Adaptación de arquitecturas profundas a problemas no estacionarios

Tesina de Grado
Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Leonardo R. Morelli

Director:
Dr. Guillermo Grinblat

Co-director:
Dr. Pablo Granitto

Febrero de 2013

Resumen

Muchos sistemas reales de gran interés práctico son claramente *no estacionarios*. Por otro lado, se sabe que las *arquitecturas profundas* pueden ser mucho más eficientes a la hora de representar ciertas funciones.

El objetivo general de esta tesina fue el estudio del rendimiento de estas arquitecturas en el ámbito de los problemas no estacionarios y su adaptación a los mismos. Para ello se usaron SRBM (Stacked Restricted Boltzmann Machines) y, para comparar con una arquitectura poco profunda de las más usadas, SVM (Support Vector Machines).

Existen diversos métodos para abordar problemas no estacionarios, el usado en este trabajo fue SEA (Streaming Ensemble Algorithm) con SRBM, para el cual se desarrolló una política de inclusión especial, y con SVM.

Para llevar a cabo las pruebas, se diseñaron dos grupos de datasets no estacionarios a partir de MNIST y NORB, ampliamente usados y conocidos. Los resultados estuvieron dentro de lo esperado y dan esperanza para seguir investigando estos y otros temas relacionados. El rendimiento general para MNIST (estudiado con más detalle) de SEA con SRBM fue superior al que se obtuvo con SVM. Sin embargo, en el caso de cambios recurrentes no hubo mejoras notables.

Índice

1. Introducción

El campo del Aprendizaje Automatizado (*Machine Learning*), es parte central de la nueva revolución tecnológica basada en el uso inteligente de la información. Su objetivo primario es la construcción de algoritmos que automáticamente mejoren su eficiencia en la solución de un problema a través de la experiencia acumulada [28], con una mínima cantidad de intervención humana. El Aprendizaje Automatizado integra conocimientos de disciplinas diversas como Inteligencia Artificial, Estadística, Teoría de Control, Ciencia Cognitiva, Neurobiología, etc. Uno de los principales problemas que se investigan en esta área es el de reconocimiento de patrones o clasificación [9], que corresponde a asignar una clase particular a una nueva observación.

La mayoría de los métodos de análisis de datos utilizados asumen como premisa básica la *estacionaridad* de los mismos (es decir, que el fenómeno bajo análisis no cambia en el tiempo). Sin embargo, muchos sistemas reales de gran interés práctico son claramente *no estacionarios*, como por ejemplo cualquier propiedad relacionada a la meteorología (que depende de la estación del año) o el problema de detección temprana de fallas en líneas de producción (que depende del desgaste de las máquinas).

Otra característica que presentan estos métodos es que se basan en lo que puede definirse como *arquitecturas poco profundas* (Redes Neuronales con una capa oculta, Support Vector Machines –SVM–, Árboles de Decisión, etc.), aunque desde hace bastante tiempo [13] se sabe que las *arquitecturas profundas* [5] pueden ser mucho más eficientes a la hora de representar ciertas funciones. Esto se debe, probablemente, a que sólo recientemente se pudo desarrollar un método de aprendizaje efectivo para estas arquitecturas [16]. Este desarrollo despertó gran interés en estas arquitecturas [5], aunque todavía no se han utilizado en problemas no estacionarios. El objetivo general de esta tesina fue el estudio del rendimiento de estas arquitecturas en el ámbito de los problemas no estacionarios y su adaptación a los mismos.

1.1. Objetivos

El primer objetivo de esta tesina fue la comparación de estas técnicas con los métodos tradicionales, utilizando como base problemas no estacionarios que presenten *highly varying functions* (funciones de variabilidad alta) [5], en particular, con métodos especialmente diseñados para el tratamiento de problemas no estacionarios pero basados en arquitecturas poco profundas.

El segundo objetivo fue el desarrollo de técnicas específicas para este tipo de problemas. Se investigó en qué forma se podía integrar la información aportada por los datos antiguos al entrenamiento de un modelo de mayor profundidad. Se esperaba que las técnicas aquí desarrolladas tuvieran un rendimiento superior a las tradicionales en los problemas no estacionarios con *highly varying functions*.

1.2. Organización del trabajo

Se comenzará dando un panorama de los conceptos y algoritmos utilizados en el trabajo y las características de los problemas no estacionarios. Luego se detalla el

tipo de red profunda utilizado y su adaptación al problema. A continuación se presentan las pruebas realizadas y la preparación de los datasets usados, los resultados obtenidos y finalmente, algunas conclusiones y trabajos futuros.

1.3. Desarrollo de software

La plataforma utilizada fue GNU/Linux y las implementaciones de los algoritmos se hicieron en el lenguaje C++ con la biblioteca para álgebra lineal *Armadillo* [3].

2. Conceptos generales

Aquí se explican algunos temas que son necesarios conocer para una mejor comprensión del trabajo, como ser el aprendizaje automatizado, redes neuronales, arquitecturas profundas y RBM (Restricted Boltzmann Machines), SVM, otros clasificadores conocidos, los problemas no estacionarios y sus algoritmos específicos.

2.1. Aprendizaje Automatizado

A fines de los '50, Arthur Samuel, que fue uno de los primeros en definir *Machine Learning*, decía que es el campo de estudio que otorga a las computadoras la capacidad de aprender sin ser explícitamente programadas. En 1997, Tom Mitchell dio una definición más formal [28]: *se dice que un programa* ***aprende*** *de la experiencia* E *con respecto a cierto tipo de tarea* T *y una medida de performance* P*, si el desempeño en tareas de* T *según la medida* P*, mejora con la experiencia* E.

El Aprendizaje Automatizado es una rama de la Inteligencia Artificial, pero es inherentemente un campo multidisciplinario. Involucra conceptos de probabilidad y estadística, teorías de complejidad computacional, de control, de información, filosofía, psicología, neurobiología, y otros campos. Busca desarrollar algoritmos capaces de aprender a solucionar problemas a partir de la experiencia acumulada. Esta experiencia se traduce en el conjunto de ejemplos observados –datos de entrenamiento. Para esto, se han creado varias técnicas, dentro de las más comunes están el aprendizaje supervisado, no supervisado, semi-supervisado, por refuerzo, etc.

En el **aprendizaje supervisado**, los datos de entrenamiento consisten de tuplas $(\mathbf{x}, y)$, donde $\mathbf{x}$ es la entrada obtenida de una distribución de probabilidad desconocida, e y, la salida que se espera obtener del algoritmo tras el entrenamiento. Si la salida admite sólo dos valores, tenemos un problema de *clasificación binaria*. Cuando admite un número finito de valores, es de *clasificación multiclase*. Y en el caso continuo, donde la salida $y \in \mathbb{R}^n$, el problema es de *regresión*.

En el **aprendizaje no supervisado** contamos sólo con la primera parte de las tuplas, es decir, la entrada $\mathbf{x}$. En este caso se busca encontrar características o estructuras ocultas en los datos. Algunos problemas relacionados con este método son la estimación de densidad de probabilidad que generó los datos, la compresión, reducción de dimensionalidad, y *clustering*, que trata de agrupar los datos según algún criterio de similitud.

El **aprendizaje semi-supervisado** cae entre los dos métodos recién vistos. No todos los datos tienen asignada una etiqueta, a veces es desconocida, o conocerla tiene un costo muy elevado. También pueden existir restricciones que indiquen por ejemplo, que dos instancias pertenecen a la misma clase. Por último, el **aprendizaje por refuerzo** hace que un agente aprenda del *feedback* (retroalimentación) que recibe del entorno como resultado de las acciones que efectúa sobre éste.

Veamos algunos ejemplos de problemas. Se tiene un conjunto de registros (que incluyen edad, nivel de glucosa en sangre, etc.) de pacientes que han sido (o no) diagnosticados con diabetes y se desea determinar si *nuevos* pacientes son diabéticos

o no a partir de sus registros. También podríamos querer estimar el precio de una casa, para esto contamos con una base de datos de viviendas con información sobre superficie cubierta, antigüedad, ubicación, etc., y el precio actual. Otro ejemplo podría ser el de crear un filtro de correo no deseado dados correos electrónicos etiquetados como spam o no spam.

En definitiva, sea X el conjunto de todos los ejemplos posibles de cierta experiencia, e Y el conjunto de todos sus resultados posibles, entonces un algoritmo típico de entrenamiento supervisado en Machine Learning buscará una función $f : X \to Y$ a partir de un conjunto de datos de entrenamiento T, donde $T \subset X \times Y$, tal que para una instancia cualquiera de X, f nos de la respuesta correcta en Y.

La función deberá generalizar sobre nuevas instancias de X que no hayan sido vistas en $Dom(T)$. Como el espacio de búsqueda puede ser muy grande y los datos de entrenamiento no abundantes, suele reducirse el espacio para mejorar el aprendizaje.

Para evaluar este aprendizaje se utilizan dos medidas, el *error de entrenamiento* y el *error de validación*. Para esto se toman todos los datos disponibles y se los divide en dos conjuntos, uno de entrenamiento (más grande) para encontrar f y otro de validación, para controlar el comportamiento de f frente a casos nuevos y tratar de evitar *sobreajustar* los datos de entrenamiento. Los errores mencionados, entonces, se calculan sobre estos dos conjuntos.

Por ejemplo, tenemos n datos que corresponden a una función cuadrática, las muestras fueron tomadas uniformemente de un intervalo en X, y sus valores en Y fueron leídos con algo de ruido, ver Figura 1.

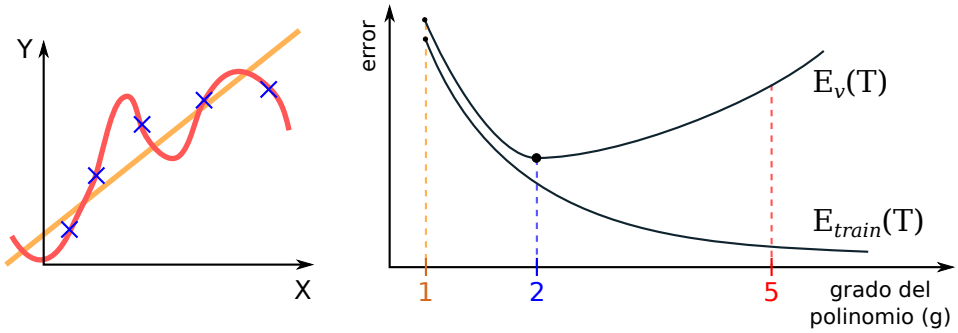

Figura 1: Sesgo y varianza (sobreajuste). $E_v(T)$ es el error de validación usando T y $E_{train}(T)$ el error de entrenamiento.

En una primera instancia decidimos buscar la función lineal que mejor ajuste los datos, y vemos que tanto el error de entrenamiento y de validación son altos, aquí cometimos un error por *sesgo*.

Si en cambio buscamos el polinomio de grado $\leq n$ que mejor ajuste los datos, vamos a encontrar uno como muestra la curva roja que tendrá un error de entrenamiento muy bajo, pero veremos que $E_v(T) \gg E_{train}(T)$, por lo que muy probablemente fallará en la generalización, es decir, el algoritmo sobreajustó los datos de entrenamiento, junto con el ruido presente en la muestra. Hubo un error por *varianza*.

En ambos casos hay presente una *suposición*: la función óptima es lineal o bien es polinómica. Estas suposiciones, a priori, que hace el sistema acerca del concepto a aprender, conforman la *base inductiva* del mismo.

2.2. Redes neuronales

Las *redes neuronales artificiales* (RNA –o ANN por sus siglas en inglés) proveen un método práctico y general para aprender funciones de valores reales, discretos o vectoriales a partir de un conjunto de ejemplos. Algoritmos como *Backpropagation* usan descenso por el gradiente para optimizar los parámetros de la red con el objetivo de ajustar mejor los ejemplos de entrenamiento. El aprendizaje de las redes es robusto a errores en los datos de entrenamiento y ha sido aplicado exitosamente a problemas como el reconocimiento de voz, de caras y el aprendizaje de estrategias de control de robots, vehículos y procesos.

Una RNA está compuesta por un conjunto de unidades simples –o neuronas– densamente interconectadas, donde cada una toma un número de valores reales como entrada y produce un único valor real como salida, que a su vez puede ser la entrada de otras unidades conectadas. Si bien el estudio de las RNA fue motivado en parte por cómo está formado el sistema nervioso central, estas no han sido modeladas con total fidelidad y son conocidas ciertas inconsistencias con el sistema biológico. Por ejemplo, una neurona artificial tendrá como salida un solo valor, mientras que una neurona biológica produce una compleja señal compuesta por una serie de impulsos nerviosos.

Una unidad u esencialmente toma la entrada $\bar{x} \in \mathbb{R}^n$, las interconexiones $\bar{w} \in \mathbb{R}^n$ (o pesos) y computa un valor $h = a(g(\bar{x}, \bar{w}))$, que es la salida de esta unidad. Una forma común de g es $g(\bar{x}, \bar{w}) = \sum_{i=0}^{n} x_i w_i$, donde $x_0 = 1$ y $w_0 = b$ es el *bias* o sesgo de la unidad, y $a(\theta)$ es la función de *activación* que puede variar de acuerdo al problema. Algunas de las más conocidas son la función signo, sigmoidea y softmax. Como ejemplo, veamos en la Figura 2 cómo es una unidad sigmoidea.

