eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Diseño e implementación de un simulador SNN

Autor

Ian Fernandez Hermida

2021

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Diseño e implementación de un simulador SNN

Autor

Ian Fernandez Hermida

Director

Jose A. Pascual

Ander Soraluze

Resumen

El objetivo principal de este trabajo es desarrollar un simulador de *Spiking Neural Network* (SNN), modelos inspirados en el comportamiento biológico del cerebro, capaces de procesar información de forma eficiente y basada en eventos. Frente a las redes neuronales tradicionales, las SNN ofrecen ventajas como un menor consumo energético y un procesamiento más natural en el tiempo.

En la parte teórica, se abordan los fundamentos biológicos que motivan este enfoque, detallando el funcionamiento del modelo *Leaky Integrate-and-Fire* (LIF) y la regla de aprendizaje no supervisado *Spike-Timing Dependent Plasticity* (STDP). Además, se analiza el uso de métricas como el error cuadrático medio (MSE) para controlar la convergencia del aprendizaje.

El simulador se ha diseñado de forma modular y orientada a objetos, con clases específicas para representar la red, capas, neuronas y sinapsis. También se ha diseñado un sistema de configuración mediante ficheros que permite definir topologías de red complejas sin necesidad de modificar el código fuente.

La implementación, se ha realizado en C++, se ha construido la lógica completa de la red desde cero, prestando especial atención al tratamiento del tiempo, la gestión de los spikes y el aprendizaje sin supervisión. Para validar su funcionamiento, se ha empleado el dataset DVS128 Gesture en una arquitectura simple.

Los resultados obtenidos muestran un comportamiento coherente con lo esperado para una red de este tipo, y aunque los niveles de precisión no son elevados, las pruebas demuestran que el simulador es funcional, extensible y una base sólida sobre la que continuar desarrollando.

Objetivos de Desarrollo Sostenible

Este proyecto se alinea de manera directa con el ODS 7 (Energía asequible y no contaminante), al abordar el desarrollo de redes neuronales de impulsos (SNN), una tecnología inspirada en el cerebro humano cuyo funcionamiento es más eficiente energéticamente que el de las redes neuronales tradicionales. Las SNN procesan información de forma esparcida en el tiempo, solamente cuando se generan eventos, lo cual reduce significativamente el consumo energético al eliminar la necesidad de cálculos constantes o sincronización global. Esta característica las convierte en una alternativa prometedora para el diseño de sistemas de inteligencia artificial sostenibles.

Además, el simulador desarrollado permite estudiar, analizar y optimizar estos modelos, fomentando la investigación en tecnologías de bajo consumo, lo cual es clave para avanzar hacia sistemas más respetuosos con el medio ambiente y menos dependientes de infraestructuras energéticas alto consumo.

De forma complementaria, el proyecto también contribuye varios ODS más, como al ODS 9 (Industria, innovación e infraestructura), al promover el avance en tecnologías emergentes de computación, y también al ODS 4 (Educación de calidad), ya que facilita la formación y experimentación con modelos neuronales de manera accesible.

Trabajar con tecnologías energéticamente eficientes, permite demostrar que es posible avanzar en este ámbito sin comprometer la sostenibilidad. Incorporar esta visión en el desarrollo de proyectos académicos es un paso necesario para formar profesionales comprometidos con un progreso tecnológico que no ignore su impacto ambiental y social.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

En los últimos años, la inteligencia artificial se ha convertido en uno de los campos más relevantes, convirtiendo a las redes neuronales en uno de sus pilares principales. Gracias a su investigación y desarrollo, se han logrado alcanzar resultados impresionantes en diversas tareas como la clasificación, el reconocimiento de patrones y la toma de decisiones, entre otras, que están siendo de gran relevancia dentro de distintos sectores, como la visión artificial o el procesamiento del lenguaje natural.

Las redes neuronales tradicionales utilizan algoritmos de aprendizaje automático inspirados en el cerebro humano para tratar de replicar su forma de aprendizaje. Generalmente, estas redes están compuestas por nodos, también conocidos como neuronas, que se agrupan en capas y se interconectan con neuronas de otras capas mediante enlaces. A estos enlaces se les asignan pesos que determinan la intensidad de la conexión, y las neuronas hacen uso de sus enlaces tanto para transmitir información a otras neuronas como para actualizar sus pesos. De esta forma, las redes realizan predicciones en función de los datos de entrada que se procesan a lo largo de la red. Las redes pasan por una fase de entrenamiento donde procesan una gran cantidad de datos de entrada y ajustan los pesos de sus enlaces en base al acierto de las predicciones realizadas. Estas predicciones se perfeccionan a medida que se actualizan los pesos, dando como resultado una red entrenada que pueda predecir con cierto margen de error las salidas esperadas para cada tarea. Cuando termina la fase de entrenamiento, se verifica el aprendizaje de la red con un conjunto de datos distinto y se concluye si la red ha aprendido, valorando la tasa de éxito obtenida de sus predicciones.

Existen multitud de tipos de redes neuronales, cada una con sus características específicas. Cada tipo de red cubre una necesidad distinta pero a su vez trae consigo una serie de desventajas que en muchos casos imposibilitan el uso de la misma, como falta de recursos, alto coste computacional, etc. Sin embargo, existe una desventaja que afecta a todas ellas en mayor o menor medida. Con la rapidez con la que avanzan las redes neuronales, la cantidad de datos que se requiere procesar cada vez es mayor y, de igual forma, escala la cantidad de recursos computacionales necesarios para procesar esos datos. Esto, combinado con el gran coste computacional que se requiere para entrenar redes de mayor envergadura, desencadena un mayor gasto energético que no solo afecta económicamente, sino también al medioambiente.

Para solventar estos problemas, se está popularizando un tipo de red neuronal llamada *Spiking Neural Network* (SNN), que propone un enfoque diferente respecto a las redes neuronales convencionales. Mientras que las redes neuronales convencionales procesan los datos de entrada de forma continua, las SNNs reaccionan únicamente a eventos puntuales, es decir, a cambios significativos en el estímulo. Este modelo, inspirado más fielmente en el comportamiento biológico del cerebro, permite reducir significativamente el consumo energético y mejorar la eficiencia en varias de las tareas mencionadas anteriormente.

En este trabajo se estudian el funcionamiento y las características principales de las SNNs. El objetivo es desarrollar un simulador utilizando el lenguaje C++, aprovechando sus ventajas como el control detallado de memoria, la orientación a objetos y su alto rendimiento. Con este simulador se busca analizar los beneficios y limitaciones que ofrecen este tipo de redes, así como sentar las bases para futuras mejoras e implementaciones.

2. CAPÍTULO

Documento de objetivos del proyecto

El objetivo principal del proyecto es desarrollar un simulador de una red neuronal de impulsos no supervisada, completamente desde cero. Este simulador permitirá explorar y comprender en profundidad el comportamiento de este tipo de redes frente a diferentes configuraciones de entrada, así como evaluar su rendimiento frente a múltiples arquitecturas de red. Al construir la herramienta desde la base, se busca no solo replicar el funcionamiento de una SNN, sino también proporcionar una plataforma flexible que facilite la experimentación.

La primera tarea a realizar y una que se repetirá a lo largo del trabajo es la investigación. A pesar de tener ciertas nociones básicas sobre redes neuronales, será necesario emplear un tiempo considerable en el estudio de estas, para arraigar los conocimientos y asegurarse de que las futuras implementaciones se lleven a cabo de forma correcta. Dicho esto, los objetivos del proyecto se dividirán en varias secciones, y se deberá realizar una fase de investigación previa, para poder abordar cada uno de ellos.

Una vez adquirida la base teórica necesaria, el siguiente paso consistirá en diseñar e implementar un modelo funcional de una neurona de tipo *Leaky Integrate-and-Fire* (LIF). Este prototipo servirá como el componente fundamental de la red, por lo que su funcionamiento será indispensable para las siguientes fases del simulador. Este primer prototipo asegurará una base estable sobre la que posteriormente construir versiones más complejas.

Tras disponer de un modelo funcional de la neurona LIF, el siguiente objetivo será desarrollar una arquitectura de red básica que permita organizar las neuronas en capas y establecer conexiones entre ellas. Esta estructura servirá como esqueleto del simulador y

deberá ser lo suficientemente flexible como para permitir futuras extensiones o modificaciones. En un primer momento, esta arquitectura inicial no incluirá reglas de aprendizaje, ya que su propósito será probar que la transmisión de actividad entre neuronas se realiza de manera correcta. Una vez verificado su correcto funcionamiento, se podrá ampliar para incorporar procesos de aprendizaje no supervisado.

Una vez establecida la arquitectura básica de la red, el siguiente paso consistirá en seleccionar, analizar y procesar un conjunto de datos que sirva como entrada al sistema. Esta fase será la clave para poder validar el comportamiento de la red. Además de la entrada, será necesario establecer de forma clara la salida que se espera obtener. En esta etapa, se buscará recrear un sistema completamente funcional que reciba datos externos y produzca una salida en un formato interpretable.

Con las entradas de la red procesadas y establecida la capa de salida, la última fase del proyecto se centrará en el almacenamiento de los datos generados durante las simulaciones y en el análisis de los resultados obtenidos. Este paso será esencial para evaluar el comportamiento de la red, detectar posibles errores y validar la eficacia de las decisiones tomadas en el diseño e implementación.

Este conjunto de tareas permitirá cerrar el ciclo de desarrollo del proyecto con una evaluación del comportamiento del simulador y servirá como punto de partida para futuras mejoras.

A continuación, se enumeran de forma estructurada los objetivos de este proyecto:

1. Estudiar el funcionamiento de las SNNs.
2. Implementar una neurona de tipo LIF como base del simulador.
3. Diseñar una arquitectura de red que permita la conexión de múltiples capas compuestas por neuronas.
4. Seleccionar y preparar un conjunto de datos para alimentar el simulador.
5. Incluir mecanismos de aprendizaje no supervisado.
6. Registrar los resultados de las simulaciones y evaluar el rendimiento general del simulador.

3. CAPÍTULO

Fundamentos teóricos

Tras realizar un estudio sobre las SNNs, en este apartado se presentan los fundamentos biológicos que han inspirado el diseño de las redes neuronales de impulsos. Primero, se repasa el comportamiento general del sistema nervioso y visual, seguido de una descripción básica de la estructura y funcionamiento de las neuronas y se presenta el concepto de plasticidad sináptica. A continuación, se introducen diferentes modelos neuronales utilizados en el ámbito de las SNNs y se justifica la elección del modelo LIF en base a sus ventajas como la simplicidad y eficiencia. Finalmente, se analiza la relación entre este modelo y el comportamiento de una neurona biológica.

3.1. Inspiración biológica

En este apartado se exponen los fundamentos biológicos que inspiran el funcionamiento de las redes neuronales de impulsos. Se describe el procesamiento sensorial en el sistema nervioso, con especial énfasis en el sistema visual, y se analiza el comportamiento de una neurona a nivel individual. Finalmente, se introduce el concepto de plasticidad sináptica como base del aprendizaje biológico.

3.1.1. Sistema visual y sistema nervioso

Las SNN se inspiran en el funcionamiento del sistema nervioso, especialmente en la forma en que el cerebro procesa la información sensorial. En algunos animales, la percepción

del entorno y la toma de decisiones, se realiza mediante una red compleja de neuronas interconectadas que se comunican a través de señales eléctricas discretas llamadas impulsos nerviosos o potenciales de acción.

En el caso del sistema visual, el proceso comienza en la retina, una estructura sensible a la luz situada en el fondo del ojo. La retina está compuesta por varios tipos de células, entre las que destacan los fotorreceptores, encargados de transformar la luz en señales eléctricas. Existen tres tipos diferentes de fotorreceptores. Los conos son los encargados de proporcionar información acerca del color, los bastones recopilan información del brillo y las células ganglionares cuya función principal es transmitir hacia diferentes regiones del cerebro, la información procesada por la retina que viaja en forma de impulsos eléctricos. Una característica clave de estas células ganglionares es que no transmiten información de manera continua, sino que generan impulsos solo cuando detectan cambios significativos en el estímulo visual. Esto es lo que permite un procesamiento eficiente, centrado en los eventos relevantes.

