Número máximo de iteraciones permitidas en el entrenamiento de la *RNA SOM*.
- **Salida**:
  - Imagen clasificada tras la ejecución del algoritmo de clasificación basado en una *RNA SOM*.

5.3.4. Dependencias

El comportamiento del componente *clasificador SOM* será independiente, porque su diseño permitirá reutilizar este módulo en otros productos de software.

Sin embargo, en el sistema de clasificación *ANNIC*, el momento en el cual comenzar la ejecución del componente dependerá de la *unidad de control* ya que esta proveerá al usuario el mecanismo para decidir el inicio la clasificación basada en una *RNA SOM*.

5.3.5. Procesamiento

El *clasificador SOM* entrenará una *RNA SOM* en base al conjunto de píxeles de entrenamiento, considerando el error máximo permitido y la cantidad máxima de iteraciones.

Concluida la fase de entrenamiento, se tendrá red neuronal tal que a cada una de sus neuronas se le asociará una clase en particular.

Finalmente, se clasificarán cada píxel de la imagen original 'introduciéndolo' a la *RNA*. Solo existirá una neurona ganadora, la neurona cuyo vector de pesos se encuentre más cerca al píxel de entrada.

Dado que un píxel puede ser considerado un vector, la forma de determinar la neurona ganadora será calculando la distancia euclidiana entre el vector de entrada y los vectores de pesos de las neuronas.

El píxel será clasificado con la clase asociada a la neurona ganadora.

5.4. Componente 4: Clasificador K-means

5.4.1. Tipo

El *clasificador K-means* será un **módulo** del sistema *ANNIC* encargado de la clasificación de imágenes.

5.4.2. Función

La función principal de este componente será la clasificación de una imagen digital utilizando el algoritmo *K-means* tanto para determinar K centroides durante la etapa de entrenamiento, como para la asignación de una clase a cada píxel.

5.4.3. Interfaces

Este módulo expondrá una función con las siguientes características:

- **Entradas**:
  - Imagen original,
  - Conjunto de píxeles de entrenamiento,
  - Número de clases a diferenciar,
  - Error máximo permitido en el entrenamiento del algoritmo *K-means*, y
  - Número máximo de iteraciones permitidas en el entrenamiento del algoritmo *K-means*.
- **Salida**:
  - Imagen clasificada tras la ejecución del algoritmo de clasificación basado en el método *K-means*.

5.4.4. Dependencias

El comportamiento del componente *clasificador K-means* será independiente, porque su diseño permitirá reutilizar este módulo en otros productos de software.

Sin embargo, en el sistema de clasificación *ANNIC*, el momento en el cual comenzar la ejecución del componente dependerá de la *unidad de control* ya que esta proveerá al usuario el mecanismo para decidir el inicio la clasificación basada en el algoritmo *K-means*.

5.4.5. Procesamiento

El *clasificador K-means* encontrará K centrodides en base al conjunto de píxeles de entrenamiento y al número de clases, considerando el error máximo permitido y la cantidad máxima de iteraciones.

Por lo tanto, concluida la fase de entrenamiento, se tendrán K centroides que representan a cada una de las clases. Es posible que el número de centroides encontrados sea menor al número de clases, en cuyo caso habrá categorías no representadas.

Finalmente, se clasificarán la imagen original de acuerdo a la metodología de asignación del algoritmo *K-means*, donde cada píxel será adscripto al grupo con la media más cercana (tomando en cuenta el centroide definido para este grupo).

El píxel será clasificado con la clase asociada al grupo perteneciente.

5.5. Componente 5: Validador

5.5.1. Tipo

El componente de validación será un **módulo** del sistema *ANNIC* encargado de la verificación de la clasificación de imágenes.

5.5.2. Función

La función principal de este componente será la verificación de la clasificación de una imagen digital utilizando el cálculo de una matriz de confusión y el coeficiente *Kappa* asociado a esta.

5.5.3. Interfaces

Este módulo expondrá una función con las siguientes características:

- **Entradas**:
  - Imagen clasificada, y
  - Muestras conocidas por clase, es decir, secciones de la imagen original para las cuales se conoce la clase real de los píxeles que la componen.
- **Salida**:
  - Matriz de confusión, y
  - Coeficiente Kappa.

5.5.4. Dependencias

El comportamiento del componente de validación será independiente, porque su diseño permitirá reutilizar este módulo en otros productos de software.

Sin embargo, en el sistema de clasificación *ANNIC*, la obtención de la imagen clasificada dependerá de la ejecución previa de alguno de los algoritmos de categorización provistos por el producto; y el momento en el cual comenzar la ejecución del componente de validación dependerá de la *unidad de control* ya que esta proveerá al usuario el mecanismo para decidir el inicio la verificación de la clasificación.