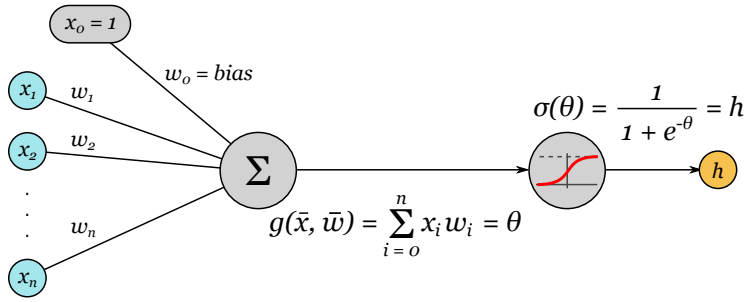

Figura 2: Unidad sigmoidea.

De acuerdo a cómo son las conexiones entre unidades, se cuentan varias categorías de redes: feedforward (con conexiones[1] a unidades de capas superiores), recurrentes (las conexiones pueden formar un ciclo dirigido, se permiten conexiones a capas inferiores o a unidades de la misma capa, ej.: Boltzmann Machine [17]), modulares (colección de pequeñas redes que cooperan o compiten para resolver un problema), entre otras.

En la Figura 3 vemos una simple red *feedforward* con dos capas ocultas (en rojo) de 3 unidades la primera y 2 la segunda. En gris se ven los sesgos b_i^1, con $i = [1, 2, 3]$ para las unidades de la primer capa oculta, b_i^2, con $i = [1, 2]$ para las unidades de la

[1]Considerar que en una conexión de a hacia b, la información fluye de a hacia b.

segunda capa oculta y b_1^3 para la unidad de salida. Cada w_{ij}^l corresponde al peso de la conexión entre la unidad i de la capa l con la unidad j de la capa anterior $l-1$.

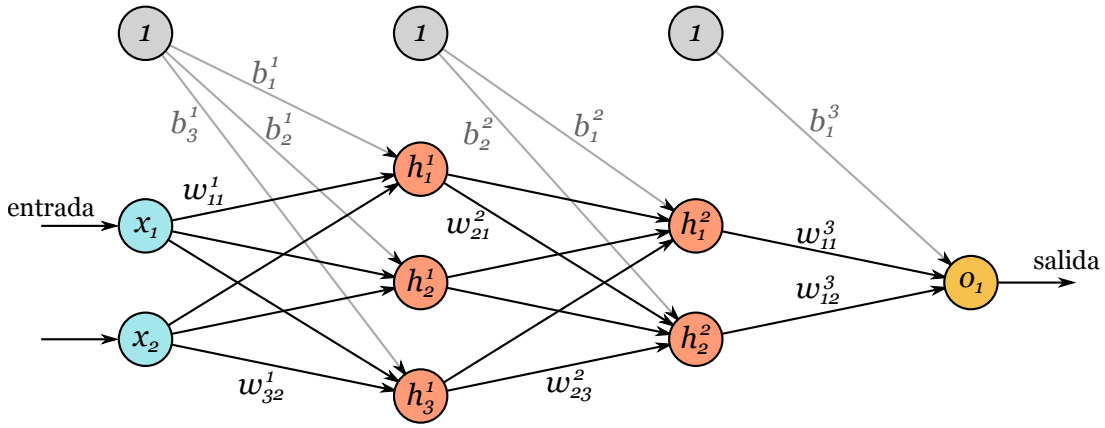

Figura 3: Red feedforward de dos capas ocultas.

2.3. Arquitecturas profundas

Las redes neuronales de una o dos capas ocultas, y otras arquitecturas poco profundas, presentan limitaciones en cuanto a las funciones que pueden representar, como las funciones de variabilidad alta. Entonces surge la necesidad de estudiar algoritmos para entrenar modelos profundos con varios niveles de abstracción con el fin de encontrar mejores representaciones de estas funciones.

Se llaman *arquitecturas profundas* a las composiciones de muchas capas de componentes adaptivos no lineales. Estas arquitecturas permiten la representación de una amplia familia de funciones de manera más compacta que las arquitecturas poco profundas utilizadas habitualmente. En particular, desde hace varios años se conocen varias funciones simples que son extremadamente difíciles de representar con una arquitectura poco profunda mientras que con una arquitectura de la profundidad adecuada se pueden representar fácilmente. Un ejemplo típico es la función paridad (ver por ejemplo [5] y las referencias citadas en ese trabajo). El uso de arquitecturas inapropiadas hace que estas funciones sean muy difíciles de aprender y se necesite una enorme cantidad de datos para aproximarlas eficazmente.

A pesar de las potenciales ventajas, el uso de las arquitecturas profundas fue muy limitado hasta hace poco tiempo debido a la dificultad de entrenar arquitecturas de más de tres capas. Hinton y otros [16] propusieron recientemente un método de entrenamiento que supera esta dificultad. El mismo se basa en el uso de una etapa de *pre-entrenamiento*, esto es entrenando con una heurística *greedy*[2] cada par de capas con el algoritmo de *Máquinas de Boltzmann Restringidas* [36] (como se detalla en la sección 2.4). Esto permite inicializar los parámetros de la red profunda en una región cerca del óptimo buscado, para luego aplicar algoritmos de descenso

[2]Voraz, codiciosa.

por el gradiente. Una red profunda así entrenada obtiene en cada una de sus capas abstracciones de distintos niveles de los datos presentados en la entrada.

A partir de la introducción de este método ha surgido un gran interés en el estudio de las arquitecturas profundas. Nuevos algoritmos de aprendizaje fueron propuestos [33, 25, 1]. Erhan y otros [10] realizaron un detallado estudio experimental con el objeto de determinar si la etapa de pre-entrenamiento tiene un efecto regularizador u optimizador, agregando información adicional para imponer una cierta distribución a priori de los parámetros del modelo a entrenar. Otros desarrollos llevaron estas técnicas al campo del aprendizaje semi-supervisado [40].

Las arquitecturas profundas fueron aplicadas sobre todo en problemas de reconocimiento de imágenes. Se realizaron trabajos en varias direcciones, por ejemplo el uso de imágenes naturales [32], la búsqueda de robustez frente a variaciones comunes, como el ruido aleatorio o el ocultamiento de parte de la imagen [38], o la adaptación de estas técnicas para ser usadas con imágenes de mayor tamaño [26].

2.4. Restricted Boltzmann Machines (RBMs)

En la sección anterior se mencionaron las RBM como un elemento clave para entrenar redes profundas, aquí las veremos en mayor detalle, empezando por su origen.

Una *máquina de Boltzmann* [17] es una red neuronal recurrente de conexiones simétricas, cuyas unidades (o neuronas) son activadas estocásticamente, que permite aprender regularidades complejas presentes en los datos de entrenamiento.

Al agregar ciertas restricciones a estas conexiones, se obtiene una *máquina de Boltzmann restringida* [36] (RBM), que consiste en una capa de unidades ocultas $\mathbf{h}$ binarias y otra de unidades visibles $\mathbf{v}$ binarias o de valores reales, tales que las conexiones entre ambas capas (W) son simétricas y no se permiten conexiones entre unidades del tipo visible-visible u oculta-oculta. De esta manera se forma un grafo bipartito como en la Figura 4, donde $\mathbf{b}$ y $\mathbf{c}$ son vectores de *biases* para las unidades ocultas y visibles respectivamente y W es la matriz de pesos entre ellas.

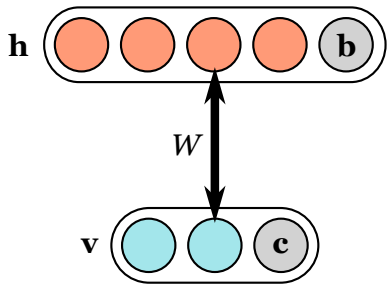

Figura 4: Restricted Boltzmann Machine.

Más formalmente, una RBM es un modelo generativo basado en una función de energía $E(\mathbf{v}, \mathbf{h})$ sobre todas las unidades de la red, y por la cual asigna una probabilidad a cada posible par de vectores visibles y ocultos:

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})}$$

donde Z es la función de partición dada por la suma sobre todos los posibles pares de vectores $\mathbf{v}$ y $\mathbf{h}$:

$$Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}$$

y la función de energía [19] es:

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{c}'\,\mathbf{v} - \mathbf{b}'\,\mathbf{h} - \mathbf{h}'\,W\,\mathbf{v}$$

Hinton [15] propuso un método para entrenar RBMs, basado en una iteración del algoritmo de *muestreo de Gibbs*, que consiste en asignar un vector de entrenamiento a las unidades visibles y actualizar, estocásticamente y en paralelo, los estados de las unidades ocultas. Luego de muestrear estos estados, se realiza una *reconstrucción* actualizando cada unidad visible, en paralelo. Finalmente, se vuelven a actualizar las unidades ocultas. Es por las restricciones antes mencionadas que pueden inferirse en paralelo y fácilmente los valores h_j dado el vector de entrada $\mathbf{v}$, es decir, hallar $p(\mathbf{h}|\mathbf{v})$, ya que los h_j son condicionalmente independientes dado $\mathbf{v}$; y de manera análoga, es fácil reconstruir $\mathbf{v}$ a partir de $\mathbf{h}$, calculando $p(\mathbf{v}|\mathbf{h})$. [25, 6]

Este procedimiento está inspirado en el método *Contrastive Divergence* [6], el cual se basa en una aproximación del gradiente de la *log-likelihood* (logaritmo de la probabilidad de los datos de entrenamiento) de los parámetros del modelo, a través de una cadena de Markov que comienza con el último ejemplo visto y en donde la transición es un paso del algoritmo de muestreo de Gibbs. Pero como aquí la convergencia se da en infinitos pasos que consisten en calcular $p(\mathbf{h}|\mathbf{v})$ y luego la reconstrucción $p(\mathbf{v}|\mathbf{h})$, al entrenar una RBM se aplica una versión más rápida denominada CD-k, donde k denota el número de pasos de muestreo de Gibbs. En la práctica, se han obtenido resultados muy satisfactorios con $k = 1$.

Ya tenemos un método efectivo de entrenamiento, pero una RBM por sí sola está limitada en cuanto a lo que puede representar. Su verdadero poder se ve cuando se usan "apiladas" para formar una red de creencia profunda (o *Deep Belief Network*), un modelo generativo de varias capas. Aquí, cada capa contiene un conjunto de unidades binarias o con valores continuos, y están conectadas con todas las unidades de las capas adyacentes pero no entre las de una misma capa. A esta pila se le agrega una capa final que mapeará la última capa oculta con las unidades de salida. En la Figura 5 se muestra una SRBM (*Stacked RBM*) con tres RBM apiladas, en donde cada unidad de una capa está conectada con todas las unidades de la capa superior. Notar que la RBM^i consta de la capa oculta $\mathbf{h}^i$ con sus biases $\mathbf{b}^i$ y la capa inferior (que sería la visible) con sus biases $\mathbf{c}^{i-1}$, según indican los colores usados.

El algoritmo propuesto por Hinton [16] para las SRBM comienza con una etapa no-supervisada, en la que se entrena una capa por vez de manera *greedy*, tratándolas como RBMs y usando las activaciones de la capa anterior como entrada de la siguiente. Los pesos y biases calculados para cada RBM, se utilizan como valores iniciales de una red (a la que se agrega una capa final como se mencionó antes) que será entrenada de forma supervisada con alguna versión de descenso por el gradiente tradicional. En este trabajo se usó el método del *gradiente conjugado*.

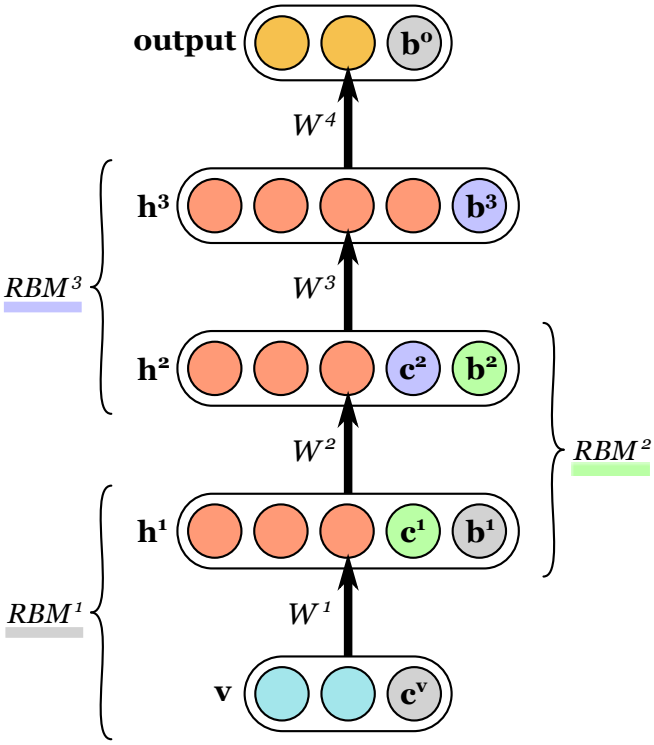

Figura 5: Red feed-forward profunda y la interpretación como Stacked Restricted Boltzmann Machines.