Una vez los impulsos llegan al cerebro, se propagan a través de estructuras jerárquicas del sistema visual, que comienza por el núcleo geniculado lateral (NGL) y continua hacia la corteza visual primaria (V1), también conocida como corteza estriada. En esta región, la información comienza a descomponerse en características visuales básicas como bordes, orientaciones y movimientos. A medida que la información se propaga por el sistema V1 hacia la corteza extra estriada, que incluye múltiples áreas como, V2, V4 o MT, se integran características más complejas, permitiendo procesar una percepción global y detección de objetos en movimiento.

3.1.2. Neurona

Para lograr una comprensión profunda de lo expuesto en el apartado anterior, es fundamental analizar el funcionamiento de una neurona individual, ya que constituye la unidad básica de procesamiento en el sistema nervioso.

Una neurona clásica se compone de tres partes principales:

- Dendritas: son ramificaciones que reciben señales de otras neuronas. Su función principal es recibir impulsos de otras neuronas y enviarlas hasta el soma de la neurona a la que pertenecen.
- Soma: es el centro metabólico de la neurona (cuerpo celular) y el lugar donde llegan

las señales sinápticas transmitidas por las dendritas. Si la suma de estas señales alcanza un cierto umbral de activación, el soma genera un impulso nervioso.

- Axón: es una prolongación que transporta el impulso eléctrico generado en el soma hacia otras neuronas. Al final del axón, se liberan neurotransmisores en respuesta al impulso.

Estas células están rodeadas por una membrana potencial que separa el interior y el exterior de la celula. Debido a esta membrana, el interior de la celula tiene mayor carga negativa que el exterior. Esto da lugar a una diferencia potencial denominada potencial de membrana. La diferencia de potencial que existe entre el interior y exterior de la célula se conoce como potencial de reposo.

Una neurona puede recibir señales excitatorias o inhibitorias. Las señales excitatorias provocan una despolarización de la membrana (disminución de la diferencia de potencial), acercando el valor al umbral de disparo. Las señales inhibitorias, en cambio, hiperpolarizan la membrana (aumentan la diferencia de potencial), alejando el valor del umbral. Cuando la suma de señales excitatorias acumuladas en el soma alcanza un cierto umbral, se desencadena un potencial de acción que se propaga a lo largo del axón como una onda eléctrica.

En las sinapsis, los axones se conectan con las dendritas de la neurona postsináptica, modificando su potencial de membrana. Esta transmisión no es fija, ya que las sinapsis son plásticas, es decir, pueden fortalecerse o debilitarse en función de la actividad entre las neuronas conectadas. A este fenómeno se le conoce como plasticidad sináptica, y constituye la base del aprendizaje y la memoria.

Después del impulso generado por la neurona, la neurona entra en un periodo llamado periodo refractario, durante el que se restablece a el potencial de reposo y la neurona no puede generar un nuevo impulso hasta pasado cierto periodo de tiempo. Este mecanismo garantiza que los impulsos viajen en una única dirección y regula la frecuencia de disparo neuronal.

3.1.3. Plasticidad sináptica

Una característica esencial del sistema nervioso es su capacidad de aprendizaje y adaptación a través de la modificación de las conexiones sinápticas entre neuronas, fenómeno

conocido como plasticidad sináptica. Esta propiedad permite que el cerebro almacene información, se adapte a nuevos estímulos y recupere experiencias previas.

La plasticidad se presenta como un fortalecimiento o debilitamiento de las sinapsis en función de la actividad neuronal. Un ejemplo clásico de esta dinámica es la regla de Hebb, que puede resumirse como "las neuronas que se disparan juntas, se conectan juntas". Es decir, si una neurona A contribuye repetidamente al disparo de una neurona B, la conexión sináptica entre ellas se refuerza.

Existen distintos tipos de plasticidad sináptica, como la plasticidad a corto plazo que es dependiente del historial reciente de actividad y la plasticidad a largo plazo, que produce cambios duraderos en la eficacia sináptica.

3.2. Modelos de neuronas en SNNs

El comportamiento de las neuronas biológicas se ha modelado de distintas formas en el ámbito computacional, dando lugar a varios tipos de neuronas artificiales. Estos modelos buscan reflejar el realismo biológico en el contexto computacional, permitiendo simular con diferentes grados de precisión el comportamiento de las neuronas reales. A continuación se presentan algunos de los modelos más conocidos.

3.2.1. Modelo de Hodgkin-Huxley

Este modelo fue propuesto en 1952 y describe con gran precisión los mecanismos biofísicos que originan el potencial de acción, utilizando un conjunto de ecuaciones diferenciales que modelan los canales iónicos dependientes del voltaje. A pesar de su alta precisión, este modelo requiere un elevado coste computacional, lo que lo hace poco práctico para simular redes a gran escala [Hodgkin AL, 1952].

3.2.2. Modelo de FitzHugh-Nagumo

Este modelo es una simplificación del de Hodgkin-Huxley. Conserva las propiedades dinámicas fundamentales del disparo neuronal, pero reduce significativamente la complejidad computacional. Aunque es más eficiente, sigue siendo demasiado complejo para muchas aplicaciones prácticas.

3.2.3. Modelo de Izhikevich

Diseñado como un equilibrio entre realismo biológico y eficiencia computacional, el modelo de Izhikevich es capaz de reproducir una gran variedad de patrones de disparo neuronal con una formulación matemática más sencilla. Sin embargo, su implementación requiere el ajuste de múltiples parámetros y una gestión matemática más elaborada, lo que puede dificultar su uso en implementaciones desde cero [Izhikevich, 2003].

3.2.4. Modelo *Leaky Integrate-and-Fire* (LIF)

Es uno de los modelos más extendidos y utilizados en la simulación de SNNs debido a su simplicidad y eficiencia. En este modelo, el potencial de membrana se incrementa con cada impulso recibido y, al mismo tiempo, se atenúa de forma continua con el tiempo. Cuando el potencial supera un umbral, la neurona genera un impulso y el potencial se reinicia.

3.2.5. Justificación de la elección del modelo LIF

Para este trabajo se ha optado por implementar el modelo LIF como base para el desarrollo del simulador. Esta elección se justifica mediante varias razones que se exponen a continuación.

En primer lugar, el modelo LIF presenta una formulación matemática sencilla que permite comprender fácilmente la dinámica del potencial de membrana y su evolución temporal, lo cual es ideal para un entorno de desarrollo didáctico como el de este proyecto.

En segundo lugar, su eficiencia computacional lo convierte en una opción viable para la simulación de redes con múltiples neuronas sin que ello suponga una carga excesiva para el sistema. Al ser un modelo ampliamente utilizado, existe además una gran cantidad de literatura y ejemplos que facilitan la validación y comparación de resultados.

Por último, el modelo LIF permite incorporar de forma directa mecanismos de aprendizaje biológicamente plausibles, lo que lo hace adecuado para explorar la evolución del comportamiento de la red en un contexto de aprendizaje no supervisado.

Por todo ello, se considera que el modelo LIF ofrece un equilibrio entre simplicidad, funcionalidad y capacidad de extensión, convirtiéndose en la opción más apropiada para los objetivos de este proyecto.

3.3. Relación entre el modelo LIF y la neurona biológica

Al igual que una neurona real, una neurona LIF acumula señales entrantes (potenciales postsinápticos) provenientes de otras neuronas. Estas señales se integran en el tiempo como una suma del potencial de membrana. El término "*leaky*" (con fuga) hace referencia a la tendencia natural de la membrana a perder carga con el tiempo y reproduce la disipación del potencial así como sucede en las neuronas reales.

Cuando el potencial de membrana acumulado supera un determinado umbral, el modelo LIF genera un impulso (spike), simulando así la generación de un potencial de acción. Posteriormente, el potencial se reinicia, y la neurona entra en un periodo refractario, durante el cual no puede generar nuevos impulsos, del mismo modo que ocurre en el sistema biológico. Esta dinámica simple pero efectiva permite que el modelo reproduzca los principios fundamentales del procesamiento neuronal basado en eventos.

Este comportamiento puede observarse en la Figura 3.1, donde se muestra cómo los impulsos entrantes de distintas neuronas provocan una acumulación del potencial de membrana en la neurona i, que dispara al alcanzar el umbral v_{th} y entra en un periodo refractario tras reiniciar su potencial a v_{reset}.

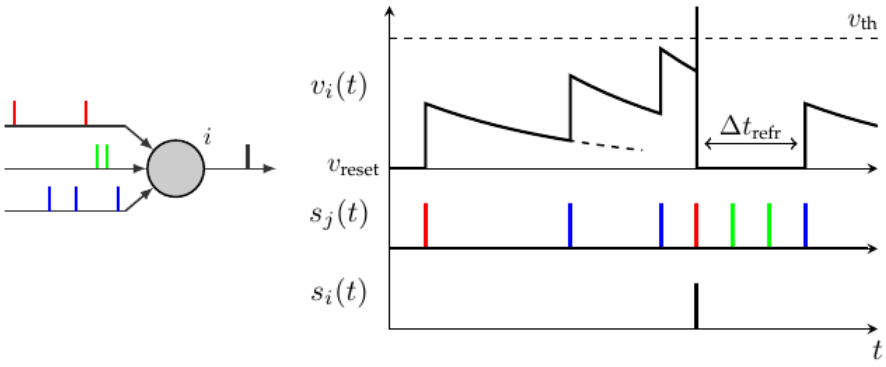

Figura 3.1: Esquema del funcionamiento de una neurona LIF. La neurona i integra los impulsos recibidos de varias neuronas presinápticas. Cuando el potencial de membrana $v_i(t)$ supera el umbral v_{th}, se genera un spike y el potencial se reinicia a v_{reset}, entrando en un periodo refractario [Paredes-Vallés et al., 2020].

Otro aspecto clave que puede modelarse con neuronas LIF es la plasticidad sináptica, que permite simular procesos de aprendizaje y adaptación. En particular, las SNNs pueden implementar reglas como *Spike-Timing Dependent Plasticity* STDP, que ajustan los pesos sinápticos en función del orden temporal entre los disparos de las neuronas pre y post-

sinápticas. Esta regla refuerza las conexiones cuando la neurona presináptica se activa poco antes que la postsináptica, y las debilita cuando ocurre lo contrario, reproduciendo de esta forma un mecanismo de aprendizaje no supervisado observado en el cerebro. De esta forma, no solo se simula el comportamiento eléctrico de las neuronas, sino también su capacidad de modificar dinámicamente las conexiones, reproduciendo la base del aprendizaje biológico.

4. CAPÍTULO

Diseño e Implementación

En este capítulo se presenta, en primer lugar, el diseño general del simulador, guiado por un diagrama de clases y un diagrama de flujo que reflejan la estructura y relaciones entre los componentes principales del sistema. Posteriormente, se aborda en detalle la implementación de cada una de las clases clave, describiendo su funcionalidad, responsabilidades internas y cómo colaboran entre sí para simular el comportamiento de una red neuronal de impulsos.

4.1. Diseño

El diseño del simulador de la red neuronal de impulsos se ha planteado desde una perspectiva modular y orientada a objetos. Para ello, se ha definido una arquitectura basada en clases bien diferenciadas, cada una con una responsabilidad clara, para simular el comportamiento de una red neuronal de impulsos no supervisada basada en el modelo LIF.

4.1.1. Diagrama de clases

La Figura 4.1 muestra el diagrama de clases general del simulador, donde se puede observar la estructura jerárquica del sistema y las relaciones entre sus distintos componentes. Para una mejor comprensión del diagrama, se han obviado los atributos y se han introducido solamente los métodos más relevantes.