5.5.5. Procesamiento

El módulo de validación calculará una *matriz de confusión* para resumir la información acerca de las clases reales o conocidas y las predicciones realizadas por el sistema de clasificación empleado.

Cada columna de la matriz representará los casos que el algoritmo predijo, mientras que cada fila reflejará los casos en una clase real. La diagonal de la matriz expresará el número de puntos de verificación en donde se produce un acuerdo entre las dos fuentes, mientras que los marginales mostrarán errores de asignación (errores de omisión y errores de comisión).

Por otra parte, una vez obtenida la matriz de confusión, el módulo de validación computará el *coeficiente Kappa*. Este mide la diferencia entre las coincidencias que observan en la diagonal de la matriz y las que se esperarían simplemente por azar.

6. Matriz de trazabilidad de Requisitos de Usuario frente a Requisitos de Software

En la siguiente tabla se muestra la correspondencia entre los requisitos de usuario y los requisitos de software que los resuelven.

ID Requerimiento de Usuario	ID Requerimiento de Software
UR-ENV-01	SR-ENV-01
UR-ENV-02	SR-ENV-02
UR-ENV-03	SR-ENV-03
UR-ENV-04	SR-INT-01
UR-ENV-05	SR-INT-02
UR-GEN-01	SR-GEN-01, SR-GEN-02, SR-INT-07, SR-INT-08, SR-INT-10, SR-INT-12, SR-OP-01, SR-OP-02
UR-GEN-02	SR-GEN-03, SR-INT-15
UR-PER-01	SR-PER-01, SR-PER-02, SR-PER-03, SR-INT-03, SR-INT-04, SR-INT-05, SR-OP-01, SR-OP-02
UR-PER-02	SR-PER-04, SR-INT-06
UR-PER-03	SR-PER-05, SR-PER-06, SR-PER-07
UR-PER-04	SR-PER-08, SR-PER-09, SR-PER-10

ID Requerimiento de Usuario	ID Requerimiento de Software
UR-PER-05	SR-PER-11, SR-PER-12, SR-PER-13, SR-PER-14, SR-PER-15
UR-PER-06	SR-PER-16, SR-PER-17, SR-PER-18, SR-PER-19, SR-PER-20
UR-PER-07	SR-PER-21, SR-PER-22, SR-PER-23, SR-PER-24, SR-PER-25, SR-INT-09
UR-PER-08	SR-PER-26, SR-PER-27, SR-INT-14
UR-PER-09	SR-PER-28, SR-PER-29, SR-PER-30, SR-INT-03, SR-INT-04, SR-INT-05
UR-SOM-01	SR-SOM-01, SR-SOM-02, SR-SOM-03, SR-INT-03, SR-INT-04, SR-INT-05
UR-SOM-02	SR-SOM-04, SR-INT-06
UR-SOM-03	SR-SOM-05, SR-SOM-06, SR-SOM-07
UR-SOM-04	SR-SOM-08, SR-SOM-09, SR-SOM-10
UR-SOM-05	SR-SOM-11, SR-SOM-12, SR-SOM-13, SR-SOM-14, SR-SOM-15
UR-SOM-06	SR-SOM-16, SR-SOM-17, SR-SOM-18, SR-SOM-19, SR-SOM-20, SR-INT-11
UR-SOM-07	SR-SOM-21, SR-SOM-22, SR-INT-14
UR-SOM-08	SR-SOM-23, SR-SOM-24, SR-SOM-25, SR-INT-03, SR-INT-04, SR-INT-05
UR-KMA-01	SR-KMA-01, SR-KMA-02, SR-KMA-03, SR-INT-03, SR-INT-04, SR-INT-05
UR-KMA-02	SR-KMA-04, SR-INT-06
UR-KMA-03	SR-KMA-05, SR-KMA-06, SR-KMA-07
UR-KMA-04	SR-KMA-08, SR-KMA-09, SR-KMA-10
UR-KMA-05	SR-KMA-11, SR-KMA-12, SR-KMA-13, SR-KMA-14, SR-KMA-15
UR-KMA-06	SR-KMA-16, SR-KMA-17, SR-KMA-18, SR-KMA-19, SR-KMA-20, SR-INT-13