2.5. Support Vector Machines

Algunos problemas de las RNA como la difícil y costosa búsqueda del mínimo global (o al menos un buen mínimo local), son solventados en las *máquinas de vectores soporte* (SVM, por su sigla en inglés), ya que siempre encuentran el mínimo global. Además, cuentan con una sólida base matemática, una interpretación geométrica simple y son ampliamente usadas en la actualidad obteniendo buenos resultados.

Las SVM comprenden una técnica útil y reconocida para resolver problemas de clasificación binaria, aunque puede extenderse a clasificación multiclase y regresión. Si bien la clasificación es lineal, puede usarse como no-lineal por medio de *kernels* especiales que mapean los datos de entrenamiento de su espacio original a uno de mayor dimensión. El resultado del algoritmo es un hiperplano que separará lo mejor posible los datos de una y otra clase. Como pueden existir infinitos separadores que cumplan esto, se busca aquel que pueda generalizar con menor error, el que maximice el *margen*.

Entonces, supongamos un problema de clasificación binaria, donde T es el conjunto de ejemplos de entrenamiento (separable linealmente) y h es la ecuación de un hiperplano.

$$T = \{(\mathbf{x}_i, y_i), \text{ con i} = 1,\ldots,\text{n}\},\ \mathbf{x}_i \in \mathbb{R}^d,\ y_i \in \{-1, +1\}$$

$$h) \quad \mathbf{w} \cdot \mathbf{x} + b = 0$$

En la Figura 6[3] (a) se muestran varios hiperplanos que separan linealmente los ejemplos positivos de los negativos. Los ejemplos de cada clase más cercanos a un hiperplano se llaman *vectores soporte* y son los que definen dicho hiperplano, en la Figura se muestran en azul. La distancia $\rho = d_+ + d_-$ entre vectores soporte de distintas clases al hiperplano es el *margen* del mismo.

Pero ¿cuál es el mejor? Para una futura generalización, el mejor será, intuitivamente, el que separe los ejemplos de forma "maximal".

Para cada $(\mathbf{x}_i, y_i)$, debe satisfacerse:

$$\begin{cases} \mathbf{w} \cdot \mathbf{x}_i + b \leq -\rho/2 & \text{si } y_i = -1 \\ \mathbf{w} \cdot \mathbf{x}_i + b \geq \rho/2 & \text{si } y_i = 1 \end{cases} \quad \Leftrightarrow \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq \rho/2 \qquad (1)$$

y como para vectores soporte $\mathbf{x}_s$, el lado derecho de 1 es una igualdad, luego de escalar $\mathbf{w}$ y b por $\rho/2$, se obtiene que la distancia de $\mathbf{x}_s$ a un hiperplano es:

$$r = \frac{y_s(\mathbf{w} \cdot \mathbf{x}_s + b)}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$$

Luego, con $\rho = 2\,r$, y teniendo $\mathbf{w}$ y b escalados, el problema se reduce a maximizar este margen:

$$\rho = \frac{2}{\|\mathbf{w}\|} \quad \text{donde para todo } i = 1, \ldots, n: \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

o bien, reformulando, a encontrar $\mathbf{w}$ y b tal que:

$$\begin{cases} \Phi(\mathbf{w}) = \|\mathbf{w}\|^2 = \mathbf{w} \cdot \mathbf{w} & \text{es minimizada, y} \\ y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 & \text{para todo } (\mathbf{x}_i, y_i) \in T \end{cases}$$

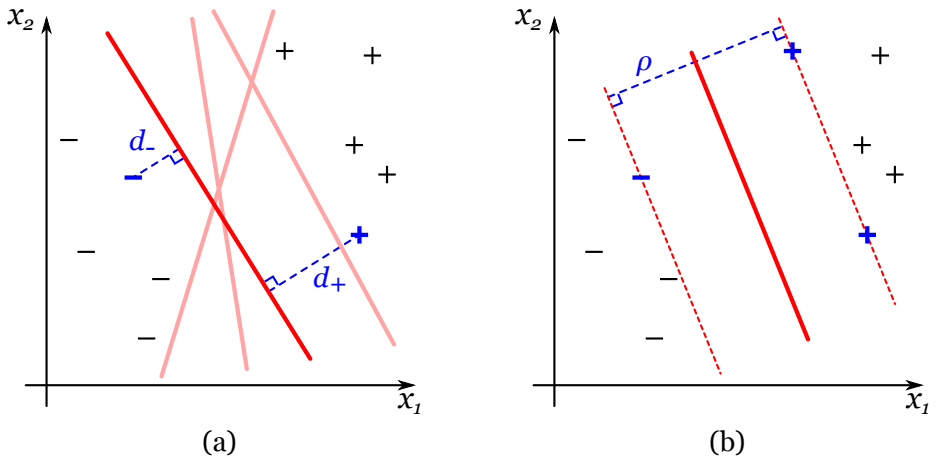

Figura 6: (a) Distancias a vectores soporte y múltiples separadores. (b) Hiperplano óptimo, su margen y vectores soporte.

En el caso en que el dataset no sea linealmente separable, ver Figura 7, se agregan *variables de holgura* (*slack variables* en inglés) ξ_i para permitir incorporar ejemplos

[3] $d = 2$ para simplificar la visualización.

difíciles o con ruido. A estas variables se le asocia un costo C que dará más importancia a maximizar el margen o ajustar mayor cantidad de datos de entrenamiento. El problema se modifica de esta manera:

$$\begin{cases} \Phi(\mathbf{w}) = \mathbf{w} \cdot \mathbf{w} + C \sum \xi_i & \text{es minimizada, y} \\ y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i & \text{para todo } (\mathbf{x}_i, y_i) \in T \text{ y } \xi_i \geq 0 \end{cases}$$

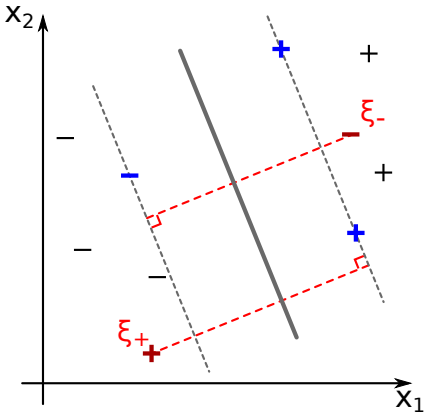

Figura 7: Variables *slack*.

Cuando un dataset es muy difícil de separar linealmente, como en la Figura 8, se hace uso de una función *kernel* que mapea los datos a un espacio de mayor dimensión $\mathcal{H}$ en el que sí sean separables, mediante una transformación $\varphi : \mathbb{R}^d \to \mathcal{H}$. La función *kernel* es equivalente a un producto interno en un espacio de mayor dimensión: $K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_j)$. Una de las más usadas y recomendada es la función RBF gaussiana (*Radial Basis Function*), cuya expresión es $e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}$, donde $\gamma = 1/(2\,\sigma^2)$. Con valores altos de γ, puede evitarse el sobreajuste de los datos de entrenamiento, y por otro lado, valores bajos permiten reproducir mejor funciones de alta variabilidad o muy irregulares. Este meta-parámetro debe ser optimizado para cada problema en particular.

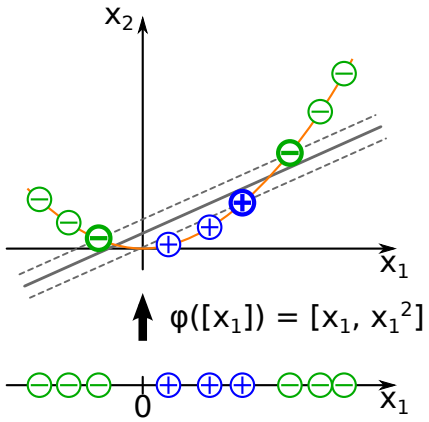

Figura 8: Problema de difícil resolución en su espacio original (abajo). Mismo problema aplicando un *kernel* (arriba).

2.6. Otros

Árboles de decisión

Este método [28] es usado para aproximar funciones de valores discretos, en donde la función aprendida es representada por un árbol de decisión. Cada nodo del árbol especifica una prueba de un determinado atributo de las instancias y cada rama se corresponde con los posibles valores que puede tomar ese atributo. Las hojas tendrán el valor con que una instancia es clasificada.

La clasificación se realiza desde la raíz hacia alguna hoja, probando en cada nodo un atributo específico y siguiendo por la rama que corresponda al valor de ese atributo en la instancia dada. Esto se repite hasta llegar a una hoja con la clasificación final.

Los árboles de decisión son robustos a errores en los datos de entrenamiento y también pueden faltar valores de algunos atributos.

Para determinar qué atributo se probará en cada nodo, se define la propiedad estadística *information gain* (ganancia de información) que mide la capacidad de un atributo para separar los datos de entrenamiento según sus clases. Esta propiedad se usa en cada nodo para seleccionar un atributo entre los candidatos, mientras crece el árbol.

La implementación usada en este trabajo es **C4.5** de Quinlan [31].

Naïve Bayes

El clasificador *Naïve Bayes* (NB) [28] se basa en una simplificación del clasificador bayesiano estándar que supone que los valores de los atributos que forman una instancia $\mathbf{x} = \langle a_1, a_2 \dots a_n \rangle$ son *condicionalmente independientes* dada su clase c_j. Es decir, la probabilidad de observar la conjunción $a_1, a_2 \dots a_n$ dada la clase c_j, es el producto de las probabilidades de los atributos individuales: $P(a_1, a_2 \dots a_n | c_j) = \prod_i P(a_i | c_j)$.

Entonces, para una nueva instancia, el clasificador usa la siguiente ecuación:

$$c_{NB} = \arg\max_{c_j \in C} P(c_j) \prod_i P(a_i | c_j)$$

donde c_{NB} denota el valor de salida del clasificador y C es un conjunto finito de valores de salida. Notar que el aprendizaje en este método se reduce a estimar los términos $P(c_j)$ y $P(a_i|c_j)$ a partir de sus frecuencias en los datos de entrenamiento. Estas probabilidades comprenden el concepto o la función aprendida. Cabe destacar que al contrario de otros métodos vistos, aquí no hay una búsqueda en un espacio de posibles soluciones, sino que la función aprendida se forma simplemente contando la frecuencia de las distintas combinaciones de valores presentes en los datos.

2.7. Problemas no estacionarios

Al hablar de *no estacionaridad*, se hace referencia a algo que cambia con el correr del tiempo. En problemas reales, muchas veces sucede que la causa del cambio está oculta, por lo que dicho cambio debe inferirse de los propios datos. Y estos datos

fluyen de manera secuencial en el tiempo, haciendo que los modelos construidos sobre datos antiguos necesiten ser actualizados o reentrenados constantemente. En inglés, este problema es llamado *concept drift*, es decir, es el concepto siendo estudiado el que sufre variaciones, y esto sucede cuando la distribución de la que se muestrean los datos varía. Kuh y otros [24] definen el cambio en la distribución de las clases de los datos, no en la distribución de los datos en sí, que es estática.

Un ejemplo típico es el pronóstico del clima, las reglas usadas pueden variar drásticamente en cada estación y más complicado aún, con las corrientes del Niño y la Niña que provocan excesivas lluvias y sequía respectivamente, en nuestra región, cada 2 a 8 años. Otro ejemplo son los patrones de preferencias de compras, que pueden cambiar con el tiempo dependiendo del día de la semana, disponibilidad de alternativas, inflación, etc.

Los algoritmos específicos para este problema deben ser capaces de identificar un cambio en el concepto estudiado sin tener que conocer directamente el cambio subyacente en la distribución. Pero si hacemos alguna presunción acerca del tipo de cambio que no se condiga con la realidad, tendremos un error por *sesgo*. Por otro lado, si no se asume nada acerca de cómo evoluciona la distribución de datos, es difícil distinguir cuándo un ejemplo distinto refleja algún tipo de cambio, o ruido; en este caso el error que se obtenga será por *varianza*. Un sistema ideal debería adaptarse rápidamente a los cambios, ser robusto al ruido y distinguirlo de un verdadero cambio, y poder usar la experiencia previa al detectar contextos recurrentes [37, 39, 41, 42].

2.7.1. Definición

Supongamos que tenemos un conjunto de datos $[(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)]$ que fueron tomados de a una instancia por vez, con $(\mathbf{x}_i, y_i)$ obtenido en el tiempo $t_i < t_{i+1}$ (esto es, los datos están ordenados por tiempo). Tendremos un problema no estacionario si cada $(\mathbf{x}_i, y_i)$ es tomado de una distribución de probabilidad P_i no necesariamente igual para todo i.