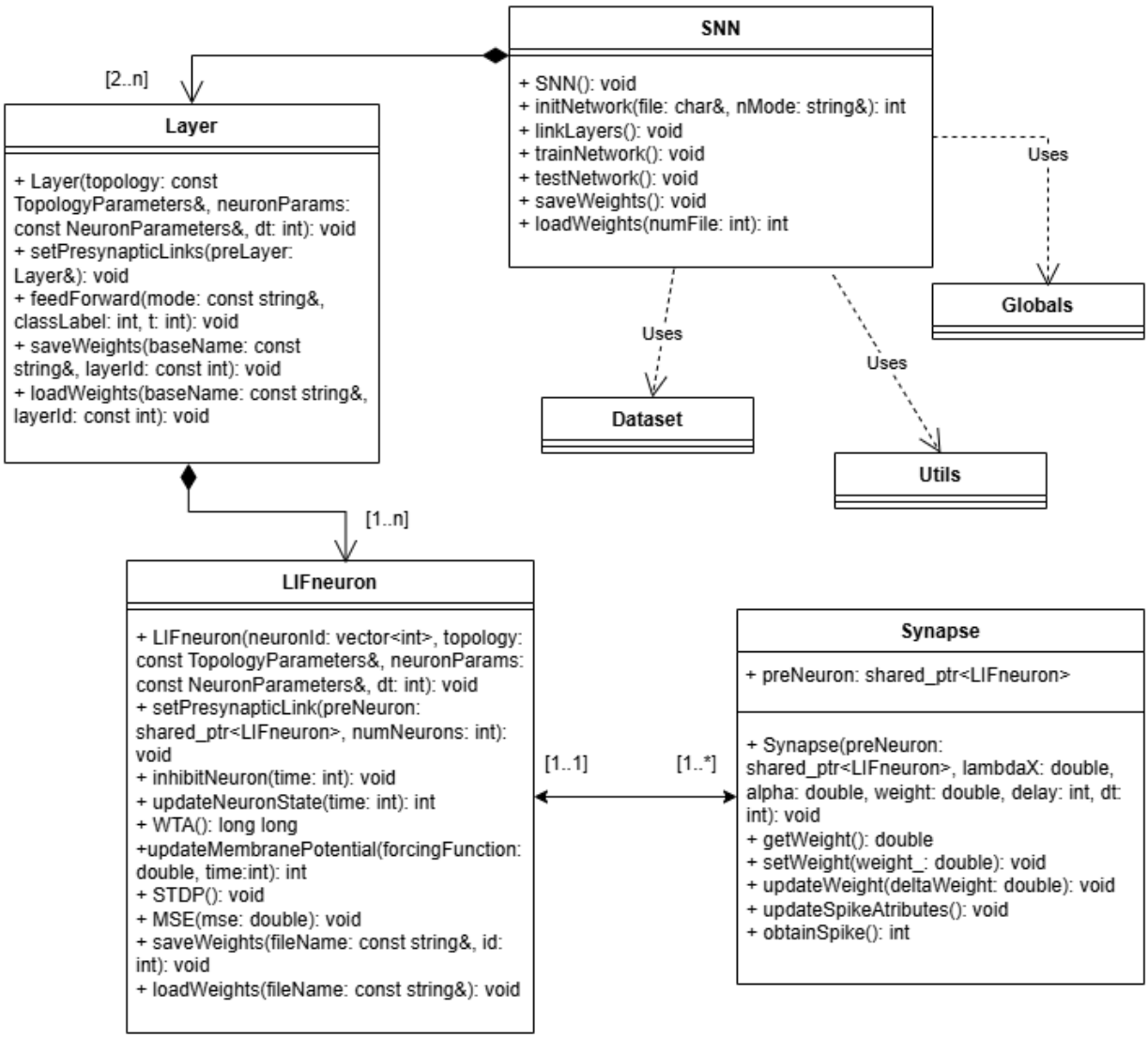

Figura 4.1: Diagrama de clases del simulador.

La clase principal es `SNN` y su función es gestionar la red en su conjunto: inicializar los parámetros y la estructura desde fichero, conectar las capas, entrenar la red, ejecutar simulaciones de test y guardar o cargar los pesos sinápticos. Esta clase utiliza módulos auxiliares para el manejo de datos de entrada (`Dataset`), parámetros globales de configuración (`Globals`) y utilidades generales (`Utils`).

La clase `Layer` representa una capa de neuronas dentro de la red. Cada capa puede contener un número arbitrario de neuronas `LIFneuron` y se encarga de establecer las conexiones sinápticas con respecto a la capa anterior. La función `feedForward()` permite propagar la actividad neuronal a través de la red.

La clase `LIFneuron` implementa el modelo de neurona de tipo LIF. Incluye los métodos necesarios para actualizar el potencial de membrana, gestionar el estado de la neurona, realizar el aprendizaje mediante la regla STDP y utilizar la medida del *Mean Square Error* (MSE) para indicar a la neurona que su aprendizaje ha concluido.

Cada conexión sináptica entre neuronas se representa con la clase `Synapse`. Esta clase almacena información clave como el peso, el retardo y la referencia a la neurona presináptica. Además, permite modificar dinámicamente el peso sináptico en función de la actividad neuronal, acorde a la regla STDP definida.

4.1.2. Diagrama de flujo

En la Figura 4.2, se presenta el diagrama de flujo que describe en detalle el comportamiento del simulador. En el se pueden contemplar tanto la fase de entrenamiento como la de prueba, diferenciando claramente los bloques de inicialización, procesamiento de datos, propagación de impulsos, aprendizaje y almacenamiento de resultados.

A continuación, se describen cada una de las etapas del flujo:

1. **Inicio y carga de parámetros.** El programa comienza cargando los ficheros de configuración necesarios. Estos ficheros contienen:

   - `PARAMETERS`: Contiene los parámetros de simulación. Tiempo total y paso temporal dt.
   - `HYPERPARAMETERS`: Los parámetros de las neuronas como, valores de umbral, potencial de reposo, constantes de tiempo, entre otros.
   - `TOPOLOGY`: La estructura de red, que incluye, el número de capas, tipo de conexión, cantidad de neuronas por capa, etc.

   Esta carga se realiza mediante el método `initNetwork()`, que interpreta los bloques del fichero de topología.

2. **Inicialización de la red.** Con la información cargada, se crean las capas de la red neuronal, representadas por objetos de la clase `Layer`. A cada capa se le asignan sus neuronas `LIFneuron` con los parámetros definidos.

   Una vez instanciadas las capas, se ejecuta el método `linkLayers()`, que establece las conexiones sinápticas entre capas consecutivas, teniendo en cuenta la configuración de conectividad establecidas en la topología de red.

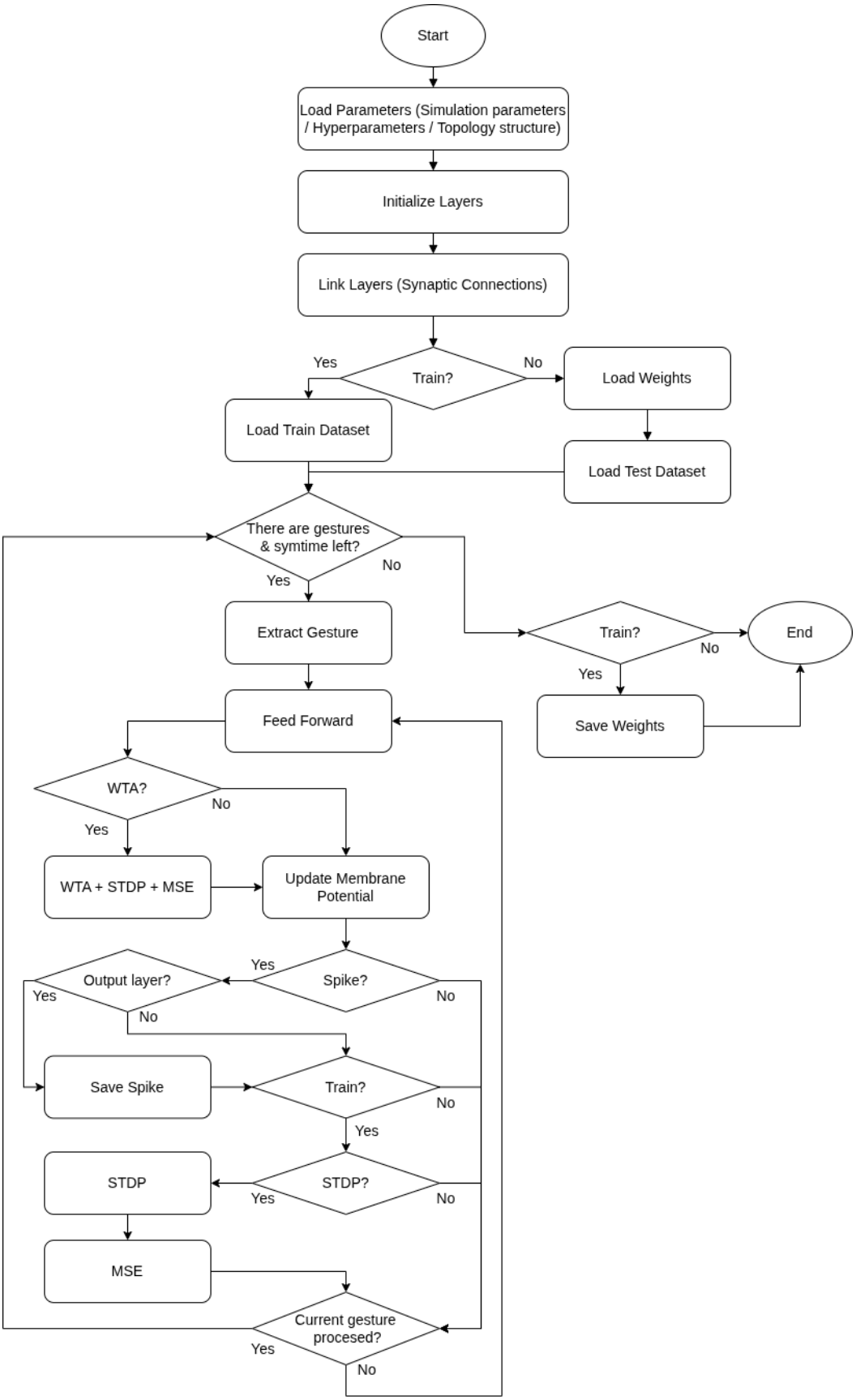

Figura 4.2: Diagrama de flujo del simulador.

3. **Modo de operación: train o test.** El flujo se bifurca según el modo de ejecución especificado:

   - Modo `-train`. Se entrenará la red desde cero con pesos inicializados de forma aleatoria.
   - Modo `-test`. Se cargarán los pesos de un entrenamiento previo y se evaluará la red sin modificar su estado.

   En función al modo seleccionado se cargarán, o bien los archivos reservados para el entrenamiento, o bien los separados para la fase de prueba. Estos archivos contienen los datos que servirán como entrada de la red, simulando las señales recibidas por el sistema visual y transmitiendo sus impulsos a través del sistema nervioso.

4. **Procesamiento de cada gesto.** Para cada gesto que se encuentre en los archivos, se extraen los eventos que se van a inyectar en la red en cada paso temporal mediante `convertToSpikeCubesByPolarity()`.

5. **Propagación de la actividad neuronal.** En cada paso temporal las neuronas de la capa de entrada reciben los spikes de los eventos extraidos y se realiza una propagación hacia adelante mediante `feedForward()`.

   Cada neurona actualiza su potencial de membrana con `updateMembranePotential()` y si esta actualización consigue que el potencial supere un umbral establecido, la neurona dispara, almacena su spike para que la siguiente capa la recoja con el debido retardo de cada sinapsis e inmediatamente entra en su periodo refractario.