ID Requerimiento de Usuario	ID Requerimiento de Software
UR-KMA-07	SR-KMA-21, SR-KMA-22, SR-INT-14
UR-KMA-08	SR-KMA-23, SR-KMA-24, SR-KMA-25, SR-INT-03, SR-INT-04, SR-INT-05
UR-VAL-01	SR-VAL-01, SR-VAL-02, SR-VAL-03, SR-VAL-04
UR-VAL-02	SR-VAL-05, SR-VAL-06, SR-VAL-07, SR-VAL-08
UR-VAL-03	SR-VAL-09, SR-VAL-10, SR-VAL-11, SR-INT-16
UR-VAL-04	SR-VAL-12, SR-INT-17
UR-VAL-05	SR-VAL-13, SR-VAL-14, SR-INT-18, , SR-INT-19
UR-RES-01	SR-DOC-01

Tabla 13: Matriz de trazabilidad de Requisitos de Usuario frente a Requisitos de Software

7. Matriz de Trazabilidad de Requisitos de Software frente a Componentes

En la siguiente tabla se muestra la correspondencia entre los requisitos software y los componentes que los soportan.

ID Requerimiento de Software	Componente
SR-INT-01, SR-INT-02, SR-INT-03, SR-INT-04, SR-INT-05, SR-INT-06, SR-INT-07, SR-INT-08, SR-INT-09, SR-INT-10, SR-INT-11, SR-INT-12, SR-INT-13, SR-INT-14, SR-INT-15, SR-INT-16, SR-INT-17, SR-INT-18, SR-INT-19, SR-GEN-01, SR-GEN-02, SR-GEN-03, SR-OP-01, SR-OP-02, SR-DOC-01, SR-ENV-01, SR-ENV-02, SR-ENV-03	Componente 1: Unidad de control

ID Requerimiento de Software	Componente
SR-PER-01, SR-PER-02, SR-PER-03, SR-PER-04, SR-PER-05, SR-PER-06, SR-PER-07, SR-PER-08, SR-PER-09, SR-PER-10, SR-PER-11, SR-PER-12, SR-PER-13, SR-PER-14, SR-PER-15, SR-PER-16, SR-PER-17, SR-PER-18, SR-PER-19, SR-PER-20, SR-PER-21, SR-PER-22, SR-PER-23, SR-PER-24, SR-PER-25, SR-PER-26, SR-PER-27, SR-PER-28, SR-PER-29, SR-PER-30, SR-OP-01, SR-OP-02, SR-DOC-01, SR-ENV-01, SR-ENV-02, SR-ENV-03	Componente 2: Clasificador Perceptrón
SR-SOM-01, SR-SOM-02, SR-SOM-03, SR-SOM-04, SR-SOM-05, SR-SOM-06, SR-SOM-07, SR-SOM-08, SR-SOM-09, SR-SOM-10, SR-SOM-11, SR-SOM-12, SR-SOM-13, SR-SOM-14, SR-SOM-15, SR-SOM-16, SR-SOM-17, SR-SOM-18, SR-SOM-19, SR-SOM-20, SR-SOM-21, SR-SOM-22, SR-SOM-23, SR-SOM-24, SR-SOM-25, SR-OP-01, SR-OP-02, SR-DOC-01, SR-ENV-01, SR-ENV-02, SR-ENV-03	Componente 3: Clasificador SOM
SR-KMA-01, SR-KMA-02, SR-KMA-03, SR-KMA-04, SR-KMA-05, SR-KMA-06, SR-KMA-07, SR-KMA-08, SR-KMA-09, SR-KMA-10, SR-KMA-11, SR-KMA-12, SR-KMA-13, SR-KMA-14, SR-KMA-15, SR-KMA-16, SR-KMA-17, SR-KMA-18, SR-KMA-19, SR-KMA-20, SR-KMA-21, SR-KMA-22, SR-KMA-23, SR-KMA-24, SR-KMA-25, SR-OP-01, SR-OP-02, SR-DOC-01, SR-ENV-01, SR-ENV-02, SR-ENV-03	Componente 4: Clasificador K-means
SR-VAL-01, SR-VAL-02, SR-VAL-03, SR-VAL-04, SR-VAL-05, SR-VAL-06, SR-VAL-07, SR-VAL-08, SR-VAL-09, SR-VAL-10, SR-VAL-11, SR-VAL-12, SR-VAL-13, SR-VAL-14, SR-OP-01, SR-OP-02, SR-DOC-01, SR-ENV-01, SR-ENV-02, SR-ENV-03	Componente 5: Validador

Tabla 14: Matriz de Trazabilidad de Requisitos de Software frente a Componentes

APÉNDICE C

Manual de Usuario del Software

User Manual Document
OF
ANNIC

Florencia Mihaich

(Spanish Version)
Revisión: 0.1

19 de mayo de 2014

I. Historial de revisiones

Versión	Fecha	Autor	Resumen de cambios
1.0	19/05/2014	Mihaich, Florencia	Primera versión del documento.