Habitualmente, tres tareas son abordadas por la literatura específica: el problema de *modelado*, esto es inferir cómo fue variando la distribución de los datos a través del tiempo; el problema de *predicción*, o cómo obtener la mejor descripción posible del estado actual; y el de *extrapolación*, esto es poder predecir cómo evolucionará la distribución de los datos en el futuro [4]. A su vez, los métodos utilizados son diseñados para atacar problemas con determinado tipo de cambios: graduales [7, 14, 12] o abruptos [34, 23, 8].

2.7.2. Tipos de cambio

La literatura normalmente distingue dos categorías de cambios que pueden darse en problemas reales: *abruptos* y *graduales*. Por ejemplo, al acercarse las elecciones presidenciales, una persona que normalmente no tiene interés en la política, puede querer informarse más acerca de los candidatos para tomar una decisión; o en una línea de producción, una herramienta que sufra un desgaste lento puede causar un cambio gradual en la calidad de la pieza final.

Consideremos dos distribuciones de datos P_1 y P_2, y una secuencia de n ejemplos $[(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)]$.

Si existe un momento t, tal que para $i < t$, los datos $(\mathbf{x}_i, y_i)$ son muestreados de P_1 y para $t \leq i$, de P_2, entonces estamos ante la presencia de un cambio *abrupto*.

Los cambios serán *graduales* cuando la distribución de la que provienen los datos se vaya transformando con el pasar del tiempo, en otra. Un ejemplo es cuando en lugar de un instante t, existe un período Δt durante el cual los datos son muestreados de P_1 y P_2 con probabilidad p y $1 - p$ respectivamente, con p creciendo en Δt desde 0 hasta 1.

Una situación común a ambos tipos se da cuando la "nueva" distribución ya hubiere aparecido en el pasado. En este caso, se dice que el cambio es *recurrente*.

2.7.3. Aplicaciones

Existen muchas aplicaciones a problemas reales, a continuación se muestran algunas agrupadas según cuatro categorías generales [42]:

Monitorización y control Usualmente emplea aprendizaje no supervisado para detectar comportamientos anormales. Incluye detección de intrusos, fraude en telecomunicaciones y transacciones financieras, inundaciones, estado del tráfico, etc.

Asistencia personal Estas aplicaciones se ocupan de personalizar información para usuarios o clientes por ejemplo, y tienen la particularidad de que las etiquetas de los datos son flexibles y por ende, el costo de cometer un error es bajo. Incluye sistemas de recomendación (noticias, películas), categorización y filtrado de spam, entre otras.

Toma de decisiones Aquí los datos suelen ser escasos y se necesita muy buena precisión en la decisión final y los costos de errores son altos. Incluye la evaluación de solvencia por créditos, efecto de antibióticos en pacientes y diagnósticos, etc. La respuesta real para compararla con la decisión tomada, suele estar disponible luego de cierto tiempo.

Inteligencia Artificial y Robótica Los cambios tienen lugar en entornos dinámicos y los agentes que interactúan con estos deben adaptarse. Incluye robots, vehículos, electrodomésticos inteligentes, etc.

2.8. Algoritmos para problemas no estacionarios

Existen diversos métodos para abordar problemas no estacionarios [11, 42], dentro de los más populares se encuentran los que usan un *ensamble de clasificadores* [22, 30]. En este caso, cada clasificador es entrenado con un subconjunto distinto de datos –cuando vienen en lotes, por ejemplo–, y de acuerdo a su rendimiento individual u otra medida de calidad, tendrán mayor o menor peso en la votación del ensamble, pudiendo ser eliminados del conjunto si dicha medida no es lo suficientemente buena.

Otros enfoques mencionados en la literatura incluyen el uso de *ventanas temporales* –o *deslizantes*– [41, 20] y el *pesado de muestras* [23, 21].

Las *ventanas temporales*, de una dada longitud, toman las muestras de datos más recientes, y para cada ventana se construye un nuevo clasificador o se adapta uno anterior. Un problema a considerar es el tamaño de la ventana; si es muy grande, el tiempo que se tomará el algoritmo para adaptarse a un cambio puede ser excesivo; o si es muy pequeña, se corre el riesgo de que sea muy sensible al ruido. Para afrontar este problema suele utilizarse una ventana de tamaño variable.

Los algoritmos basados en el *pesado de muestras*, asignan un peso variable a cada muestra. Éste indicará la importancia de un ejemplo en el presente, la cual decrecerá a medida que pase el tiempo. Suelen usarse funciones lineales o exponenciales que decrezcan con el tiempo.

En estos métodos, los clasificadores siempre son entrenados con datos que ya se han visto en el pasado. Además, estos métodos pueden combinarse para formar otros más complejos. En este trabajo se estudiaron e implementaron tres: *Streaming Ensemble Algorithm* (SEA), *Dynamic Weighted Majority* (DWM) y *Gradual Forgetting* (GF) [30, 22, 23].

2.8.1. SEA

El *Streaming Ensemble Algorithm* es un método de clasificación basado en múltiples clasificadores, que se construyen a partir de una secuencia de lotes de datos de entrenamiento. Estos clasificadores luego son combinados en un ensamble de tamaño fijo para clasificar datos no vistos. Aquellos miembros del ensamble que van perdiendo calidad son reemplazados por nuevos y mejores clasificadores. Originalmente, estos eran árboles de decisión [30], en este trabajo se usaron además Redes Profundas y SVM's.

El algoritmo es bastante sencillo, ver Figura 9. Para cada lote de datos leído, ajusta un árbol independiente de los anteriores, que son combinados en un ensamble de tamaño fijo, donde los árboles nuevos reemplazan a los viejos según una medida de calidad basada en su nivel de error y diversidad. En particular, si el árbol clasifica bien un punto del lote actual, su puntaje sube respecto a cómo lo clasifica el resto del ensamble. Así, los puntos que son fáciles o imposibles de clasificar correctamente tienen poca incidencia en la medida.

La *calidad* se calcula sobre el lote actual siendo evaluado, según estos porcentajes de votos:

P_1 = porcentaje de la clase con más votos
P_2 = porcentaje de la segunda clase con más votos
P_C = porcentaje de la clase correcta
P_T = porcentaje de la clase predicha por el nuevo árbol T
P_{E_j} = porcentaje de la clase predicha por el miembro E_j

Si el ensamble E y el nuevo árbol T aciertan, la calidad de T se incrementa en $1 - |P_1 - P_2|$. Es decir, si la votación fue pareja, el nuevo árbol recibe un puntaje alto, ya que puede influir directamente sobre votos futuros. Si T acertó pero E no, la calidad de T se incrementa en $1 - |P_1 - P_C|$. Finalmente, si T se equivoca (sin

```
Streaming-Ensemble-Algorithm(d, EMS)

    d: cantidad de datos a leer en cada iteración
    m: cantidad de miembros del ensamble
    EMS: Ensemble Max Size, cantidad máxima de expertos

1.  m ← 0
2.  while hay-datos-disponibles()
3.      D ← leer-datos(d)
4.      C_i ← entrenar(crear-nuevo-experto(), D)
5.      clasificar(C_{i-1}, D)
6.      for j ← 1, ..., m
7.          clasificar(E_j, D)
8.      if (m < EMS)
9.          m ← m + 1
10.         E_m ← C_{i-1}
11.     else if (para algún j : calidad(C_{i-1}) > calidad(E_j))
12.         E_j ← C_{i-1}
13.     end
14.     C_{i-1} ← C_i
15. end
16. end.
```

Figura 9: Pseudocódigo de SEA. E_j representa el j-ésimo clasificador del ensamble E.

importar la decisión de E), su calidad se decrementa en $1 - |P_C - P_T|$. Para los miembros del ensamble, la calidad se calcula de la misma manera que para T.

Entre los árboles del ensamble y el evaluado con el nuevo lote (C_{i-1}), aquel que obtenga el menor puntaje será eliminado. Todos los clasificadores tienen el mismo peso dentro del conjunto.

2.8.2. DWM

El algoritmo *Dynamic Weighted Majority* [22] también se basa en un conjunto de clasificadores, ver Figura 10. Se crean expertos dinámicamente en respuesta a los cambios en performance. El algoritmo entrena todos los clasificadores con los datos nuevos que vayan apareciendo, creando nuevos si la performance del conjunto no es adecuada. A medida que un clasificador se equivoca, su peso se reduce (normalizando los pesos al final de cada iteración de manera que el máximo sea 1) y si cae debajo de cierto umbral es eliminado del conjunto. Este umbral, el factor por el que se reducen los pesos y el período tras el cual se crean, eliminan y cambian los pesos de los expertos, son parámetros ajustables.

Usa cuatro mecanismos para tratar el problema de la no estacionaridad: los expertos del ensamble son entrenados de manera online, les asigna un peso a cada uno basado en sus rendimientos individuales, los elimina también según el rendimiento, y agrega nuevos expertos de acuerdo al rendimiento global del ensamble.

2.8.3. GF

El método de *Gradual Forgetting* [23] sugiere la introducción de una función de olvido basada en el tiempo, haciendo que las últimas muestras vistas sean más

Dynamic-Weighted-Majority($\{\vec{x}, y\}_n^1, c, \beta, \theta, p$)

$\{\vec{x}, y\}_n^1$: datos de entrenamiento, vector de features y clase
$c \in \mathbb{N}^*$: cantidad de clases, $c \geq 2$
β: factor por el que se decrementan los pesos, $0 \leq \beta < 1$
θ: umbral para eliminar expertos
p: período entre eliminación, creación y actualización de peso de expertos
$\{e, w\}_m^1$: conjunto de expertos y sus pesos
$\Lambda, \lambda \in \{1, \ldots, c\}$: predicciones globales y locales
$\vec{\sigma} \in \mathbb{R}^c$: suma de predicciones pesadas para cada clase

1. $m \leftarrow 1$
2. $e_m \leftarrow$ crear-nuevo-experto()
3. $w_m \leftarrow 1$
4. **for** $i \leftarrow 1, \ldots, n$ // Iterar sobre casos de entrenamiento
5. $\vec{\sigma} \leftarrow \vec{0}$
6. **for** $j \leftarrow 1, \ldots, m$ // Iterar sobre expertos
7. $\lambda \leftarrow$ clasificar$(e_j, \vec{x}_i)$
8. **if** $(\lambda \neq y_i$ **and** i mód $p = 0)$
9. $w_j \leftarrow \beta w_j$
10. $\sigma_\lambda \leftarrow \sigma_\lambda + w_j$
11. **end**
12. $\Lambda \leftarrow \arg\max_j \sigma_j$
13. **if** $(i$ mód $p = 0)$
14. $w \leftarrow$ normalizar-pesos(w)
15. $\{e, w\} \leftarrow$ eliminar-expertos$(\{e, w\}, \theta)$
16. **if** $(\Lambda \neq y_i)$
17. $m \leftarrow m + 1$
18. $e_m \leftarrow$ crear-nuevo-experto()
19. $w_m \leftarrow 1$
20. **end**
21. **end**
22. **for** $j \leftarrow 1, \ldots, m$
23. $e_j \leftarrow$ entrenar$(e_j, \vec{x}_i, y_i)$
24. **mostrar** Λ // Clasificación global del i-ésimo caso
25. **end**
26. **end.**

Figura 10: Pseudocódigo de DWM.

significativas para el clasificador que las más viejas. La función de olvido le asigna un peso a cada dato de entrenamiento según su aparición a lo largo del tiempo. Los clasificadores (C_{gf}) deben ser modificados para que puedan manejar datos pesados.

Una mejora es combinar esta función con una ventana temporal de tamaño fijo o variable para directamente olvidar los datos que estén por fuera, es decir, los que sean más viejos que una determinada *edad*, y pesar sólo los que estén dentro de la ventana.

En la Figura 11 se muestra una idea de este enfoque que utiliza una función de olvido en una ventana.

Respecto a la función, puede ser lineal, logarítmica, exponencial, etc. En este trabajo se implementó una función lineal con un parámetro (k) que representa el factor de decremento (incremento) del peso de la primera (última) observación y que tiene estas restricciones: $k \in [0, 1]$, $w_i \geq 0$ y $\frac{\sum_{i=1}^{n} w_i}{n} = 1$.

Gradual-Forgetting(d, v, k)

d: cantidad de datos a leer en cada iteración
v: tamaño máximo de la ventana, $d \leq v$
k: factor para la función (lineal) de olvido gradual
V: la ventana mantiene los últimos v datos leídos

1. $D \leftarrow$ leer-datos(d)
2. $V \leftarrow$ crear-ventana(D)
3. $W \leftarrow$ actualizar-pesos(k, size(V))
4. $C_{gf} \leftarrow$ entrenar(crear-nuevo-experto-gf(), V, W)
5. **while** hay-datos-disponibles()
6. $D \leftarrow$ leer-datos(d)
7. clasificar(C_{gf}, D)
8. $V \leftarrow$ mover-ventana(V, D)
9. $W \leftarrow$ actualizar-pesos(k, size(V))
10. $C_{gf} \leftarrow$ entrenar(crear-nuevo-experto-gf(), V, W)
11. **end**
12. **end.**

Figura 11: Pseudocódigo de GF.