   Si el disparo se produce en la capa de salida, se registra en un fichero para su posterior análisis.

6. **Mecanismo *Winner Takes All* (WTA).** Si está habilitado el modo WTA en una capa intermedia, antes de actualizar la membrana potencial, se realiza una competición entre las neuronas de la capa previa, donde se selecciona la pre-neurona con mayor activación como una única ganadora por competición. Esto implica que solo esa neurona puede disparar, y el resto se inhiben temporalmente. Además, la ganadora será la única que actualice sus pesos en función a la regla STDP y evalúe si detiene el aprendizaje teniendo en cuenta el MSE.

7. **Aplicación de STDP en modo de entrenamiento.** Cuando se detecta un spike en una capa donde se ha habilitado el aprendizaje, se ejecuta la regla de STDP. Esta regla ajusta los pesos sinápticos en función del tiempo relativo entre los spikes pre y

post sinápticos. Asimismo se realiza el calculo del MSE con el objetivo de mantener el aprendizaje de cada neurona bajo control.

8. **Final del programa y almacenamiento de pesos.** Cuando se completa la simulación de un gesto, se continúa con el siguiente hasta terminar todos con todos los gestos del archivo y con todos los archivos de entrenamiento o pruebas. Si se ha establecido el tiempo de simulación, el programa se detendrá cuando haya pasado dicho tiempo.

   Antes de finalizar y cuando se haya procesado toda la red, si el programa está en modo entrenamiento, se guardan los pesos actualizados de la red para su uso posterior en pruebas.

4.2. Implementación

Tras haber definido en el apartado anterior la arquitectura del simulador, en esta sección se describe en detalle su implementación. El objetivo es mostrar cómo se ha traducido cada una de las decisiones de diseño a código, explicando el funcionamiento interno de las clases principales, los métodos más relevantes y la lógica que permite la simulación del comportamiento de una SNN. Para facilitar la comprensión de este apartado, como se aprecia en la Figura 4.3, se ha elaborado un diagrama que representa la conectividad entre los distintos métodos principales de las clases que componen el sistema.

4.2.1. main.cpp

El archivo main.cpp contiene el punto de entrada del programa. Su función es analizar los argumentos de entrada `-train` o `-test` y delegar la ejecución de las funciones correspondientes al objeto principal `SNN`.

Sus responsabilidades son:

- Validar los argumentos introducidos por el usuario.
- Invocar la inicialización del simulador mediante `initNetwork()`.
- Mostrar la topología de red haciendo uso del método `viewTopology()`.

Figura 4.3: Diagrama de conectividad entre métodos. Los óvalos indican métodos privados de la clase.

- Ejecutar el entrenamiento con `trainNetwork()` o la evaluación con `testNetwork()` según el modo seleccionado.
- Guardar o cargar los pesos sinápticos mediante `saveWeights()` o `loadWeights()`.

4.2.2. SNN.cpp

La clase SNN constituye el núcleo de la simulación y se encarga de coordinar el comportamiento general del programa. Esta clase se compone de cuatro funciones clave que se describen con exactitud a continuación.

Primero, el método `initNetwork()` permite inicializar la red neuronal leyendo la información contenida en un fichero de topología. Este fichero contiene tres bloques principales que ya se han expuesto en la sección 4.1.2 Diagrama de flujo. Para cada uno de ellos, se ha implementado un método de *parsing* específico: `parseParameters()`, `parseHyperparameters()` y `parseTopology()`. Durante esta fase se crean objetos `Layer` para cada una de las capas especificadas en el apartado `TOPOLOGY`. Junto con la llamada a este constructor, también se instancian las neuronas de la capa que se acaba de crear.

Una vez creada la estructura de capas, el método `linkLayers()` establece las conexiones sinápticas entre capas consecutivas. Esto se realiza llamando a `setPresynapticLinks()` de cada capa, donde se instancian las sinapsis entre las neuronas de la capa actual y la anterior. Esta conexión se explica con detalle en la sección 4.2.3 Layer.cpp.

El método `trainNetwork()` gestiona el ciclo completo de entrenamiento. Comienza leyendo una lista de archivos de entrenamiento (trials_to_train.txt) y recorre cada uno de ellos. Por cada gesto anotado en los archivos CSV, se realiza lo siguiente:

- Se extraen los eventos del archivo `.aedat`.
- Se trata la polaridad de los datos transformándolos en un mapa de spikes bidimensional ON/OFF con `convertToSpikeCubesByPolarity()`.
- Se inyectan los spikes a la capa de entrada en cada instante de tiempo a través de `processGestureData()`.
- Se propagan los impulsos a través de la red usando `feedForward()`.

El parámetro `symTime` se incrementa por cada paso de tiempo, y se utiliza para gestionar los límites temporales de la simulación, deteniendo la misma al alcanzar `symCap` el

tiempo de simulación establecido en el fichero de configuración. Una vez finalizado el entrenamiento, desde el `main.cpp`, se le indica a la red que debe almacenar los pesos entrenados hasta el momento con el método `saveWeights()`.

En `testNetwork()`, el flujo es similar al entrenamiento, pero sin aplicar aprendizaje. En este caso, se carga una lista de archivos de test (trials_to_test.txt), se cargan los pesos previamente entrenados mediante `loadWeights()` y se simula la actividad de la red con el objetivo de evaluar su rendimiento sin modificar sus sinapsis.

Adicionalmente, el método `viewTopology()` imprime por consola la configuración de la red, mostrando por cada capa su tipo, dimensiones, modelo de neurona, forma de conectividad y parámetros adicionales como tamaño de ventana o número de kernels en capas locales. Esta funcionalidad resulta útil para verificar que la red ha sido correctamente inicializada.

4.2.3. Layer.cpp

La clase Layer representa una capa de neuronas dentro de la red neuronal de impulsos. Su implementación permite instanciar un conjunto de neuronas LIF, definir su estructura espacial (alto, ancho y canales), establecer las conexiones sinápticas respecto a la capa anterior y propagar la actividad neuronal mediante el método `feedForward()`. Al igual que la clase SNN, también se encarga del almacenamiento y recuperación de los pesos sinápticos.

El constructor `Layer()` recibe como parámetros la estructura topológica de la capa (encapsulada en `TopologyParameters`), los parámetros del modelo de neurona (`NeuronParameters`) y el paso temporal `dt`. A partir de esta información, se generan e inicializan todos los objetos LIFneuron correspondientes a la capa, utilizando punteros inteligentes `shared_ptr` para facilitar la gestión de memoria y la interconexión con otras capas. Cada neurona se instancia con un identificador único que codifica su posición tridimensional en la capa (coordenadas x, y y canal).

El método setPresynapticLinks() establece las conexiones entre las neuronas de la capa actual y las de la capa previa, en función del tipo de conectividad especificada:

- `local` (convolucional): cada neurona se conecta a una ventana $r \times r$ de neuronas en la capa anterior, simulando un filtro de convolución sobre cada canal. Si se establece está conectividad, en la configuración también será posible seleccionar el tamaño del filtro, tanto la altura o anchura (r), como la profundidad (k).

- `sparse`: se conecta cada neurona a neuronas concretas de la capa previa según un listado explícito de pares (origen, destino).
- `dense` (fully-connected): cada neurona se conecta con todas las neuronas de la capa anterior.

La lógica del método permite gestionar distintas topologías de conexión, adaptándose dinámicamente a los parámetros de entrada.

El método `feedForward()` ejecuta el avance de la actividad neuronal para todos los objetos LIFneuron de la capa. Si la capa es de tipo `Output`, los spikes generados se registran en un fichero de texto para su posterior análisis. En las capas intermedias o de entrada, simplemente se actualiza el estado interno de cada neurona. La actualización del estado se realiza mediante el método `updateNeuronState()`, que devuelve un valor binario indicando si la neurona ha disparado.

En esta clase vuelven a estar presentes los métodos `saveWeights()` y `loadWeights()` que a pesar de realizar ciertas operaciones para formalizar el nombre del fichero, lo único que hacen es recorrer las neuronas ya entrenadas de la capa e indicarles que guarden y carguen los pesos respectivamente.

4.2.4. LIFneuron.cpp

La clase LIFneuron representa la unidad fundamental de procesamiento del simulador. Su implementación encapsula tanto el estado interno de la neurona como el disparo, periodo refractario, aprendizaje por STDP, y detención del aprendizaje basada en MSE.

El constructor `LIFneuron()` recibe tres parámetros fundamentales:

- Un identificador `neuronId`,
- Los parámetros topológicos de red, número de sinapsis, retardo mínimo y máximo, uso de reglas `WTA` y `STDP`, y el modo de operación.
- Los parámetros neuronales, incluyendo los potenciales de reposo y umbral, la constante de tiempo de membrana, el periodo refractario y factores de aprendizaje.

Durante la inicialización, se establece el mismo potencial de membrana que de reposo, sin comenzar en periodo refractario, sin spike activo, y con trazas sinápticas y pesos aún sin ajustar. Si la red está en modo test, el aprendizaje se desactiva.

Cada neurona mantiene una colección de objetos Synapse, que representan enlaces con neuronas presinápticas. Las sinapsis se crean mediante el método `setPresynapticLink()` y se inicializan con pesos aleatorios y retardos distribuidos entre los valores definidos en el fichero de configuración. A la hora de establecer los enlaces si las neuronas poseen múltiples sinapsis, se realizan tantas conexiones hacia cada neurona de la capa anterior como número de sinapsis.

El método updateNeuronState() es el encargado de actualizar el estado interno de una neurona en un instante de tiempo determinado. Durante su ejecución, la neurona recopila la actividad de sus sinapsis, calcula el valor de excitación total que recibe (forzamiento) y decide si genera un spike.

En primer lugar, si la neurona tiene activada la regla WTA, se llama al método `WTA()` para identificar si alguna sinapsis ha sido ganadora. A continuación, se itera sobre todas las sinapsis para obtener la actividad presináptica usando `obtainPreviousSpike()`. La información recibida sobre los impulsos generados por las neuronas de la capa previa, se utiliza para actualizar las trazas presinápticas mediante `updatePresinapticTrace()` y calcular la carga sináptica acumulada a través de `updateForcingFunction()`.

En el caso de que una sinapsis corresponda con la preneurona ganadora identificada por `WTA()`, se aplica sobre ella la regla de aprendizaje `STDP()` y se marca para su posterior inhibición. Las sinapsis vecinas, es decir, aquellas que han participado en la competición pero no han sido ganadoras, también son marcadas. Cuando se detecta que una preneurona ya no tiene más sinapsis por actualizar en el ciclo actual y está marcada, se interpreta que ha perdido la competición. En consecuencia, dicha neurona es inhibida mediante el método `inhibitNeuron()` y entra en su periodo refractario.

Además, durante esta iteración sobre las sinapsis, se monitoriza el rango de actividad sináptica para normalizar más adelante las trazas y los pesos, como se puede observar en las ecuaciones (4.8) y (4.9). Cuando se ha procesado toda la información, se llama a `updateMembranePotential()` para actualizar el potencial de membrana de la neurona en función de la *forcing function*.

Finalmente, si la neurona está en modo aprendizaje y ha disparado, se aplica la regla de aprendizaje mediante `STDP()`, la cual ajusta los pesos sinápticos con `updateWeight()` y calcula el error cuadrático medio con `MSE()` para determinar si se ha alcanzado un estado de convergencia.

La implementación del método `WTA()` se ha basado en el enfoque descrito en el artículo [Thorpe, 1990], con la diferencia de que para este proyecto han sido añadidas neuronas

intermedias para inhibir las neuronas vecinas, si no que son las propias neuronas postsinápticas las que toman la decisión de que neuronas inhibir. Cuando se activa la regla WTA, cada neurona de la capa actual selecciona entre todas sus entradas presinápticas que hayan transmitido un impulso, la que ha alcanzado el mayor potencial de membrana (`vMax`). Este método únicamente identifica si ha habido ganador o no. En caso de que haya habido, devuelve la posición de la neurona ganadora. El resto del trabajo se realiza en `updateNeuronState()`. Este mecanismo busca promover la especialización de las neuronas para que aprendan a diferenciar patrones.