Tabla 1: Historial de revisiones

II. Documentos relacionados

ID	Nombre	Fecha	Autor
URD	*User Requirement Document of ANNIC*. (Documento de requerimientos de usuario del sistema de clasificación de imágenes *ANNIC*).	09/04/2013	Mihaich, Florencia
SRD	*Software Requirement Document of ANNIC*. (Documento de requerimientos de software del sistema de clasificación de imágenes *ANNIC*).	03/05/2013	Mihaich, Florencia
BSSC96	Guía para la aplicación de estándares de Ingeniería de Software ESA (Agencia Espacial Europea) para proyectos de software pequeños.	1996	ESA Comité de Estandarización y Control de Software (BSSC)

Tabla 2: Documentos relacionados

III. Tabla de contenidos

1. Introducción

1.1. Destinatarios

El sistema de clasificación de imágenes ***ANNIC*** (*Artificial Neural Network Image Classification*) está dirigido a todo usuario con conocimiento en métodos de categorización de imágenes digitales.

1.2. Aplicabilidad

Este manual es aplicable a la *versión 1.0* del software de clasificación de imágenes *ANNIC*.

1.3. Propósito

El propósito de este manual es proporcionar al usuario la información necesaria para utilizar el sistema de clasificación de imágenes *ANNIC*, el cual expone algoritmos de clasificación no convencionales y la posibilidad de comprarlos con otros ampliamente conocidos.

1.4. Cómo usar este documento

Con la finalidad de describir, a los usuarios finales, el sistema de clasificación de imágenes *ANNIC* y su forma de uso, este documento se estructura en las siguientes secciones:

- Sección 1: Provee una primera aproximación de este manual. Define el grupo de personas a las cuales está dirigido y detalla el modo de explorar el contenido de esta documentación.
- Sección 2: Expone nociones generales acerca del software y cómo utilizar el producto.
- Sección 3: Especifica los procedimientos necesarios para el correcto funcionamiento sistema en la máquina de destino.
- Sección 4: Detalla como ejecutar cada una de las funcionalidades provistas por el producto.

2. Descripción general

El sistema de clasificación de imágenes *ANNIC* fue diseñado con el fin de explorar algoritmos de clasificación no tradicionales que involucran la utilización de redes neuronales artificiales y poder comparar sus resultados y eficiencia con respecto a algoritmos de categorización ampliamente conocidos.

Con este objetivo, el sistema *ANNIC* permite clasificar imágenes digitales utilizando alguno de los siguientes métodos supervisados: red neuronal Perceptrón, mapa autoorganizado de Kohonen (redes SOM) o el tradicional algoritmo k-meas.

A su vez, proporciona métodos de validación como cálculo de una matriz de confusión y determinación del coeficiente *Kappa*.

Durante la etapa de clasificación, el usuario es considerado el maestro o moderador del proceso. Se asume que posee conocimiento previo del área de estudio y, por lo tanto, que es capaz de delimitar regiones de identidad conocida sobre la imagen original y proporcionar todos los parámetros necesarios para hacer efectiva la clasificación.

En base a la información recibida y el algoritmo elegido, el software ejecutará el proceso de categorización correspondiente y mostrará como resultado una imagen clasificada.

Una vez generada la imagen de categorías, el rol del usuario es fundamental para verificar la calidad de la clasificación. En este punto es considerado el experto que determina las áreas de realidad representativas de cada clase. En base a éstas, el sistema podrá calcular y reflejar el nivel de certeza del algoritmo empleado.

3. Sección de instalación

Si bien el software de clasificación de imágenes *ANNIC* se ejecuta desde línea de comando (por lo tanto no es necesaria su instalación), en esta sección se pretende determinar los pre-requisitos necesarios para lograr su correcto funcionamiento. Ellos son:

- El sistema operativo donde se ejecute el producto deberá tener instalado ***Python***.
- El sistema operativo donde se ejecute el producto deberá tener pre-instaladas las siguiente librerías: *'neurolab'*, *'numpy'*, *'PIL'* y *'csipy'*.

Para instalar las librerías de terceros mencionadas anteriormente, es posible utilizar cualquiera de las modalidades proporcionadas por *Python* según la preferencia del usuario. En particular, dos formas sugeridas son *Distributable* o *pip*.

4. Descripción funcional

Considerando las distintas formas de clasificación provistas por este software y el mecanismo expuesto para su verificación, en esta sección se detalla cómo el usuario podrá acceder a cada una de estas funcionalidades.