3. Adaptación de Redes Profundas

Para poder utilizar una red profunda con alguno de los algoritmos vistos en la sección 2.8, fue necesario hacer modificaciones y establecer nuevas políticas de entrenamiento. Fueron tomados en consideración los algoritmos SEA, DWM y GF, siendo el primero de estos el único elegido para las pruebas por ser más rápida su adaptación, en la sección 3.1 se detalla la política de inclusión al SEA.

En DWM, no hubiera sido necesario realizar alguna modificación en la red, ya que el paso de reentrenarla con nuevos datos podría haber sido simplemente entrenar la red partiendo con los parámetros (pesos y biases) calculados para los datos anteriores. El problema es que en DWM, todos los expertos del ensamble deben ser reentrenados, y como una red tarda un tiempo considerable en ser entrenada, la adaptación aquí hubiera sido temporalmente prohibitiva en las pruebas, en comparación con lo que tardaría el SEA.

En el caso de GF, el algoritmo es más sencillo pero requiere modificar el algoritmo de la red profunda para incluir una política de pesado de muestras. Esto hubiera sido posible pero en esta oportunidad se optó por la opción con SEA.

3.1. Política de inclusión al SEA

La idea detrás de la política aplicada en este algoritmo supone que las features detectadas en las capas inferiores se mantienen útiles a lo largo del tiempo aunque el problema final cambie, por lo que tiene sentido hacer un pre-entrenamiento con todos los puntos, mientras que la etapa de fine tuning se encarga de adaptar estas features generales al problema actual. Para esto se mantiene una única red *base* entrenada sólo de manera no-supervisada con cada nuevo lote D. Luego, en el momento de crear un experto nuevo, se realiza una copia de la red base, la cual es entrenada de forma supervisada (ver línea 4 de la Figura 9), completando así el entrenamiento del nuevo experto. En la etapa no-supervisada, se entrena de a una capa por vez en forma *greedy*, es decir, recién después de terminar de entrenar la i-ésima capa durante un número de épocas, es cuando se propagan los datos hasta esa capa y se prosigue con el entrenamiento de la capa $i + 1$. Larochelle y otros [25] proponen un método similar para el caso de contar con infinitas muestras.

Por lo tanto, se realiza un entrenamiento no-supervisado y uno supervisado por cada iteración del SEA. A continuación se detalla brevemente la complejidad temporal que requiere el uso de SEA con redes profundas.

3.2. Análisis de complejidad

El análisis tiene como objetivo aproximar la complejidad temporal que agrega el uso de redes profundas (en este caso SRBM) al algoritmo SEA. No es tenido en cuenta lo que es independiente del clasificador elegido, ya que no aporta un costo significativo, como puede comprobarse en la rapidez con que terminaron las pruebas usando SVM.

Las operaciones tenidas en cuenta fueron sumas, multiplicaciones, divisiones, exponenciales, logaritmos, comparaciones y generación de números aleatorios en los

bloques de código que incluyen mayormente operaciones entre matrices o vectores. A los tiempos de estas operaciones se le asignaron los siguientes costos:

$$\begin{array}{llll} T(sum/resta) = c_s & T(mul) = c_m & T(div) = c_d & T(comp.) = c_c \\ T(exp) = c_e & T(log) = c_l & T(rand) = c_r & T(copia) = c_p \end{array}$$

Las operaciones frecuentes de matrices que aparecen en el algoritmo y fueron incorporadas al análisis, tienen estos costos:

$$\begin{array}{lll} T_\times(a,b,c) = T(M_{a\times b} \cdot M_{b\times c}) & = abc\, c_m + a\,(b-1)\,c\, c_s \\ T_+(a,b) = T(M_{a\times b} + M_{a\times b}) & = ab\, c_s \\ T_C(a,b) = T(\mathrm{col\,sum}(M_{a\times b})) & = (a-1)\, b\, c_s \\ T_R(a,b) = T(\mathrm{row\,sum}(M_{a\times b})) & = a\,(b-1)\, c_s \\ T_\sigma(a,b) = T(\mathrm{sigmoide}(M_{a\times b})) & = ab\,(c_e + c_s + c_d) \\ T_\#(a,b) = T(\mathrm{sample}(M_{a\times b})) & = ab\,(c_r + c_c) \\ T_S(a,b) = T(\mathrm{softmax}(M_{a\times b})) & = ab\,(c_e + c_s + c_d) - b\, c_s \end{array}$$

donde $M_{a\times b}$ es una matriz de tamaño $a \times b$ y las operaciones sample, sigmoide y softmax son aplicadas a cada elemento de la matriz, estando las dos últimas definidas como:

$$\mathrm{sigmoide}(m_{ij}) = \frac{1}{1 + e^{-m_{ij}}}$$

$$\mathrm{softmax}(m_{ij}) = \frac{e^{m_{ij}}}{\sum_k e^{m_{kj}}}$$

El algoritmo simplificado al análisis se ve en la Figura 12. Se usaron las siguientes referencias[4]:

ensembleMaxSize	s	capas	L
tamaño dataset	d	unid. en capa j	$l_{j=0,\ldots,(L-1)}$
batch size	b_s	unid. en capa de entrada	l_0
clases	k	unid. en capas ocultas	$l_{j=1,\ldots,(L-2)}$
épocas (no-sup.)	e	unid. en capa de salida	l_{L-1}
mini-batch size (no-sup.)	b_u	momentum	μ
llamadas a $function\_grad$ (sup.)	n_f	learning rate	η
variables	a, b, c	weight cost	ω

unsupervisedTraining

El costo T_u de esta etapa, está dado por las actualizaciones de parámetros en $rbmUpdate$, los cálculos del error para cada mini-batch y las propagaciones de los datos a medida que se van entrenando las capas.

[4] unid.: unidades

```
1.  for i ← 1, ..., (d/b_s)
2.      D ← leer-datos(b_s)
3.      unsupervisedTraining(redBase, D) : {
4.          for j ← 1, ..., (L − 1)
5.              for ep ← 1, ..., e
6.                  for b ← 1, ..., (b_s/b_u)
7.                      rbmUpdate(l_j)
8.                      miniBatchError(b)
9.                  end
10.             propagarDatosACapa(j)
11.         end
12.     }
13.     red_i ← redBase      # se copia
14.     supervisedTraining(red_i, D) : {      # método Gradiente Conjugado
15.         aplanar()
16.         gc_optimizar(function_grad())      # vía alglib
17.         desaplanar()
18.     }
19.     test(red_{i−1}, D)
20.     for c ← 1, ..., s
21.         test(ensamble[c], D)
22. end
```

Figura 12: Algoritmo simplificado para el análisis.

rbmUpdate En las siguientes líneas se muestran las operaciones involucradas donde H y V son matrices cuyas columnas se corresponden con **h** y **v** para una capa y muestra determinadas; H_r y V_r corresponden a la reconstrucción; y C y B son la replicación de las columnas **c** y **b** de la misma capa.

1. $H = \text{sigmoide}(B + W \cdot V)$
2. $H_s = \text{sample}(H)$
3. $V_r = \text{sigmoide}(C + W' \cdot H_s)$
4. $H_r = \text{sigmoide}(B + W \cdot V_r)$
5. $\Delta W = \mu\, \Delta W + \eta\, ((H \cdot V' - H_r \cdot V_r')/b_u - \omega\, W)$
6. $\Delta c = \mu\, \Delta c + (\eta/b_u)\ \text{rowsum}(V - V_r)$
7. $\Delta b = \mu\, \Delta b + (\eta/b_u)\ \text{rowsum}(H - H_r)$
8. $W = W + \Delta W$
9. $c = c + \Delta c$
10. $b = b + \Delta b$

T_r es el costo para las operaciones de todas las capas:

$$
\begin{aligned}
T_r = \sum_{j=1}^{L-2} [\, & 2\,(T_\times(l_j, l_{j-1}, b_u) + T_+(l_j, b_u) + T_\sigma(l_j, b_u)) \;+\; T_\#(l_j, b_u) \\
& +\; T_\times(b_u, l_j, l_{j-1}) + T_+(b_u, l_{j-1}) + T_\sigma(b_u, l_{j-1}) \\
& +\; 2\,T_\times(l_j, b_u, l_{j-1}) + 3\,T_+(l_j, l_{j-1}) + (3\,c_m + c_d)\,l_j\,l_{j-1} \\
& +\; T_+(b_u, l_{j-1}) + T_C(b_u, l_{j-1}) + 2\,c_m\,l_{j-1} + T_+(1, l_{j-1}) \\
& +\; T_+(l_j, b_u) + T_R(l_j, b_u) + 2\,c_m\,l_j + T_+(l_j, 1) \\
& +\; T_+(l_j, l_{j-1}) + T_+(1, l_{j-1}) + T_+(l_j, 1)]
\end{aligned}
$$

miniBatchError Aquí hay una suma y producto elemento-a-elemento de matrices, una suma de filas y la suma de un vector. En el costo T_{er} se cuentan las operaciones para todas las capas.

1. $mbEr = mbEr + \text{rowsum}(\text{colsum}((V - V_r)^2))$

$$T_{er} = \sum_{j=1}^{L-2} [T_+(b_u, l_{j-1}) + T_C(b_u, l_{j-1}) + c_m\, b_u\, l_{j-1} + c_s\, l_{j-1}]$$

propagarDatosACapa Un producto, una suma y una sigmoidea de matrices. En el costo T_p se cuentan las operaciones para todas las capas.

1. $X^j = \text{sigmoide}(B^j + W^j \cdot X^{j-1})$

$$T_p = \sum_{j=1}^{L-2} [T_\times(l_j, l_{j-1}, b_s) + T_+(l_j, b_s) + T_\sigma(l_j, b_s)]$$

Entonces, multiplicando por la cantidad de épocas y lotes, se obtiene:

$$\begin{aligned}
T_u &= e\,\frac{b_s}{b_u}\,(T_r + T_{er}) + T_p \\
&= b_s\,(5\,e + 1)(c_m + c_s) \sum_{j=1}^{L-2} l_j\, l_{j-1} \;+\; e\,\frac{b_s}{b_u}\,(3\,c_m + 2\,c_s + c_d) \sum_{j=1}^{L-2} l_j\, l_{j-1} \\
&\quad +\; b_s\,(e\,(4\,c_s + 2\,c_e + 2\,c_d + c_r + c_c) + (c_e + c_s + c_d)) \sum_{j=1}^{L-2} l_j \\
&\quad +\; e\,b_s\,(5\,c_s + c_m + c_e + c_d) \sum_{j=1}^{L-2} l_{j-1} \;+\; e\,\frac{b_s}{b_u}\,(2\,c_m + c_s) \sum_{j=1}^{L-2} (l_j + l_{j-1})
\end{aligned}$$

supervisedTraining

En esta etapa, al usar la biblioteca *alglib* [2], son necesarias conversiones (*aplanar* y *desaplanar*) entre las estructuras de datos que usan la función implementada para el cálculo del gradiente ($function\_grad$) y el resto del método de gradiente conjugado de la biblioteca (esto último indicado como T_{gc}^{lib}); todo esto está contemplado más las n_f llamadas a la función que calcula el gradiente.

aplanar y desaplanar Se copian los pesos y biases de formas matriciales a un arreglo lineal y viceversa. El costo de ambos es el mismo, por lo que se usará T_{ap} para indicar a los dos.

$$T_{ap} = T_{desap} = c_p \sum_{j=1}^{L-1} l_j\, l_{j-1} + c_p \sum_{j=1}^{L-1} l_j$$

gc_optimizar Este es el método de la biblioteca que además de sus cálculos (T_{gc}^{lib}), hace n_f llamadas a $function\_grad$.

$$T_{gc} = T_{gc}^{lib} + n_f \, T_f$$

function_grad Desaplana al inicio y vuelve a aplanar al final, en el medio calcula el error logarítmico y el gradiente.

$$T_f = 2\,T_{ap} + T_{ler} + T_{grad}$$

log error Esta operación requiere propagar los datos por la red (líneas 1-3) y realizar sumas, productos elemento-a-elemento ($\cdot^*$) y logaritmos. La j-ésima columna de la matriz con activaciones finales X^{L-1} (que corresponde a una muestra) tiene k elementos, del mismo modo la j-ésima columna de Y tendrá k elementos todos en 0 excepto el i-ésimo que estará en 1 si la j-ésima muestra es de clase i.

1. **for** $j \leftarrow 1, \ldots, (L-2)$
2. $\quad X^j = \text{sigmoide}(B^j + W^j \cdot X^{j-1})$
3. $X^{L-1} = \text{softmax}(B^{L-1} + W^{L-1} \cdot X^{L-2})$
4. $lEr = -\,\text{rowsum}(\text{colsum}(Y \cdot^* \log(X^{L-1})))$

$$\begin{aligned} T_{ler} = \sum_{j=1}^{L-1} [T_\times(l_j, l_{j-1}, b_s) + T_+(l_j, b_s)] \; + \; \sum_{j=1}^{L-2} T_\sigma(l_j, b_s) \; + \; T_S(l_{L-1}, b_s) \\ + \; (c_m + c_l)\, k\, b_s \; + \; T_C(k, b_s) \; + \; T_R(1, b_s) \end{aligned}$$

gradiente El gradiente se calcula de la misma manera que en el algoritmo Backpropagation. Esto es, calcula los gradientes (∂) de los pesos de entrada de cada capa utilizando los gradientes calculados previamente de la capa superior.