El método `updateMembranePotential()` implementa la dinámica del potencial de membrana de la neurona. Esta función recibe el valor de forzamiento (`forcingFunction`) como parámetro de entrada, calculado previamente en `updateNeuronState()`, y actualiza el estado de la neurona cada instante de tiempo (t).

En primer lugar, se verifica si la neurona se encuentra en periodo refractario. En tal caso, se comprueba si dicho periodo ha finalizado en función del tiempo transcurrido desde el último disparo. Si la neurona aún está dentro del periodo de refracción, no se actualiza su estado y se retorna 0, indicando que no se ha producido ningún spike.

Si la neurona está activa, se aplica la ecuación diferencial discreta que rige la evolución de su potencial de membrana v:

$$v(t) = v(t) + \left(-\frac{v(t) - v_{\text{rest}}}{\tau_m} + \text{forcingFunction} \right) \tag{4.1}$$

Esta fórmula modela el decaimiento del potencial hacia el potencial de reposo `vRest`, así como el efecto de la *forcing function*. En la implementación, el coeficiente de decaimiento se aproxima mediante $\frac{1}{\tau_m}$, simplificando la expresión.

Tras aplicar la actualización, se impone una condición para evitar que el potencial caiga por debajo del valor de reposo, y se evalúa si el nuevo valor ha alcanzado o superado el umbral de disparo (`vTh`). Si esto ocurre, la neurona genera un spike, se registra el valor máximo alcanzado (`vMax`), se reinicia el potencial al valor de reseteo (`vReset`) y se activa el estado refractario. De lo contrario, se retorna 0, indicando ausencia de disparo en este paso temporal.

El método `STDP()` es el que permite que se lleve a cabo el aprendizaje no supervisado de la red. Básicamente, por cada neurona que deba realizar el aprendizaje, se recorren sus trazas presinápticas y se actualizan sus pesos en función a la ecuación (4.2). A continuación se explica en detalle la fórmula en la que se fundamenta este principio de aprendizaje

no supervisado, para comprender cómo la red modifica sus conexiones en función de la actividad y cómo esto permite capturar correlaciones espaciotemporales sin necesidad de supervisión externa.

$$\Delta W = \eta \cdot (\text{LTP} + \text{LTD}) \tag{4.2}$$

Donde ΔW es el cambio aplicado al peso sináptico, η es la tasa de aprendizaje (*learning rate*), LTP (*Long-Term Potentiation*) representa el refuerzo sináptico, representado en la ecuación (4.3), y LTD (*Long-Term Depression*) representa el debilitamiento sináptico, que se desglosa en la ecuación (4.4).

Componente LTP:

$$\text{LTP} = e^{-(W - u_{\text{init}})} \cdot \left(e^{\hat{X}} - a\right) \tag{4.3}$$

Esta expresión se divide en dos factores:

1. Dependencia del peso: $e^{-(W - u_{\text{init}})} \rightarrow$ cuanto más grande es el peso respecto a su valor inicial, menor será la contribución al refuerzo. Es decir, los pesos grandes tienden a estabilizarse, frenando su crecimiento.

2. Dependencia de la traza presináptica: $e^{\hat{X}} - a \rightarrow$ cuanto más alta es la traza normalizada $\hat{X}$, mayor será el refuerzo. Esto refleja la contribución reciente de la neurona presináptica al disparo.

Componente LTD:

$$\text{LTD} = -e^{(W - w_{\text{init}})} \cdot \left(e^{(1-\hat{X})} - a\right) \tag{4.4}$$

Esta expresión al igual que la anterior, también se divide en dos factores:

1. Dependencia del peso: $e^{(W - w_{\text{init}})} \rightarrow$ cuanto más grande es el peso, mayor será el debilitamiento. Esto evita que los pesos se disparen hacia valores muy altos.

2. Dependencia inversa de la traza: $e^{(1-\hat{X})} - a \rightarrow$ cuanto más baja es la traza presináptica, mayor es la penalización. Las conexiones poco activas tienden a debilitarse.

Para una mejor comprensión de las formulas, se describe el significado y la interpretación de los elementos que las componen: W es el peso sináptico, w_{init} representa el peso inicial asignado al comienzo del aprendizaje, $\hat{X} \in [0,1]$ la traza presináptica normalizada (representada en la ecuación 4.8) y $a \in (0,1)$ simboliza un parámetro que regula la sensibilidad del refuerzo a la traza.

Esta regla de aprendizaje, combina así dos principios clave, un refuerzo exponencial para las sinapsis activas cuyo peso aún no ha crecido en exceso y un debilitamiento exponencial para sinapsis con baja contribución o pesos excesivos.

Por último, a medida que la red aprende, las sinapsis que se alejan de su valor de equilibrio, es decir, el punto en el que las contribuciones de LTP y LTD se compensan mutuamente, son modificadas progresivamente hasta que se alcanza una estabilidad. Esta convergencia puede seguirse gracias al cálculo de MSE mediante la ecuación (4.5), que se realiza con un método con el mismo nombre `MSE()`.

$$\mathcal{L}_i = \frac{1}{n^{l-1}m} \sum_{j=1}^{n^{l-1}} \sum_{d=1}^{m} \left(\hat{X}_{i,j,d}(t) - \hat{W}_{i,j,d}\right)^2 \tag{4.5}$$

Donde n^{l-1} es el numero total de sinapsis, m el numero de multisinapsis, i, j y d hacen referencia a la neurona, sinapsis y multisinapsis actuales respectivamente, $\hat{X}_{i,j,d}(t)$es la traza presináptica normalizada y $\hat{W}_{i,j,d}$ es el peso sinaptico normalizado. Nuevamente la normalización de las trazas y los pesos se muestra en las ecuaciones (4.8) y (4.9).

A medida que el aprendizaje progresa, el valor medio del MSE disminuye, lo que indica que los pesos se están ajustando correctamente a la información recibida. Cuando el MSE alcanza un valor inferior a un umbral definido L_{th}, se considera que la neurona ha alcanzado un estado de equilibrio, y se detiene la plasticidad sináptica para evitar sobreajustes o inestabilidad.

Este enfoque tiene la ventaja de ser simple y totalmente local para cada neurona, lo que lo convierte en una manera útil de establecer criterios de parada durante el entrenamiento y para evaluar el grado de convergencia de la red.

4.2.5. Synapse.cpp

La clase Synapse modela las conexiones sinápticas entre neuronas en el simulador. Cada sinapsis conecta una neurona presináptica con una postsináptica y transmite los spikes

a través de una serie de ciclos temporales simulando el retardo sináptico. Además, lleva a cabo el cálculo de trazas presinápticas, función de forzamiento (*forcing function*) y actualiza los pesos sinápticos en función de la regla de aprendizaje STDP.

La clase se construye a partir de los siguientes parámetros:

- Un puntero compartido a la neurona presináptica (`preNeuron`).
- La constante de tiempo de la traza presináptica (`lambdaX`).
- El factor de escala de la traza (`alpha`).
- El peso inicial de la sinapsis (`weight`).
- El retardo sináptico en microsegundos (`delay`).
- El paso temporal del simulador (`dt`).

El método `obtainPreviousSpike()` permite simular el retraso en la llegada del spike. Cada vez que la neurona presináptica dispara, se añade un valor (`cycles` previamente calculado para cada sinapsis) a la cola (`spikesQ`). En cada iteración, estos valores se decrementan. Cuando uno de los valores llega a cero, se considera que ha llegado una spike y se retorna un 1 como señal de disparo.

Las trazas presinápticas (`preSynapticTrace`) acumulan información sobre la actividad reciente de una sinapsis. Su propósito es registrar la influencia que ha tenido recientemente la neurona presináptica sobre la postsináptica. Se actualizan en cada paso temporal mediante una ecuación de decaimiento, que simula cómo la información sobre spikes pasados se va desvaneciendo con el tiempo:

$$X(t) = \frac{1}{\lambda_x} - X(t) + \alpha s^{l-1}(t - \tau_d) \tag{4.6}$$

Donde $X(t)$ es la traza presináptica en el instante actual, λ_x es la constante de decaimiento, α es la amplitud de incremento cuando hay spike y $s^{l-1}(t - \tau_d)$ representa la spike presinaptica con un retardo τ_d.

Los impulsos presinápticos recientes generan una traza elevada, lo que implica que esa sinapsis ha tenido una influencia activa en el disparo postsináptico. Gracias a la traza presináptica, se logra que el aprendizaje sea sensible al tiempo relativo entre los disparos pre y post-sinápticos.

La *forcing function* se actualiza mediante la función `updateForcingFunction()` y representa el estímulo sináptico efectivo que una sinapsis aporta a la neurona postsináptica en un instante dado. Se calcula con la siguiente ecuación:

$$i_i(t) = \sum_{j=1}^{n^{l-1}} \sum_{d=1}^{m} \left(W_{i,j,d} s_j^{l-1}(t - \tau_d) - X_{i,j,d}(t) \right) \tag{4.7}$$

Esta expresión busca el equilibrio entre excitación e inhibición. Por un lado, un spike presináptico aporta un valor proporcional al peso sináptico, activando a la neurona postsináptica. Por otro lado, se resta la traza presináptica, que actúa como una forma de fatiga. Si una sinapsis ha estado muy activa recientemente (traza alta), su influencia actual se ve atenuada. Esta penalización temporal ayuda a evitar una sobreexcitación, permitiendo a la neurona integrar estímulos de manera más equilibrada.

El peso de la sinapsis puede ser actualizado dinámicamente durante el aprendizaje. El método `updateWeight()` modifica el valor actual del peso, sumando el valor calculado para cada sinapsis en la fase `STDP()` de la clase LIFneuron. Para que los pesos se puedan actualizar de forma sencilla, es necesario normalizar tanto las trazas presinápticas como los pesos entre los valores mínimos y máximos observados. Para calcular la normalización de ambos parámetros, se ha optado por utilizar las siguientes ecuaciones basadas en el algoritmo presentado en el artículo [Shantal et al., 2023].

$$\hat{X} = \frac{X - \min_X}{\max_X - \min_X} \tag{4.8}$$

$$\hat{W} = \frac{W - \min_W}{\max_W - \min_W} \tag{4.9}$$

Esto se realiza mediante los métodos `getNormPreSynapticTrace()` y `getNormWeight()`. Estas normalizaciones permiten aplicar criterios uniformes de aprendizaje y se utilizan también para el cálculo del MSE.

5. CAPÍTULO

Evaluación experimental

En este capítulo se presentan los experimentos realizados respecto a la implementación del simulador. A través de distintos tests se analiza la evolución de la actividad neuronal, la distribución de los spikes y la precisión por clase, utilizando como referencia una topología simplificada. Al final, se realiza un breve análisis sobre la totalidad de los resultados obtenidos y se determina si estos concuerdan con los objetivos del proyecto.

5.1. Entorno experimental

El entrenamiento realizado no tiene como objetivo alcanzar un rendimiento óptimo en la clasificación, sino validar el correcto funcionamiento del simulador y su comportamiento general. Para utilizar esta red en un entorno real, sería necesario un proceso exhaustivo de ajuste de hiperparámetros, tanto en lo referente a los parámetros de simulación, como a los hiperparámetros de las neuronas y la topología de la red.

Con este fin, se han realizado tres entrenamientos independientes tomando como base la topología representada en la Figura 5.1, de la cual se han replicado únicamente las capas *Input*, *SS-Conv* y *Dense*. Las capas restantes han sido omitidas para reducir la carga computacional y el tiempo de computo necesario para percibir resultados. La única diferencia entre los test realizados, es la semilla aleatoria utilizada para inicializar pesos y retardos, lo que permite evaluar el simulador frente a variaciones en la inicialización.