4.1. Clasificación basada en una RNA Perceptrón

Para realizar una clasificación de imagen basada en una *RNA Perceptrón* se deberán seguir las siguientes instrucciones:

1. Abrir la imagen a clasificar:

1.1. Seleccionar *'Abrir imagen'* (*'Open image'*) dentro del menú de manejo de archivos (*'File menu'*).

1.2. Elegir la ubicación física del imagen original.

1.3. Presionar el botón *'Abrir'* (*'Open'*).

Figura 1: Abrir imagen original.

2. Seleccionar el clasificador Perceptrón (*'Perceptron Classification'*) dentro del menú de clasificación (*'Classify menu'*).

Figura 2: Selección de clasificador Perceptrón.

3. Definir los parámetros de clasificación a utilizar durante el entrenamiento de la red neuronal *Perceptrón*:

3.1. Seleccionar la cantidad de clases (*'Number of classes'*).

3.2. Tomar muestras conocidas por clase (*'Training samples'*) para utilizar durante el entrenamiento de la *RNA Perceptrón*.

- Para ello dibujar rectángulos sin levantar el cursor dentro de la imagen original.

3.3. Definir el máximo error permitido en el entrenamiento de la red neuronal (*'Training Error'*).

3.4. Elegir el número máximo de iteraciones permitidas en el entrenamiento de la red neuronal (*'Max Iteration'*).

3.5. Determinar la estructura de capas ocultas de la *RNA Perceptrón*:

3.5.a. Especificar el número de capas ocultas (*'Number of layers'*).

3.5.b. Fijar el número de neuronas disponibles en cada capa oculta (*'Number of neurons per layer'*).

4. Presionar el botón clasificar (*'Classify'*).

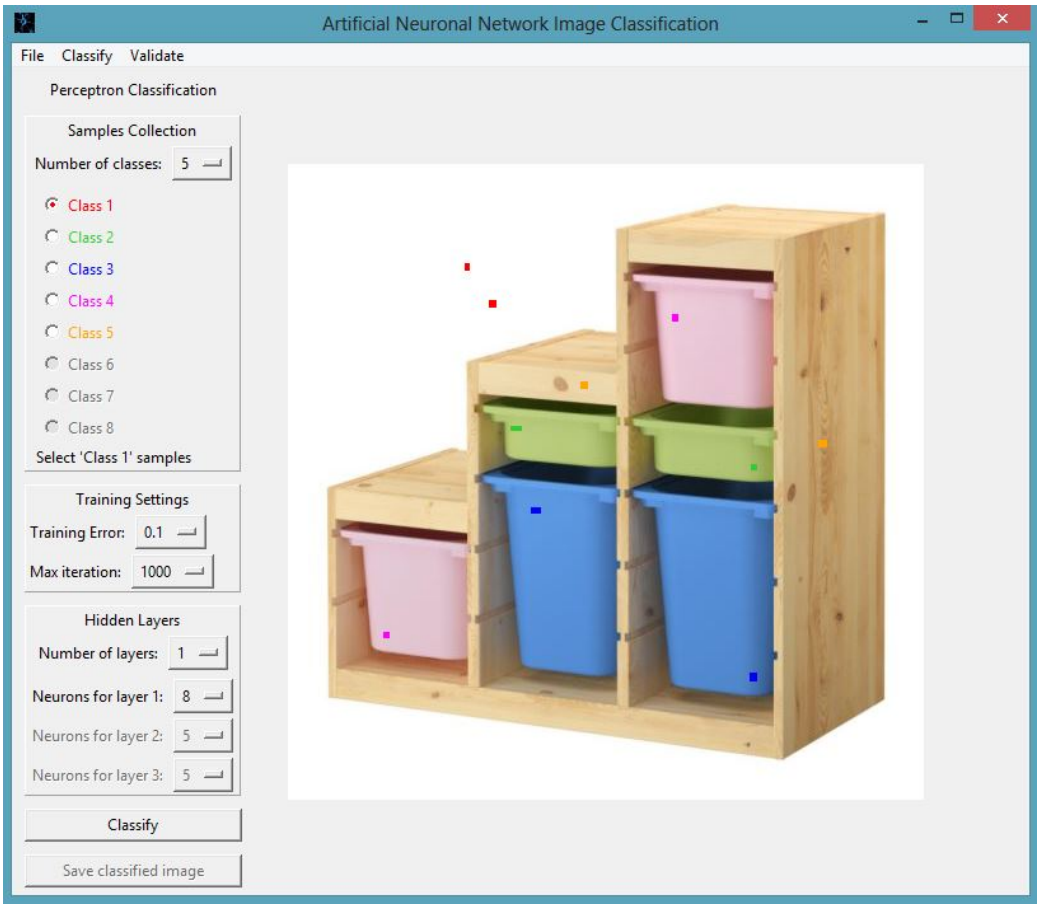


Figura 3: Menú de clasificación Perceptrón.