1. $\delta^{L-1} = X^{L-1} - Y$
2. **for** $j \leftarrow (L-2), \ldots, 1$
3. $\quad \delta^j = X^j \cdot^* (1 - X^j) \cdot^* ((W^{j+1})' \cdot \delta^{j+1})$
4. **for** $j \leftarrow 1, \ldots, (L-1)$
5. $\quad \partial_W^j = \delta^j \cdot (X^{j-1})'$
6. $\quad \partial_b^j = \text{rowsum}(\delta^j)$
7. **end**

$$\begin{aligned} T_{grad} = T_+(k, b_s) \; + \; \sum_{j=1}^{L-2} [T_\times(b_s, l_{j+1}, l_j) + T_+(l_j, b_s) + 2\, c_m\, l_j\, b_s] \\ + \; \sum_{j=1}^{L-1} [T_\times(l_j, b_s, l_{j-1}) + T_R(l_j, b_s)] \end{aligned}$$

Sumando estas partes, el costo para el entrenamiento supervisado queda así:

$$\begin{aligned} T_s &= 2\,T_{ap} + T_{gc}^{lib} + n_f\,T_f \\ &= T_{gc}^{lib} \;+\; (2\,c_p + 2\,n_f\,c_p + 2\,n_f\,b_s\,(c_m + c_s) - n_f\,c_s) \sum_{j=1}^{L-1} l_j\,l_{j-1} \\ &\quad +\; (2\,c_p + 2\,n_f\,c_p + n_f\,b_s\,(c_e + c_d + 2\,c_s) - n_f\,c_s) \sum_{j=1}^{L-1} l_j \\ &\quad +\; n_f\,b_s\,(c_m + c_s) \sum_{j=1}^{L-2} l_j\,l_{j+1} \;+\; 2\,n_f\,b_s\,c_m \sum_{j=1}^{L-2} l_j \\ &\quad +\; n_f\,k\,b_s\,(c_l + c_m + 2\,c_s) \;-\; n_f\,b_s\,c_s \;-\; n_f\,c_s \end{aligned}$$

test

Cada uno de los *test*s que aparecen sobre el final de la Figura 12 incluye la propagación completa de los datos, es decir, hasta la capa de salida con activación softmax, y tiene el siguiente costo (T_t):

1. **for** $j \leftarrow 1, \ldots, (L-2)$
2. $\quad X^j = \text{sigmoide}(B^j + W^j \cdot X^{j-1})$
3. $X^{L-1} = \text{softmax}(B^{L-1} + W^{L-1} \cdot X^{L-2})$

$$\begin{aligned} T_t &= \sum_{j=1}^{L-1} [T_\times(l_j, l_{j-1}, b_s) + T_+(l_j, b_s)] \;+\; \sum_{j=1}^{L-2} T_\sigma(l_j, b_s) \;+\; T_S(l_{L-1}, b_s) \\ &= b_s\,(c_m + c_s) \sum_{j=1}^{L-1} l_j\,l_{j-1} \;+\; b_s\,(c_e + c_s + c_d) \sum_{j=1}^{L-1} l_j \;-\; b_s\,c_s \end{aligned}$$

Total Sumando los costos de todas las etapas multiplicándolas por la cantidad de veces que se repite cada una, se tiene que:

$$T_{total} = \frac{d}{b_s}\,T_u \;+\; \frac{d}{b_s}\,T_s \;+\; \frac{d}{b_s}\,(s+1)\,T_t$$

Tomando el término de mayor complejidad en cada etapa y siendo l_s una cota superior del número de unidades en cada capa, se tiene que el orden aproximado de complejidad temporal al ejecutar una vez cada etapa es entonces:

$$O(T_u) = O(T_s) = O(T_t) = O(L.l_s^2)$$

es decir, lineal en el número de capas y cuadrático en el tamaño de cada capa. Cabe destacar que el costo final también dependerá linealmente del tamaño del dataset d y el número de épocas e.

4. Pruebas

Para llevar a cabo las pruebas, se diseñaron dos grupos de datasets no estacionarios a partir de MNIST [29] y NORB [35], ampliamente usados y conocidos. Desde ahora los llamaremos *ne*-MNIST y *ne*-NORB para abreviar.

MNIST La base de datos MNIST de dígitos escritos a mano, consta de 60000 imágenes de entrenamiento y 10000 de test. El tamaño de los dígitos fue normalizado y se centraron en imágenes con una resolución fija de 28×28 píxeles.

Las imágenes originales eran en blanco y negro. En MNIST, los dígitos fueron escalados –manteniendo su relación de aspecto– y ajustados a un cuadrado de 20 píxeles de lado. Como el algoritmo de normalización usó la técnica de *anti-aliasing*, las imágenes resultantes contienen píxeles en escala de grises. Finalmente, se centraron en un cuadro de 28×28 según el centro de masa de los píxeles. En la Figura 13 se muestran algunos dígitos de este dataset.

Figura 13: Ejemplos de dígitos de MNIST. Los de una misma columna pertenecen a la misma clase.

NORB Esta base de datos se presenta en dos versiones: una más completa con objetos en diferentes tamaños, fondos uniformes, con texturas o con otros objetos y perturbaciones aleatorias, y otra reducida a imágenes con fondo uniforme y tamaño de objetos normalizado, esta se denomina *smallNORB* y fue la que se utilizó en este trabajo. En adelante, se usará NORB también para referirse a esta última versión.

La base de datos smallNORB contiene imágenes en escala de grises de 50 juguetes agrupados en 5 categorías: animales cuadrúpedos, figuras humanas, aviones, camiones y autos. Hay 10 figuras distintas por cada categoría. Las imágenes fueron tomadas por dos cámaras (en disposición estéreo) bajo 6 iluminaciones, 9 elevaciones (30° a 70° cada 5°) y 18 azimut (de 0° a 340° cada 20°) distintos. El fondo de las imágenes es liso, sin textura.

El conjunto de entrenamiento está compuesto por 5 figuras de cada categoría (4, 6, 7, 8 y 9) y el resto de las figuras (0, 1, 2, 3 y 5) corresponden al conjunto de test. Cada dataset contiene 24300 pares de imágenes de 96×96 píxeles cada una. En la Figura 14 se muestran algunas imágenes de ejemplo.

Para las pruebas principales se usó el *Streaming Ensemble Algorithm* (SEA) con dos tipos de clasificadores: *Support Vector Machines* (SVM) y *Stacked Restricted Boltzmann Machines* (SRBM).

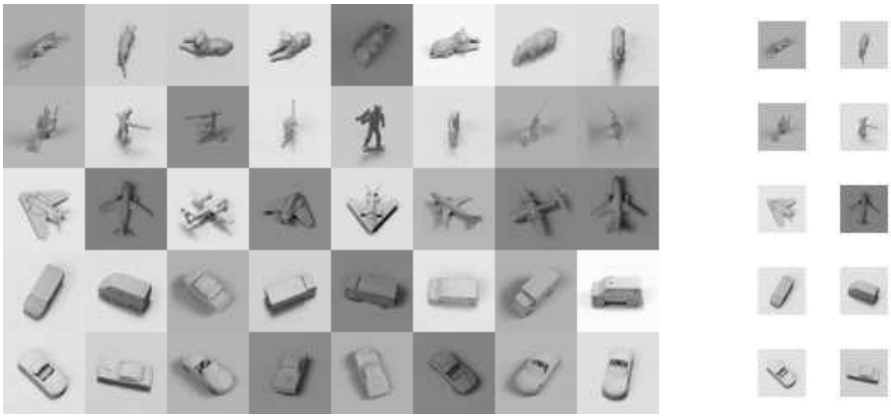

Figura 14: Ejemplos de imágenes de NORB a 48×48 píxeles. Las dos últimas columnas muestran las imágenes reescaladas al tamaño usado en las pruebas, 28×28. Los ejemplos de una misma fila pertenecen a la misma categoría.

Debido a que una ejecución del algoritmo SEA & SRBM requiere muchas horas para finalizar, las pruebas más exhaustivas se hicieron sólo con *ne*-MNIST, realizándose 30 ejecuciones con cada dataset. Las que corresponden a *ne*-NORB fueron más bien secundarias, con 10 ejecuciones por dataset. Los meta-parámetros de los algoritmos involucrados fueron optimizados mayormente con *ne*-MNIST y en menor medida con el restante.

En lo que respecta a hardware, se contó con un cluster con 40 CPUs de cuatro núcleos con 16 GB de RAM.

4.1. Preliminares

SRBM Se realizaron algunas pruebas sobre datasets estacionarios para comprobar rendimientos y fijar algunos meta-parámetros de las redes.

El mejor resultado con una SRBM se dio entrenando sobre el dataset estacionario MNIST de 60000 ejemplos, y testeando con el dataset de 10000. Para entrenar, los dígitos fueron ordenados de modo que haya uno de cada clase (0 a 9) cada 10 ejemplos. El error fue de 1.16 % y los meta-parámetros se muestran en la Tabla 1.

Arquitectura Indica el número de unidades en cada capa, siendo el de más a la izquierda la capa de entrada, el de más a la derecha, la de salida y el orden determina cómo se conectan las unidades.

Learning rate (RBM) Factor de aprendizaje en la etapa no supervisada, para el descenso por el gradiente estocástico en *Contrastive Divergence* usado en *rbmUpdate* para actualizar pesos y biases.

Momentum (RBM) Este método sirve para acelerar el aprendizaje cuando el gradiente no varía demasiado entre actualizaciones. En las épocas iniciales, se usa el momentum inicial, luego el final. Se aplica sumando un factor de la actualización anterior.

Weight cost (RBM) Coeficiente del término extra agregado al gradiente por el método Weight-decay. Este término es la derivada de una función que penaliza

pesos grandes, siendo la más simple (L2), la mitad de la suma de los pesos al cuadrado por este coeficiente.

Épocas (RBM) Cantidad de épocas para el entrenamiento no supervisado. Las épocas iniciales están asociadas al momentum (ver más arriba).

Tamaño de mini-batch (RBM) Los datos presentados a una RBM para entrenar, son divididos en pequeños lotes. Luego de procesar cada mini-batch se realiza una actualización de los pesos y biases.

Épsilon G/F/X/Iter. máx. (GC) Determinan cuándo debe finalizar el algoritmo GC basándose en la norma del gradiente, el cambio de la función en iteraciones consecutivas, la norma del paso o la cantidad de iteraciones. Al dejarlos en 0, el método de parada es automático (según la implementación de Alglib) y selecciona un épsilon X pequeño.

StepMax (GC) Tamaño máximo del paso, si es 0, no tiene límite.

Tamaño de mini-batch (GC) Tiene el mismo significado que para las RBM. Para las pruebas el mejor resultado se dio igualando este tamaño al de los datos de entrenamiento (por eso se indicó *completo* en la tabla).

Etapa	Meta-parámetro	Valor
Red	Arquitectura	$784\times1000\times700\times700\times$ $\times700\times700\times2000\times10$
No-sup. (RBM)	Learning rate*	0,1
	Momentum inicial*	0,5
	Momentum final*	0,9
	Weight cost*	0,0002
	Épocas (por capa)	10
	Épocas para mom. inicial*	5
	Tamaño de mini-batch	100
Superv. (GC)	Épsilon G/F/X/StepMax*	0
	Iteraciones máxima	350
	Tamaño de mini-batch*	completo (60000)

Tabla 1: Meta-parámetros de la SRBM con mejor rendimiento obtenido en dataset estacionario.

De estos meta-parámetros, los marcados con (*) se usaron en las pruebas tal como están (basándose en trabajos anteriores con este mismo dataset [15, 18]) y los otros fueron optimizados. En la etapa supervisada de toda la red, se usó el algoritmo de gradiente conjugado de la biblioteca *alglib* [2], se dejaron todos los parámetros en 0 (excepto el número máximo de iteraciones, que fue limitado) para que el algoritmo seleccione el criterio de parada automáticamente.

También se realizaron pruebas con SEA en datasets Abruptos de *ne*-MNIST (ver sección siguiente 4.2) usando tres tipos de clasificadores –SRBM, SVM y C4.5– para comparar y corroborar sus rendimientos siendo los dos primeros los mejores (del orden del 98 % y 96 % contra 91 %), como era de esperar. Luego, se continuó sólo con SRBM y SVM en las pruebas principales.

Con DWM y GF se hicieron algunas pruebas usando SVM y NB, pero como no iban a usarse con SRBM por las razones mencionadas en la sección 3, no fueron profundizadas.

4.2. *ne*-MNIST

Con la base de datos de entrenamiento de MNIST, se diseñaron seis tipos de datasets no estacionarios: *Abrupto*, *AbruptoRepite*, *Gradual*, *Abrupto2*, *AbruptoRepite2* y *Gradual2*, explicados a continuación.