En cada test se han generado tres tipos de visualizaciones: un *heatmap* de activación, un

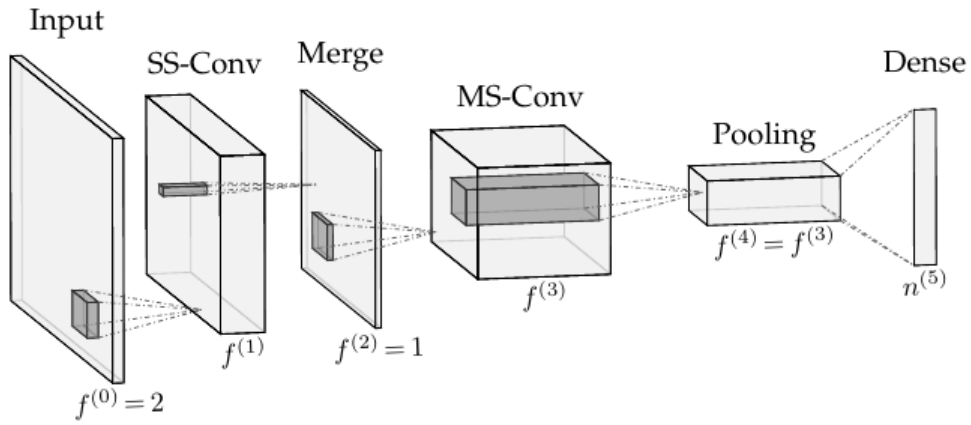

Figura 5.1: Topología usada como guía [Paredes-Vallés et al., 2020].

Clase	Neurona ganadora	Spikes ganadora	Total de spikes	Precisión
1	8	318	2754	11.55 %
2	8	189	864	21.88 %
3	8	66	206	32.04 %
4	8	18	61	29.51 %
5	8	4	8	50.00 %
6	4	4	10	40.00 %
7	2	1	2	50.00 %

Tabla 5.1: Precisión proyectada total (Test 1): 33.57 %.

gráfico de barras para percibir la precisión por clase, y un mapa de actividad temporal.

Para evaluar el rendimiento del sistema, se ha calculado la precisión por clase. Esta métrica se obtiene identificando, para cada clase C, la neurona que ha generado el mayor número de impulsos (neurona ganadora), y calculando el cociente entre el número de spikes generados por dicha neurona y el total de spikes registrados para esa clase:

$$\text{Precisión}_C = \frac{\text{Número de spikes de la neurona ganadora en clase } C}{\text{Total de spikes en clase } C} \tag{5.1}$$

5.2. Test 1

La visualización de la Figura 5.2, muestra una clara dominancia de la neurona 8, que responde con una alta frecuencia de disparos ante casi todas las clases. Esta saturación de la

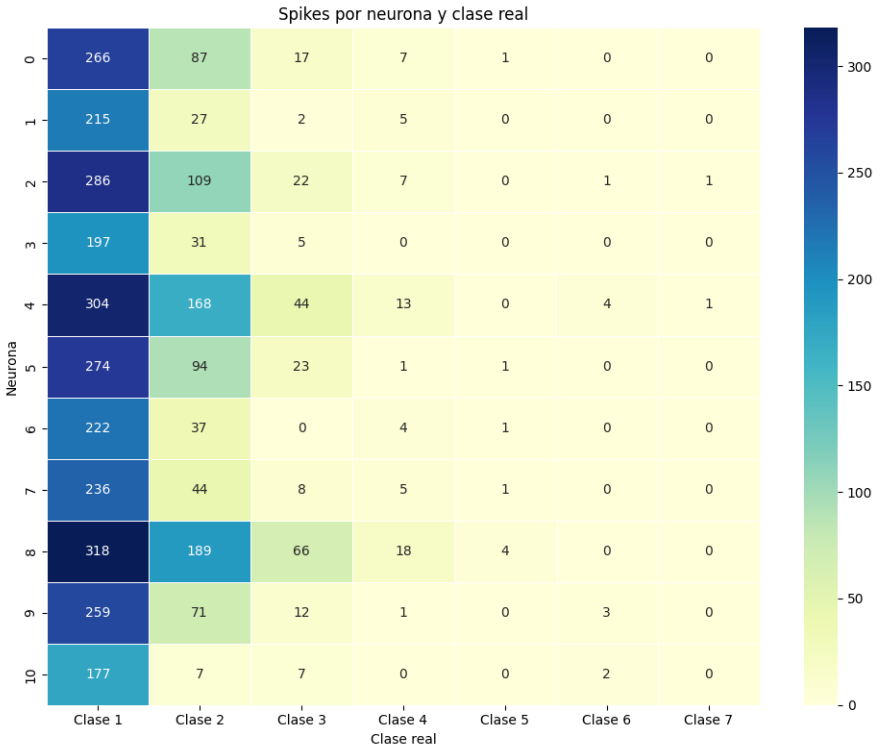

Figura 5.2: Mapa de calor (Test 1).

actividad en una única neurona revela que el proceso de especialización aún no se ha consolidado y que la red no ha distribuido adecuadamente las representaciones neuronales. El resto de neuronas permanecen con una actividad casi nula, lo cual limita la capacidad del sistema para realizar una clasificación adecuada. A pesar de ello, se puede confirmar que los mecanismos de propagación, aprendizaje y disparo están operando como se espera, ya que se observa una respuesta que sigue un patrón lógico y que se repite a lo largo del tiempo.

Como se aprecia en la Figura 5.3, la precisión por clase muestra un patrón desigual. Mientras que algunas clases minoritarias como la clase 5 (50 %) o la clase 7 (50 %) alcanzan valores notables de precisión, las clases más representadas, como la clase 1 (11.55 %) y clase 2 (21.88 %), muestran un rendimiento bajo. Este contraste sugiere que la red ha empezado a desarrollar asociaciones entre entradas y salidas, pero de forma poco generalizable. La proyección total de precisión del 33.57 % es pobre, pero coherente con una red entrenada sin ajuste fino de parámetros.

En la Figura 5.4, la actividad neuronal se incrementa de forma progresiva, lo que sugiere que las neuronas están siendo estimuladas correctamente durante el aprendizaje. Sin embargo, aunque hay actividad, siempre se activan las mismas neuronas, o un grupo re-

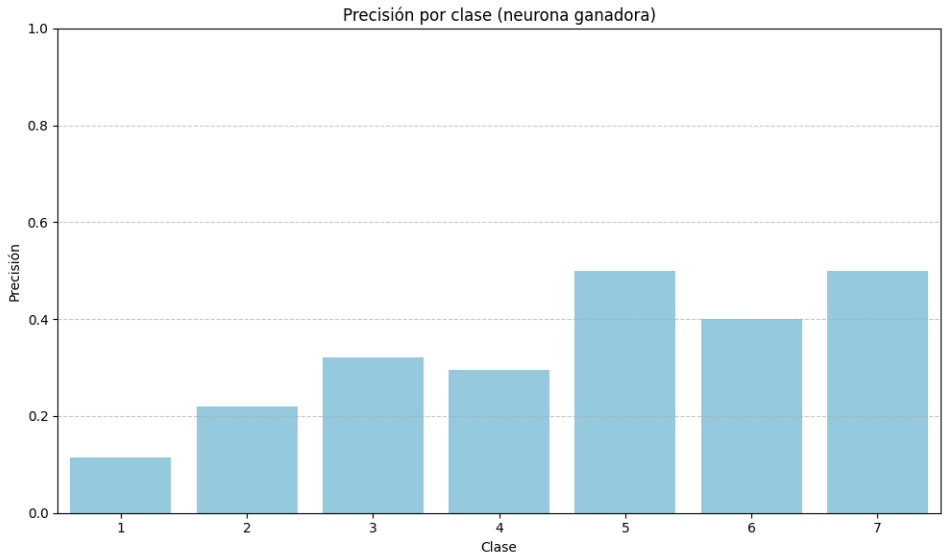

Figura 5.3: Diagrama de barras que representa la precisión alcanzada por clase (Test 1).

Clase	Neurona ganadora	Spikes ganadora	Total de spikes	Precisión
1	5	327	2799	11.68 %
2	5	194	965	20.10 %
3	5	92	273	33.70 %
4	7	18	78	23.08 %
5	5	3	11	27.27 %
6	6	2	5	40.00 %
7	2	1	4	25.00 %

Tabla 5.2: Precisión proyectada total (Test 2): 25.83 %

ducido, lo cual no es ideal. La red no está diferenciando bien entre distintos patrones o clases, y eso perjudica el rendimiento en la clasificación.

5.3. Test 2

En la Figura 5.5 se observa una mayor dispersión de la actividad neuronal. Aunque la neurona 5 sigue siendo la más activa y dominante, otras neuronas como la 7 (clase 4), la 6 (clase 6) y la 2 (clase 7) también participan con cierto nivel de actividad. Esto representa una mejora respecto al primer test, ya que indica que la red está empezando a distribuir la representación de clases entre múltiples neuronas, aunque sigue siendo limitada. El patrón sigue siendo sesgado, pero muestra un intento inicial de diversificación.

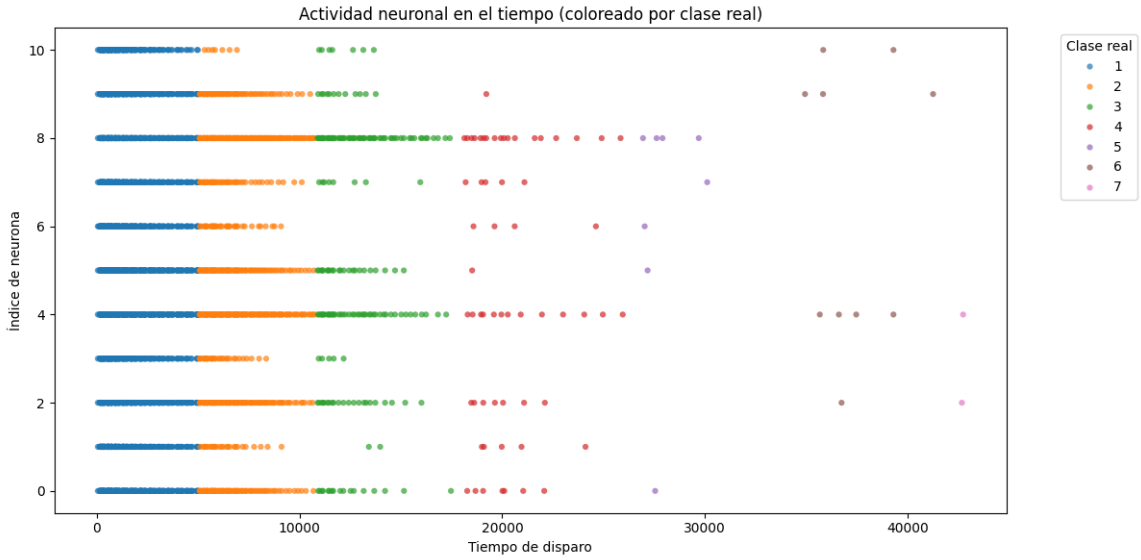

Figura 5.4: Diagrama de la actividad temporal (Test 1).

La precisión por clase en la Figura 5.6, presenta una tendencia similar a la del test anterior, aunque ligeramente más baja en términos globales (25.83 %). Las clases 3 y 6 obtienen mejores resultados (33.70 % y 40.00 % respectivamente), mientras que las más abundantes siguen mostrando rendimientos bajos. Este comportamiento puede estar indicando que, al no haber una competición o inhibición lo suficientemente fuerte, las neuronas más activas siguen absorbiendo la mayoría de los patrones sin permitir que otras se especialicen.