4.2. Clasificación basada en una RNA SOM

Para realizar una clasificación de imagen basada en una *RNA SOM* se deberán seguir las siguientes instrucciones:

1. Abrir la imagen a clasificar:
   - **1.1.** Seleccionar *'Abrir imagen'* (*'Open image'*) dentro del menú de manejo de archivos (*'File menu'*).
   - **1.2.** Elegir la ubicación física del imagen original.
   - **1.3.** Presionar el botón *'Abrir'* (*'Open'*).

Figura 4: Abrir imagen original.

2. Seleccionar el clasificador SOM (*'SOM Classification'*) dentro del menú de clasificación (*'Classify menu'*).

Figura 5: Selección de clasificador SOM.

3. Definir los parámetros de clasificación a utilizar durante el entrenamiento de la red neuronal *SOM*:

3.1. Seleccionar la cantidad de clases (*'Number of classes'*).

3.2. Tomar muestras conocidas (*'Training samples'*) para utilizar y hacer más eficiente el entrenamiento de la *RNA SOM*.

- Para ello dibujar rectángulos sin levantar el cursor dentro de la imagen original.

3.3. Definir el máximo error permitido en el entrenamiento de la red neuronal (*'Training Error'*).

3.4. Elegir el número máximo de iteraciones permitidas en el entrenamiento de la red neuronal (*'Max Iteration'*).

4. Presionar el botón clasificar (*'Classify'*).

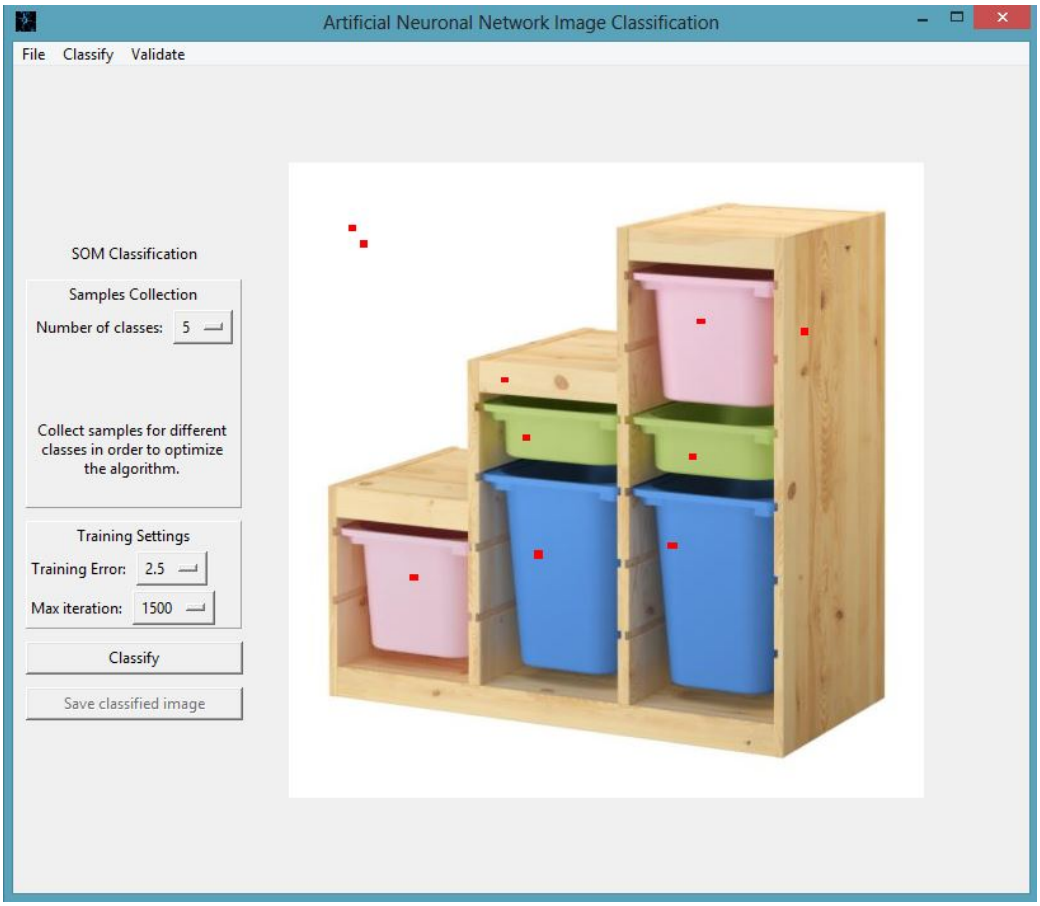


Figura 6: Menú de clasificación SOM.