4.2.1. Datasets no estacionarios generados

Para transformar el dataset en *no estacionario* se tuvieron en cuenta los tipos de cambio abrupto y gradual, así como también los recurrentes. Aquí lo que cambiará con el paso del tiempo es la distribución de la cual se muestrean los datos. Además, se establecieron dos formas de ordenar los dígitos, y se pasó de un problema de clasificación multiclase (0 a 9) a uno de clasificación binaria (0 ó 1). Para esto último, se usó el siguiente criterio: los dígitos pares (0, 2, 4, 6 y 8) tendrán la clase 0 y los impares (1, 3, 5, 7 y 9), la clase 1.

En la Figura 15 pueden verse los datasets *ne*-MNIST con el primer orden de dígitos. Para el segundo orden, las gráficas son similares con la excepción de que las distribuciones son: $D_0 : \{0, 1, 8\}$, $D_1 : \{3, 4, 6, 7\}$ y $D_2 : \{2, 5, 9\}$. Este segundo orden fue organizado según la similitud y errores frecuentes de clasificación entre dígitos. Se buscó ubicar pares de dígitos similares ($\sim$) divididos en distribuciones consecutivas. Por ejemplo, $0 \sim 6$, $8 \sim 3$, $1 \sim 7$, $7 \sim 2$, $3 \sim 5$, $4 \sim 9$ y $6 \sim 5$. El fin de esto es evaluar si las redes pueden aprovechar características similares aprendidas de una distribución anterior en la actual.

Un dato tenido en cuenta fue que la cantidad máxima de imágenes común a todos los dígitos es de 5421 (en el conjunto de entrenamiento de MNIST de 60000 imágenes). Además, en cada tramo $[t_i, t_{i+1}]$ los dígitos de la distribución con probabilidad 1, fueron mezclados entre sí. Se generaron 30 datasets distintos para cada uno de los 6 tipos de datasets.

Abrupto Este dataset presenta dos cambios abruptos de la distribución en los instantes t_1 y t_2, como muestra la Figura 15. Se usaron 5421 imágenes de cada dígito, por lo que $\#D_0 = \#D_1 = 3 \times 5421 = 16263$ y $\#D_2 = 4 \times 5421 = 21684$, haciendo el tamaño total del dataset igual a 54210.

AbruptoRepite Los cambios en los instantes t_1, t_2, t_3, t_4 y t_5 también son abruptos y en los últimos tres además son recurrentes. La cantidad de datos en $[t_3, t_4]$, $[t_4, t_5]$ y $[t_5, t_6]$ es de 1000 por dígito, es decir, 3000, 3000 y 4000 en cada período,

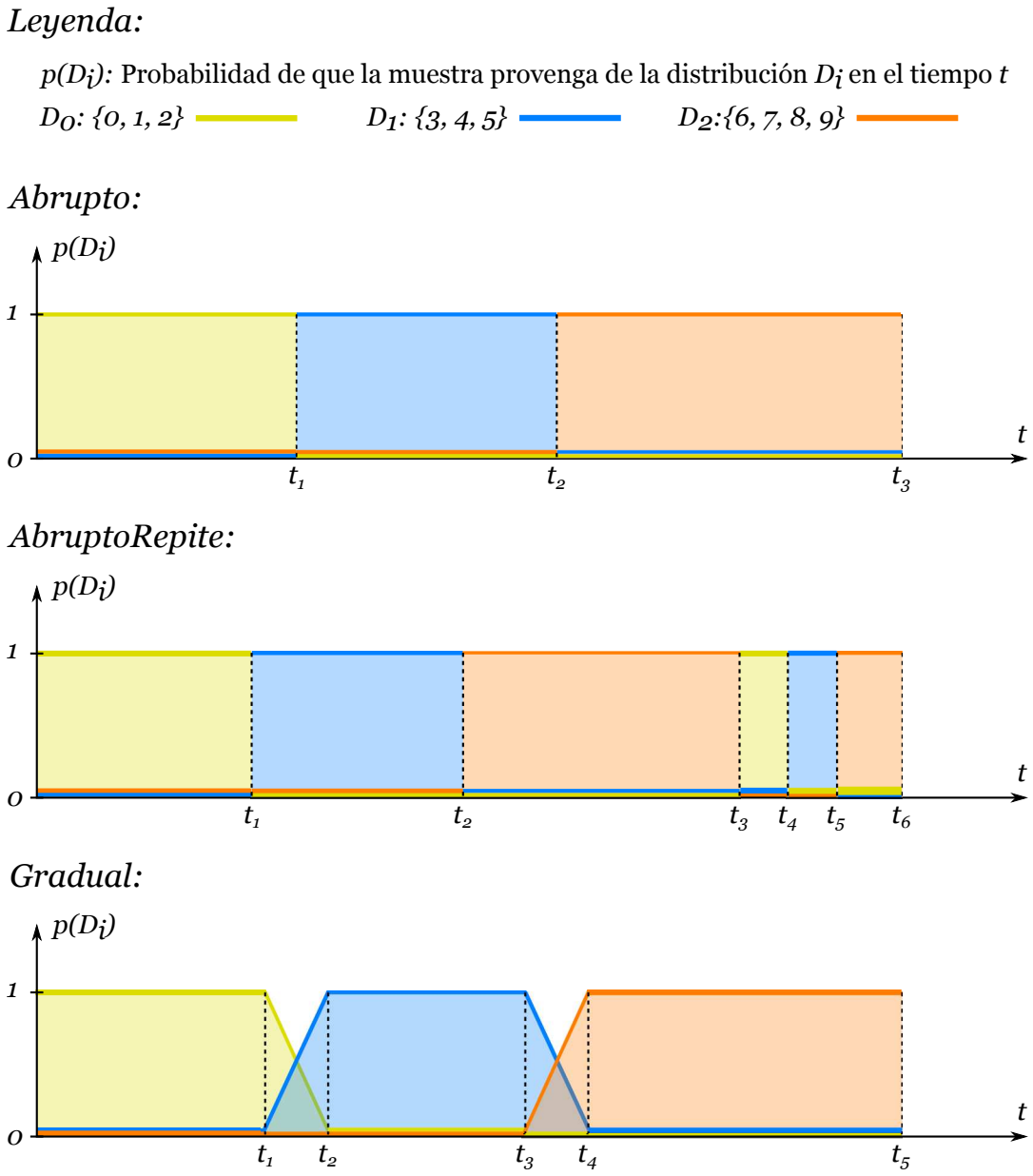

Figura 15: Datasets *ne*-MNIST: Abrupto, AbruptoRepite y Gradual, para el primer orden.

respectivamente. En los tres tramos iniciales hay 4421 por dígito. El total es nuevamente de 54210 imágenes.

Gradual En este dataset se dan dos cambios graduales en los períodos $[t_1, t_2]$ y $[t_3, t_4]$. En el primero hay 1200 imágenes y en el segundo 1400. En cada cambio gradual entre D_i y D_{i+1}, la probabilidad de que se muestree un dato de D_i es $p(D_i) = q$, con q decreciendo linealmente de 1 a 0 en 1200 instantes (o 1400 según corresponda), y $p(D_{i+1}) = 1 - q$. En los tramos en que alguna distribución tiene probabilidad 1 ($[t_0, t_1]$, $[t_2, t_3]$ y $[t_4, t_5]$) se volvió a buscar el máximo común de imágenes por dígito ya que en primer lugar se armaron los tramos con cambios, y como allí el muestreo se hace estocásticamente, no se puede conocer a priori ese

máximo. Por lo tanto, el tamaño de los 30 datasets graduales será variable.

Abrupto2, AbruptoRepite2, Gradual2 Similar a *Abrupto*, *AbruptoRepite* y *Gradual* pero ahora las distribuciones son: $D_0 : \{0, 1, 8\}$, $D_1 : \{3, 4, 6, 7\}$ y $D_2 : \{2, 5, 9\}$.

4.2.2. Optimización de meta-parámetros

La optimización de los meta-parámetros de la red SRBM y de SEA se realizó sobre uno de los 30 datasets del tipo *Abrupto*. En la Tabla 2 se muestran los valores finales de este proceso de optimización que luego se usaron para completar las restantes pruebas (con 29 datasets *Abrupto*'s y los 30 de cada uno de los otros 5).

	Meta-parámetro	Valor
SEA	Tamaño de lote: d (ver fig. 9)	40
	Tamaño máximo de ensamble: EMS	5
SRBM	Arquitectura	$784 \times 1000 \times 500 \times 500 \times 2000 \times 2$
	Épocas (por capa)	10
	Tamaño de mini-batch (no-sup.)	5
SVM	Costo	6
	Gamma	0,013

Tabla 2: Meta-parámetros optimizados finales con *ne*-MNIST. Los correspondientes a SEA fueron iguales con ambos clasificadores.

SEA & SRBM A la hora de encontrar los mejores meta-parámetros en SEA & SRBM, existen dificultades propias del uso de redes. La más significativa es la elección de una buena arquitectura de red, mientras más capas y más unidades por capa, mucho más tardará el entrenamiento, como ya vimos en el análisis de complejidad. El tiempo que tardó cada ejecución fue bastante alto (desde 20 minutos a 14 horas) en comparación con SVM, dependiendo de la arquitectura principalmente y de la carga de la máquina usada. Una prueba con los mejores meta-parámetros rondó las 12 horas.

Por otro lado, no es sencillo encontrar la mejor arquitectura, hay que tener en cuenta la profundidad, las unidades en cada capa, la cantidad de datos de entrenamiento, no provocar sobre-ajuste, etc., y ni siquiera es tan fácil con una receta [18]. Es por todo esto que la optimización no pudo realizarse de manera tan exhaustiva, aunque los resultados sean dentro de todo aceptables.

SEA & SVM La optimización usando SVM fue más completa y rápida por los cortos tiempos de ejecución (menor a 1 minuto) y por tener sólo dos meta-parámetros: `costo` y `gamma`. El valor de `gamma` corresponde al parámetro γ en la expresión de un kernel gaussiano RBF: $e^{-\gamma \|u-v\|^2}$.

Como pudieron realizarse más pruebas en esta optimización, podría decirse que el mejor rendimiento obtenido estuvo muy cerca de lo mejor que se pueda obtener

de SVM, por lo menos con un kernel RBF, lo que supondría una ventaja frente a SEA & SRBM.

4.3. *ne*-NORB

Con la base de datos de entrenamiento de smallNORB, se diseñaron tres tipos de datasets no estacionarios: *Abrupto*, *AbruptoRepite* y *Gradual*, explicados a continuación.

4.3.1. Datasets no estacionarios generados

De la misma manera que con MNIST, se generaron tres datasets según el tipo de cambio que presentan: *Abrupto*, *AbruptoRepite* y *Gradual*. Se utilizó sólo la primer imagen de cada par del conjunto de entrenamiento, contando un total de 24300 (4860 por categoría). Estas imágenes fueron escaladas a 28×28 píxeles para acotar los tiempos de ejecución de las pruebas y pensando que no se perderá tanta información.

En este caso, el concepto que cambiará con el correr del tiempo es el azimut y cada muestra deberá ser clasificada según la categoría a la que pertenezca (5 clases). De los 18 azimut distintos, se agruparon de a 3 consecutivos en 6 grupos, conformando así 6 distribuciones D_j, con $j = [0 \dots 5]$. Cada una de estas contiene un total de $5fig \times 9elev \times 6ilum \times 3az \times 5cat = 4050$ imágenes todas mezcladas.

Se generaron 10 datasets distintos para cada uno de los 3 tipos de datasets. Los esquemas de cada tipo pueden verse en la Figura 16.

Abrupto Este dataset presenta cinco cambios abruptos de la distribución en los instantes t_1, t_2, t_3, t_4 y t_5, como muestra la Figura 16. En cada tramo $[t_i, t_{i+1}]$ de la gráfica están todas las imágenes de la distribución D_i, mezcladas como ya fue mencionado. El tamaño del dataset es de 24300 imágenes.

AbruptoRepite Los cambios en los instantes t_i, con $i = [1 \dots 9]$, también son abruptos. Y además, en t_3, t_5, t_7 y t_9, son recurrentes. Cada tramo contiene la mitad de los ejemplos de la distribución que corresponda. En los tramos recurrentes, los ejemplos son los de la mitad restante. El tamaño del dataset es también de 24300 imágenes.

Gradual En este dataset se dan cinco cambios graduales en los períodos $[t_i, t_{i+1}]$, con $i = [1, 3, 5, 7, 9]$. En cada uno hay 1500 imágenes. La forma de muestrear los datos durante los cambios es la misma que se explicó con el dataset *Gradual* de *ne*-MNIST, con la única diferencia que q decrece en 1500 pasos. Por lo tanto, el tamaño de los 10 datasets graduales será variable.