En la Figura 5.7, se percibe una dinámica similar a la anterior, con una evolución temporal que muestra estabilidad en el disparo de las neuronas y confirma que los mecanismos de refracción, retardo y actualización de potencial están funcionando de manera consistente. La repetitividad del patrón entre los test refuerza la fiabilidad del simulador.

5.4. Test 3

En la Figura 5.8, la neurona 10 monopoliza casi por completo la actividad en este test, especialmente en las clases más frecuentes. Esta excesiva concentración representa una regresión en comparación con el test anterior, lo cual es esperable al tratarse de una red con una inicialización aleatoria. Otras neuronas, como la 3, la 1 o la 0, presentan una actividad puntual en clases minoritarias, pero sin llegar a una especialización clara. Este resultado expone la sensibilidad de la red a la inicialización.

La Figura 5.9, presenta la precisión total es la más baja entre los tres test (24.07 %), con re-

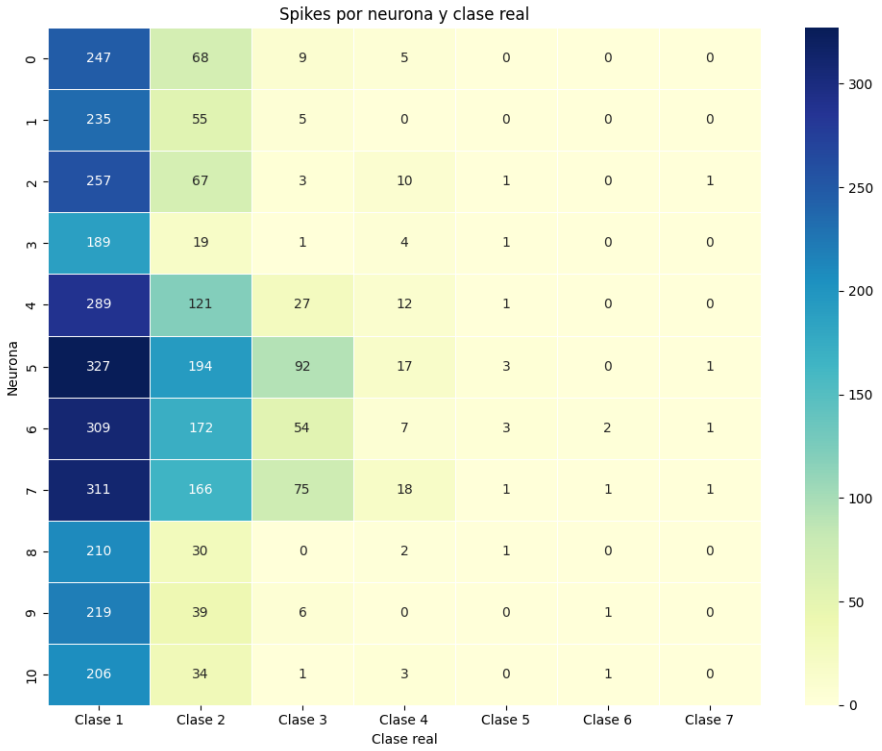

Figura 5.5: Mapa de calor (Test 2).

sultados pobres en las clases principales. No obstante, algunas clases con menos muestras, como la 5, 6 y 7, logran resultados aceptables. Esta disparidad sugiere que el simulador permite un aprendizaje parcial incluso en condiciones desfavorables. También confirma que la estructura básica de aprendizaje (STDP y WTA) está funcionando, aunque requiere una mejor configuración de la arquitectura y parámetros para explotar su potencial.

Al igual que en los test anteriores, en la Figura 5.10, la actividad a lo largo del tiempo sigue una trayectoria similar, con un incremento progresivo del disparo y una respuesta mantenida. La consistencia del comportamiento entre los test sugiere que los mecanismos internos del simulador son estables, y que el problema de rendimiento se encuentra más en la arquitectura y parámetros utilizados que en la ejecución del motor de simulación.

5.5. Análisis de resultados

Los resultados obtenidos en los test de entrenamiento demuestran de forma clara que el simulador desarrollado es capaz de ejecutar correctamente todas las fases clave de una red neuronal de impulsos: propagación de la actividad, aplicación de mecanismos

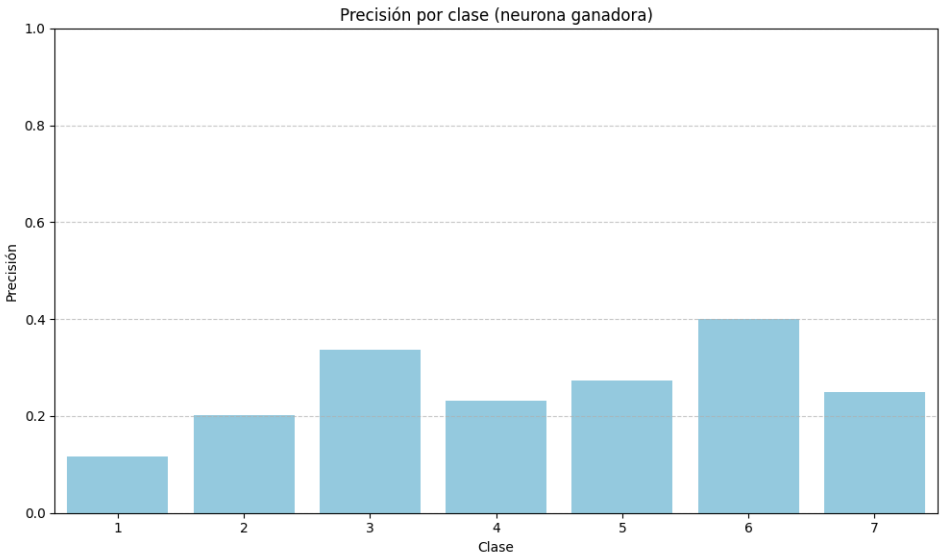

Figura 5.6: Diagrama de barras que representa la precisión alcanzada por clase (Test 2).

Clase	Neurona ganadora	Spikes ganadora	Total de spikes	Precisión
1	10	302	2814	10.73 %
2	10	144	931	15.47 %
3	10	48	207	23.19 %
4	3	11	49	22.45 %
5	1	3	10	30.00 %
6	0	2	6	33.33 %
7	0	1	3	33.33 %

Tabla 5.3: Precisión proyectada total (Test 3): 24.07 %

de aprendizaje sin supervisión (STDP), actualización del estado de las neuronas (modelo LIF) y registro de los disparos generados en la capa de salida. De está forma, se valida la solidez del diseño y la correcta implementación de los diferentes componentes del sistema, y respalda su utilidad como herramienta de simulación.

Pese a no ser el objetivo de este proyecto conseguir una alta precisión en el modelo entrenado, en su estado actual, el rendimiento clasificatorio de la red es limitado. La baja precisión alcanzada y la distribución desigual de la actividad entre las neuronas muestran que la red aún no es capaz de especializar adecuadamente sus salidas para resolver tareas complejas de clasificación. Esto no debe interpretarse como una limitación del simulador, sino como una consecuencia lógica de la ausencia de un proceso exhaustivo de ajuste de hiperparámetros. El hecho de que el comportamiento general de la red sea reproducible,

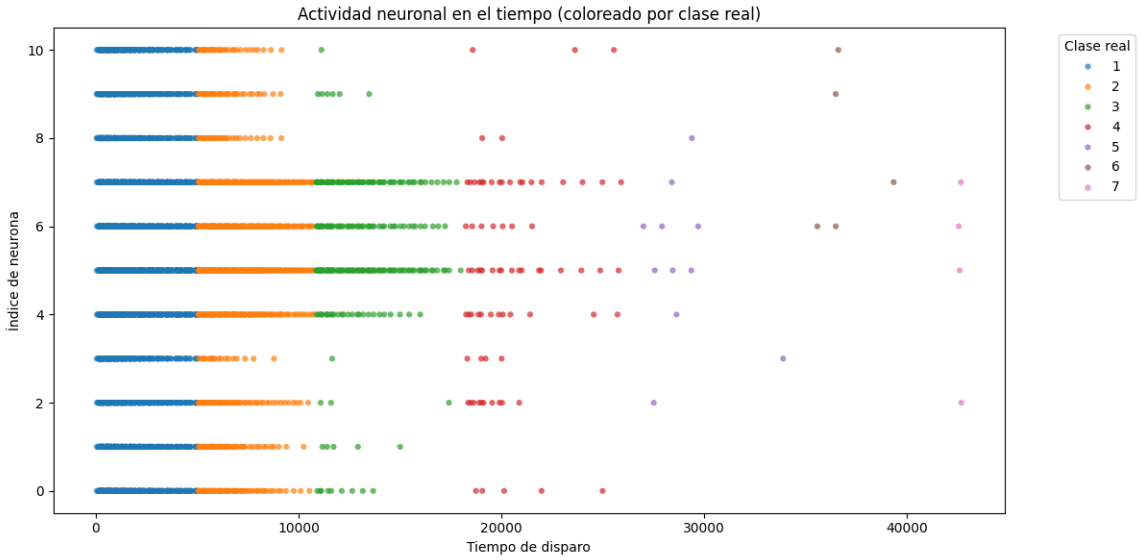

Figura 5.7: Diagrama de la actividad temporal (Test 2).

consistente entre cada test y sensible a las condiciones de inicialización, indica que el entorno simulado responde de forma fiel a los principios de las SNNs.

Estos test deben entenderse, por tanto, como una validación estructural y funcional del simulador, y no como una evaluación definitiva de su capacidad de clasificación. La riqueza de parámetros configurables, desde la topología de red hasta los valores neuronales individuales, permite una exploración extensiva del espacio de diseño.

En definitiva, el simulador no solo permite analizar el comportamiento interno de las neuronas, sino también observar con detalle la evolución del aprendizaje sin supervisión. Su modularidad, configurabilidad y estabilidad lo convierten en una herramienta útil para el estudio de los modelos implementados.

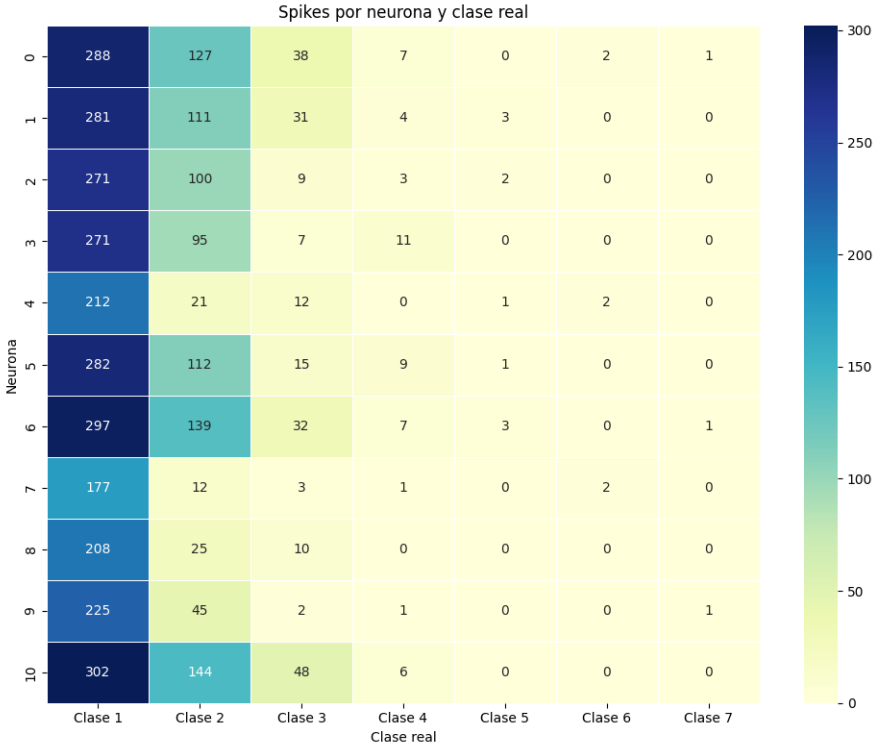

Figura 5.8: Mapa de calor (Test 3).