4.3. Clasificación basada en el algoritmo K-means

Para realizar una clasificación de imagen basada en el tradicional algoritmo *K-means* se deberán seguir las siguientes instrucciones:

1. Abrir la imagen a clasificar:

1.1. Seleccionar *'Abrir imagen'* (*'Open image'*) dentro del menú de manejo de archivos (*'File menu'*).

1.2. Elegir la ubicación física del imagen original.

1.3. Presionar el botón *'Abrir'* (*'Open'*).

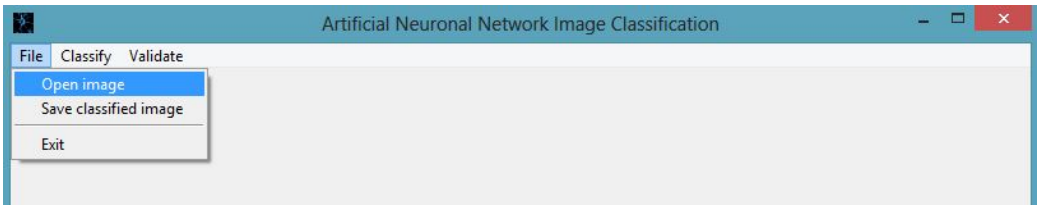


Figura 7: Abrir imagen original.

2. Seleccionar el clasificador K-means (*'K-means Classification'*) dentro del menú de clasificación (*'Classify menu'*).

Figura 8: Selección de clasificador K-means.

3. Definir los parámetros de clasificación a utilizar el entrenamiento del algoritmo *K-means*:

3.1. Seleccionar la cantidad de clases (*'Number of classes'*).

3.2. Tomar muestras conocidas (*'Training samples'*) para utilizar y hacer más eficiente el algoritmo de entrenamiento *K-means*.

- Para ello dibujar rectángulos sin levantar el cursor dentro de la imagen original.

3.3. Definir el máximo error permitido en el algoritmo de entrenamiento *K-means* (*'Training Error'*).

3.4. Elegir el número máximo de iteraciones permitidas en el algoritmo de entrenamiento *K-means* (*'Max Iteration'*).

4. Presionar el botón clasificar (*'Classify'*).

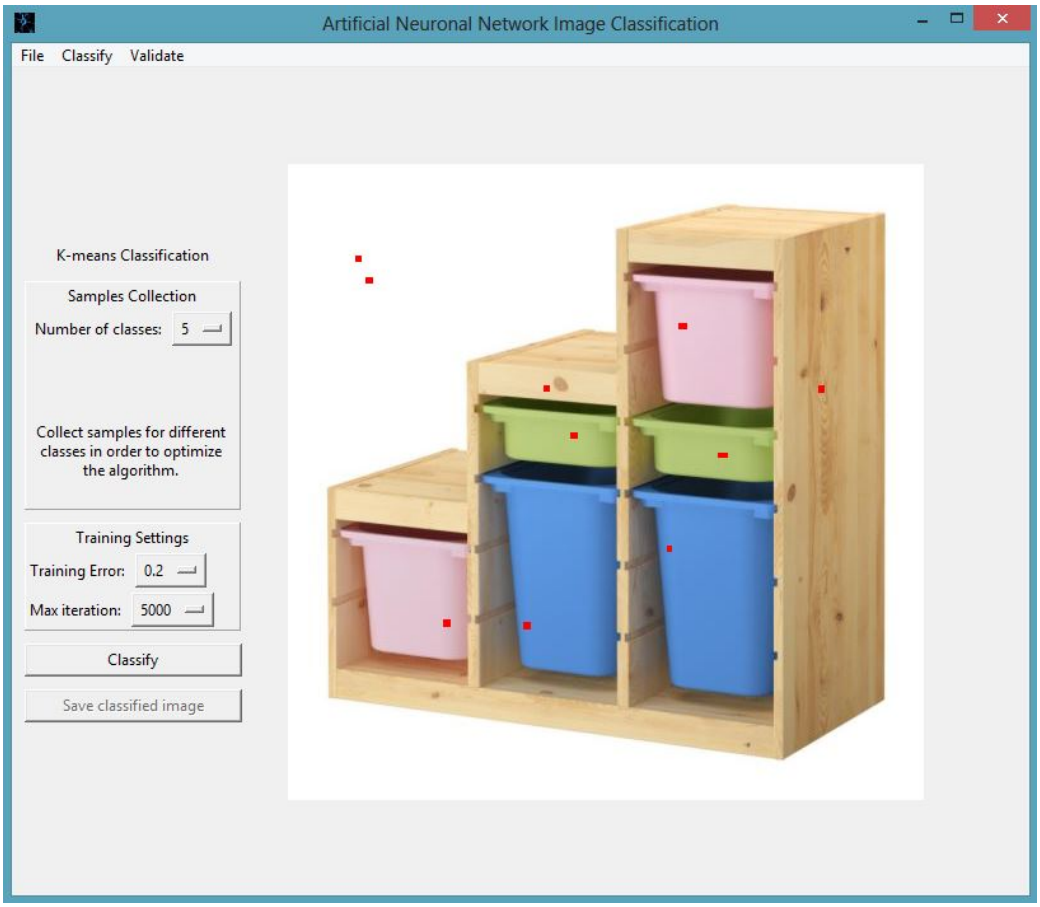


Figura 9: Menú de clasificación K-means.

4.4. Validación de la clasificación

Para realizar la validación de una clasificación se asume la existencia de una imagen de categorías previamente generada por alguno de los algoritmos provistos por este software.

Se deberán seguir las siguientes instrucciones:

1. Seleccionar el validador 'matriz de confusión' (*'Confusion matrix'*) dentro del menú de validación (*'Validate menu'*).

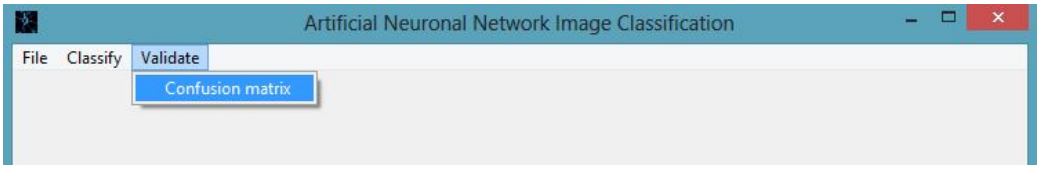


Figura 10: Selección de validación mediante matriz de confusión.

2. Definir los parámetros a utilizar durante la verificación de la clasificación:

3.1. Seleccionar la cantidad de clases (*'Number of classes'*).

3.2. Tomar muestras representativas de cada clase real (*'Samples'*) para utilizar durante la validación.

- Para ello dibujar rectángulos sin levantar el cursor dentro de la imagen original.
- Se debe asegurar la coincidencia entre el color de la muestra y el color expuesto en la imagen clasificada.

3. Presionar el botón 'Calcular' (*'Calculate'*).

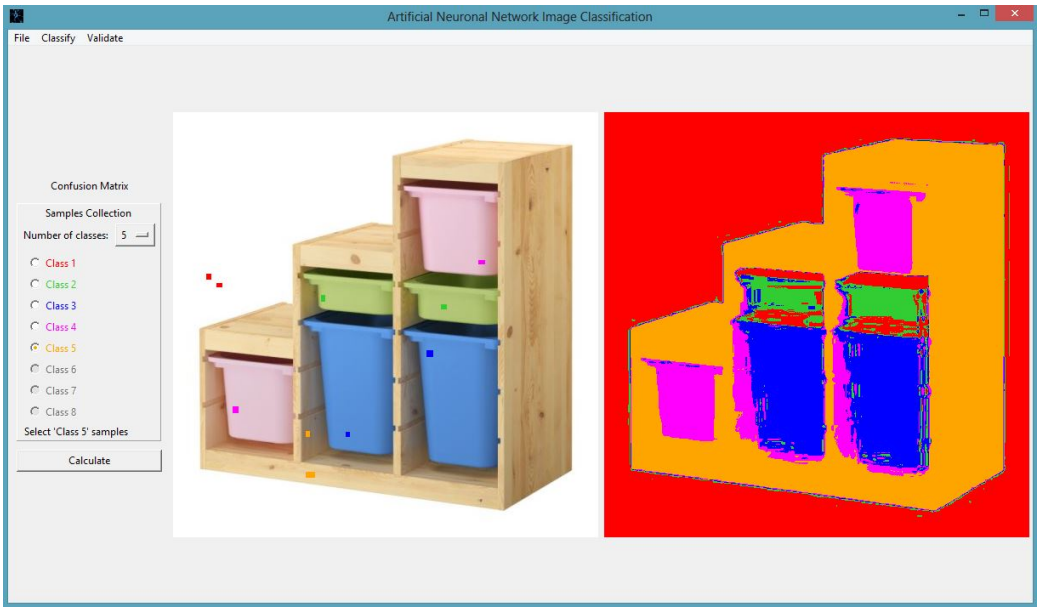


Figura 11: Menú de validación.