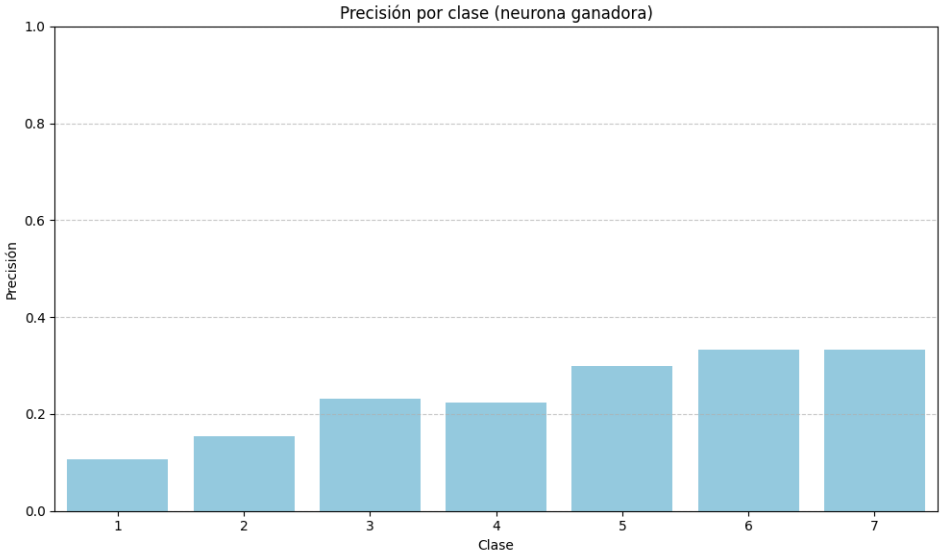

Figura 5.9: Diagrama de barras que representa la precisión alcanzada por clase (Test 3).

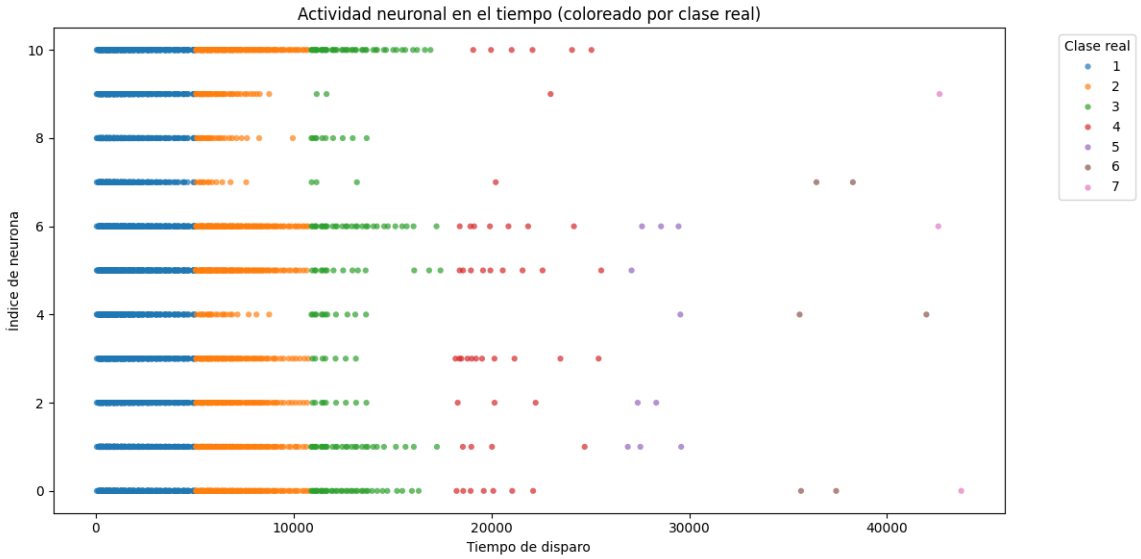

Figura 5.10: Diagrama de la actividad temporal (Test 3).

6. CAPÍTULO

Conclusiones

En este capítulo se presenta una reflexión final sobre el trabajo realizado, evaluando los resultados obtenidos en el desarrollo del simulador de redes neuronales de impulsos, así como las dificultades afrontadas a lo largo del proyecto. El simulador desarrollado se encuentra disponible públicamente en el repositorio de GitHub[1], donde se puede consultar el código y reproducir los experimentos realizados.

El desarrollo de este simulador ha supuesto un proceso complejo y exigente, que ha requerido una profunda comprensión tanto de los fundamentos biológicos que inspiran este tipo de redes como de los modelos computacionales que los representan. El objetivo principal ha sido implementar un sistema modular, flexible y funcional capaz de simular el comportamiento de una red SNN no supervisada basada en el modelo LIF.

El simulador ha sido validado mediante la aplicación al dataset DVS Gesture, obteniendo resultados que, si bien no son óptimos en términos de clasificación, cumplen con el propósito principal del proyecto y se sientan las bases para futuros desarrollos. Las pruebas han demostrado que el sistema es funcional, coherente con la teoría, y adaptable a diferentes configuraciones y topologías.

El desarrollo de este proyecto ha supuesto varios retos. Uno de los principales ha sido el diseño e implementación de la lógica interna de las neuronas. Este proceso ha implicado la definición de mecanismos de recepción de estímulos sinápticos, el cálculo del potencial de membrana, la gestión del estado de disparo y de refracción, la dinámica de competición a través del mecanismo WTA, así como la actualización sináptica mediante la regla STDP

[1] https://github.com/Ianfhca/TFG.git

y el seguimiento del aprendizaje con MSE. El llevar a cabo esta lógica desde cero, junto con la complejidad derivada del elevado número de parámetros, funciones y ecuaciones involucradas, ha requerido una cuidadosa implementación y validación constante para garantizar el correcto funcionamiento del sistema.

A pesar de la dificultad técnica, el resultado ha sido un simulador funcional que permite explorar, analizar y entrenar redes neuronales de impulsos, y que constituye una base sólida para futuras extensiones y mejoras.

7. CAPÍTULO

Trabajo futuro

En este capítulo se describen las principales líneas de mejora y ampliación detectadas durante la realización del proyecto. Estas propuestas no solo buscan incrementar el rendimiento del simulador, sino también extender sus capacidades funcionales, mejorar su usabilidad y facilitar el análisis de los resultados obtenidos.

Una de las principales mejoras pendientes es la optimización del rendimiento mediante técnicas de paralelización. Actualmente, la ejecución del simulador es secuencial, lo que limita su escalabilidad al trabajar con arquitecturas más profundas o con mayores volúmenes de datos. El uso de tecnologías como CUDA para la ejecución en GPU o bibliotecas paralelas en CPU permitiría acelerar los cálculos y reducir considerablemente el tiempo de simulación.

El simulador ha sido diseñado con soporte para arquitecturas de red flexibles, permitiendo la incorporación de diferentes tipos de capas. No obstante, en esta implementación se han omitido algunas capas intermedias y de codificación por limitaciones de tiempo, centrándose únicamente en las capas de entrada, convolucional y densa. Como trabajo futuro, se propone incorporar nuevas capas funcionales que permitan explorar arquitecturas más complejas.

Otro aspecto relevante es el ajuste fino (*tuning*) de los parámetros de simulación, hiperparámetros neuronales y estructura topológica. El rendimiento de una red SNN está altamente condicionado por valores como el umbral de disparo, constantes de tiempo, tasa de aprendizaje o distribución de retardos sinápticos. Como parte del trabajo futuro, se propone una optimización de todos estos parámetros, lo que implicaría evaluar el impacto

individual de cada parámetro sobre el comportamiento de la red y ajustar sus valores en función de criterios observables, como la tasa de disparo, la convergencia del aprendizaje o la distribución de la actividad entre neuronas. Además, se plantea la posibilidad de incorporar mecanismos de autoajuste, como la adaptación dinámica de la tasa de aprendizaje o del umbral de disparo en función del rendimiento. Estas estrategias podrían facilitar una evolución más eficiente del comportamiento de la red a lo largo del entrenamiento.

Finalmente las herramientas de visualización y análisis son un tanto básicas, por ello se propone el desarrollo de unas nuevas, que permitan entender mejor el comportamiento de la red. Estas herramientas facilitarían la interpretación de los resultados de entrenamiento, los patrones de activación y la evolución del aprendizaje. También se ha contemplado el uso de herramientas de análisis en tiempo real, aunque no se hayan implementado debido a la naturaleza de ejecución de estos modelos, como en este caso, que la simulación ha sido ejecutado en un clúster.

A. ANEXO

Uso del simulador

Este anexo tiene como objetivo explicar cómo utilizar el simulador desarrollado durante el proyecto, desde su preparación inicial hasta la evaluación de resultados.

Estructura del proyecto

El repositorio del proyecto está organizado de la siguiente forma:

- `documentation/`: contiene documentos, imágenes y recursos utilizados durante el desarrollo del simulador y la redacción de la memoria.
- `Project/`: carpeta principal del simulador, donde se encuentra el código fuente, scripts y archivos necesarios para la ejecución.

Compilación del proyecto

Dentro de la carpeta `Project/` se incluye un archivo `Makefile` que permite compilar automáticamente el simulador. Para compilar, simplemente acceda a la carpeta y ejecute:

Listing A.1: Compilar el simulador.

```
make
```

Tras la compilación, se generará un ejecutable llamado `SNN`.

Requisitos del dataset

El simulador utiliza el dataset **IBM DVS128 Gesture**, el cual no se incluye por razones de tamaño y licencia. Puede descargarse desde la siguiente dirección:

https://ibm.ent.box.com/s/3hiq58ww1pbbjrinh367ykfdf60xsfm8/folder/50167556794

Una vez descargado, debe colocarse en la ruta: `Project/dataset/DvsGesture/`

Configuración de la red

La carpeta `topology/` contiene archivos de configuración que definen la arquitectura y los parámetros de la red neuronal. Cada archivo representa una topología diferente y puede modificarse o ampliarse siguiendo la misma estructura que los ejemplos existentes.

Ejecución del simulador

Para ejecutar el simulador, hay que colocarse dentro de la carpeta `Project/` y utilizar el siguiente comando:

Listing A.2: Ejecutar el simulador.

```
./SNN -modo nombreTopología
```

Donde:

- `-modo` puede ser `-train` o `-test`
- `nombreTopología` es el nombre del archivo de topología sin la extensión `.txt`

Listing A.3: Ejemplo de uso.

```
./SNN -train 0_topology_1
```

Es importante tener en cuenta que para realizar una prueba en modo `-test`, primero debe haberse entrenado previamente la red con la misma topología.

Evaluación de resultados

Tras ejecutar el entrenamiento o test, se pueden evaluar los resultados mediante el script `evaluate_output.py`. Este generará en la carpeta `evaluation/` varios gráficos y archivos `.csv` con métricas relevantes para el análisis del comportamiento de la red.

Listing A.4: Instalar dependencias para la ejecución de evaluate_output.py.

```
pip install numpy matplotlib seaborn scikit-learn
```

Bibliografía

[Hodgkin AL, 1952] Hodgkin AL, H. A. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*.

[Izhikevich, 2003] Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572.

[Paredes-Vallés et al., 2020] Paredes-Vallés, F., Scheper, K. Y. W., and de Croon, G. C. H. E. (2020). Unsupervised learning of a hierarchical spiking neural network for optical flow estimation: From events to global motion perception. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(8):2051–2064.

[Shantal et al., 2023] Shantal, M., Othman, Z., and Bakar, A. A. (2023). A novel approach for data feature weighting using correlation coefficients and min–max normalization. *Symmetry*, 15(12).

[Thorpe, 1990] Thorpe, S. (1990). Spike arrival times: A highly efficient coding scheme for neural networks. *Parallel Processing in Neural Systems and Computers*.