4.3.2. Optimización de meta-parámetros

Con estos datasets se realizó una optimización poco profunda por el poco tiempo disponible y para evaluar la técnica en otro problema un poco más difícil como es NORB. Como el tamaño de las imágenes fue escalado a la misma dimensión que en

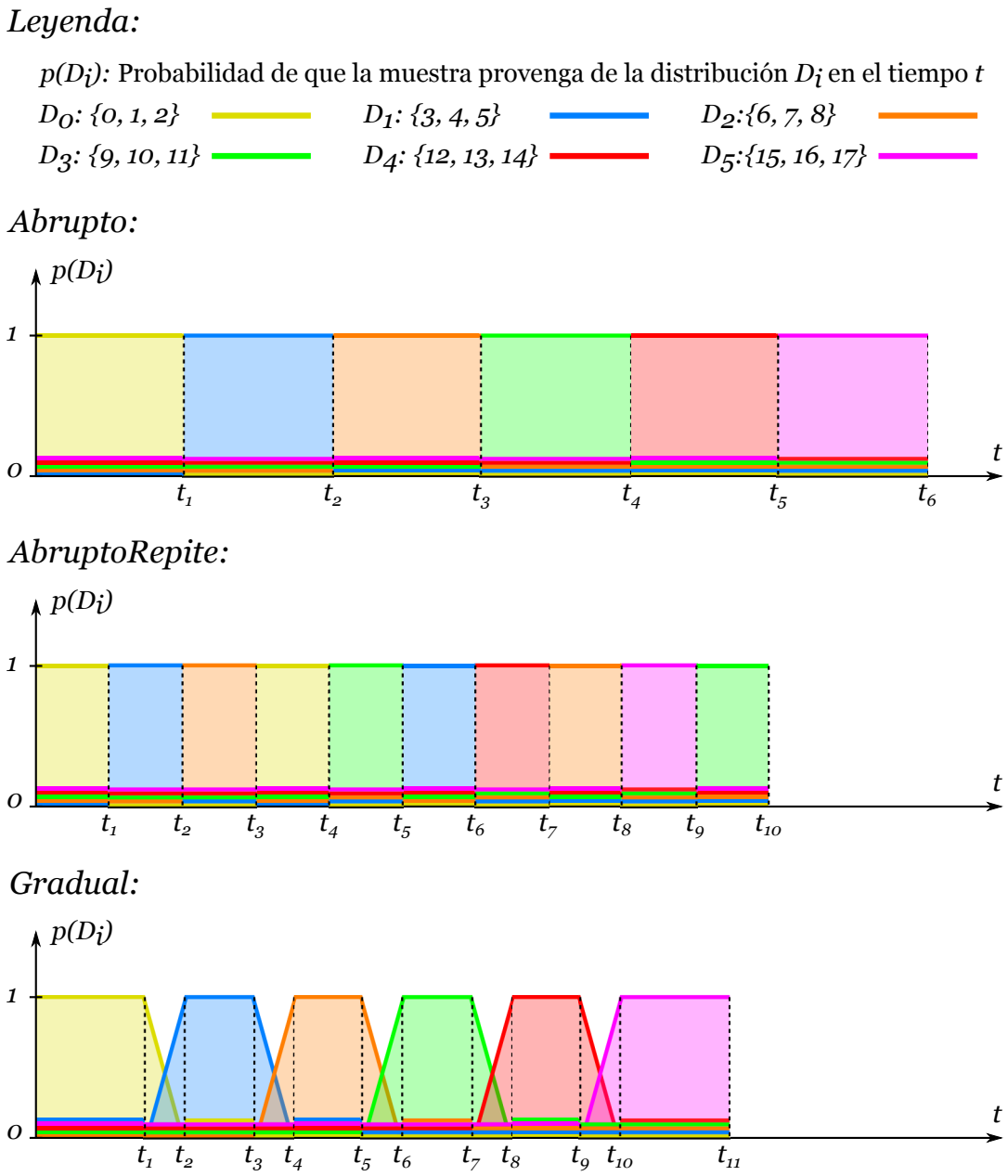

Figura 16: Datasets *ne*-NORB: Abrupto, AbruptoRepite y Gradual.

MNIST (28×28), no hubo mucha diferencia en los tiempos de ejecución salvo por la cantidad de ejemplos a procesar, que fueron menos en este caso.

La optimización se realizó sobre uno de los 10 datasets del tipo *Abrupto*. El tamaño de lote (d) y tamaño máximo de ensamble (EMS) no fueron optimizados en esta oportunidad, se mantuvieron iguales que en las pruebas con MNIST. En la Tabla 3 se muestran los valores finales que luego se usaron para completar las restantes pruebas (con 9 datasets *Abrupto*'s y los 10 de cada uno de los otros 2).

SEA & SRBM Como recién se mencionó, la optimización fue corta, no se probaron muchas arquitecturas de red y la más usada fue la misma que para el caso

	Meta-parámetro	Valor
SRBM	Arquitectura	$784 \times 1000 \times 500 \times 500 \times 2000 \times 5$
	Épocas (por capa)	10
	Tamaño de mini-batch (no-sup.)	8
	Iteraciones máxima en GC (sup.)	250
SVM	Costo	115
	Gamma	0,012

Tabla 3: Meta-parámetros optimizados finales con *ne*-NORB.

anterior. Sin embargo, un meta-parámetro que antes no había sido optimizado fue el número de iteraciones máxima requerido por el algoritmo de Gradiente Conjugado, con el que se mejoró unos puntos el rendimiento.

SEA & SVM Con SVM también se pudo realizar una optimización más completa del `costo` y `gamma`, los tiempos de ejecución fueron menores a 30 segundos.

5. Resultados para *ne*-MNIST

Las Figuras 17, 18, 19, 20 y 21 muestran los resultados de las pruebas realizadas sobre los datasets *ne*-MNIST.

Para simplificar la gráfica, cada punto de la misma corresponde al promedio de los valores de 10 lotes consecutivos (es decir, 400 ejemplos), excepto durante cambios y períodos de adaptación, en donde hay un punto por cada lote para una mejor apreciación.

En las líneas continuas, estos valores representan la media de los 30 porcentajes de acierto para un lote, correspondientes a las ejecuciones sobre las 30 variantes del dataset, y en las líneas de trazos, representan una medida de dispersión de los valores $\pm\sigma$, donde σ es su desviación estándar.

El **mejor rendimiento** tanto con SRBM como con SVM, se obtuvo en el dataset con el que sus meta-parámetros fueron optimizados: *Abrupto*. En todos los datasets siempre fue superior SEA & SRBM. Los resultados pueden verse en la Tabla 4.

Dataset	Acierto % ($\pm\sigma$)	
	SEA & SRBM	SEA & SVM
Abrupto	98,22 (0,05)	97,30 (0,06)
Abrupto2	98,22 (0,04)	96,92 (0,05)
AbruptoRepite	97,76 (0,05)	96,89 (0,07)
AbruptoRepite2	97,80 (0,06)	96,52 (0,05)
Gradual	98,16 (0,05)	97,18 (0,06)
Gradual2	98,16 (0,07)	96,84 (0,07)

Tabla 4: Resultados con *ne*-MNIST sobre las 30 ejecuciones. El porcentaje de acierto mostrado representa la media de las 30 ejecuciones y el valor entre paréntesis, la desviación estándar correspondiente a cada uno.

Respecto de la **capacidad de SEA en adaptarse** a los cambios abruptos ya sea con SRBM o SVM, es notable la fuerte caída que sufrió el sistema al producirse un cambio, incluso si fue recurrente. Sin embargo, con ambos clasificadores la recuperación dependió de la cantidad de miembros que tenía el ensamble, fijado en 5. Puede verse que luego de 5 lotes el sistema recuperó su performance, y a medida que se iban eliminando expertos desactualizados, los recién entrenados ayudaban a levantar el rendimiento hasta estabilizarlo.

Agregar más expertos al ensamble podría retrasar aún más la recuperación ya que puede verse cómo afectan la adaptación (ver párrafo “Detectar abruptos” en sección 8).

El **escalonado** entre las diferentes distribuciones y en ambos órdenes, puede estar relacionado con la composición particular de cada D_i y la cantidad de dígitos distintos (3 ó 4), siendo algunas más difíciles que otras. Y el hecho de que las distribuciones recurrentes volvieran a tener un rendimiento similar al de la primera

aparición, ratifica esta idea y desestima un problema de degradación temporal de la performance. Estos saltos se destacaron más con SVM, en donde siempre hubo un aumento del error tras pasar a una nueva distribución (no recurrente), mientras que el uso de SRBM mostró mayor robustez.

Usando SRBM, en las distribuciones de 4 dígitos (D_2 y D_1' –el apóstrofe indica que corresponde al segundo orden) la performance fue más baja y los intervalos de confianza un poco mayores que en las distribuciones de 3 dígitos. Evidentemente tener más dígitos para clasificar afectó el rendimiento. En general, los intervalos de confianza fueron más estrechos con SRBM.

Cuando se dieron los **cambios recurrentes**, el sistema volvió a perder rendimiento y a recuperarse durante los próximos 5 lotes. La caída luego de que una distribución apareció por primera vez fue similar a la caída después de la segunda aparición. Una vez que el sistema se recuperó tras la segunda aparición de una misma distribución, la performance se mantuvo en el mismo nivel que antes.

Los **cambios graduales** se dan durante 30/35 lotes dependiendo de si las distribuciones involucradas tienen 3 y 3 dígitos o 3 y 4. Durante estos cambios existió una caída del rendimiento menor y más lenta que en los cambios abruptos. Luego, el sistema fue ganando efectividad *lentamente* a medida que los ejemplos de la nueva distribución aparecían con mayor frecuencia.

Tanto en los períodos estables, para los seis datasets, como durante los cambios graduales, el rendimiento usando SRBM fue superior. Sin embargo, la adaptación a las nuevas distribuciones fue similar con ambos clasificadores.

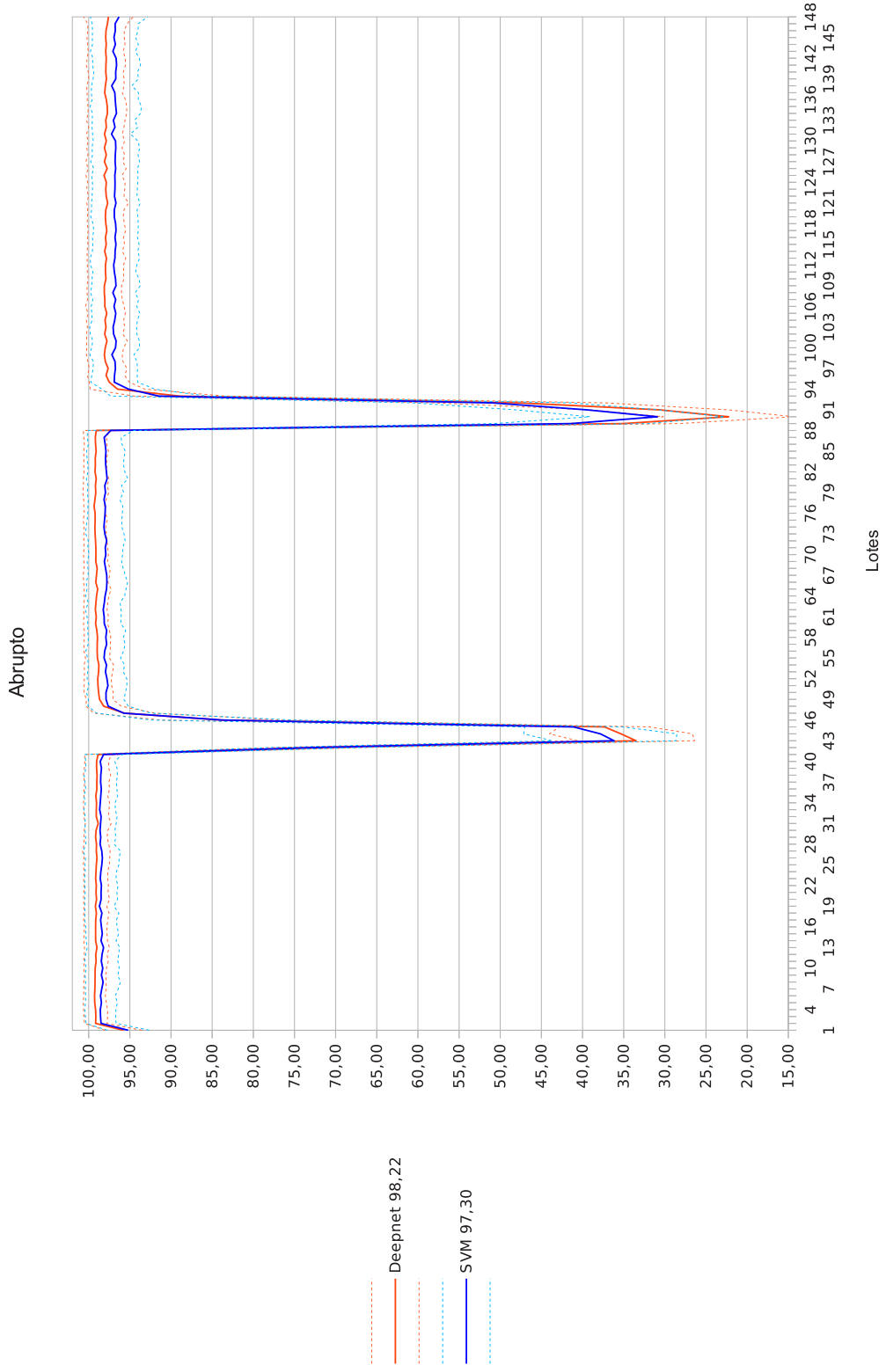

Figura 17: Resultado con dataset Abrupto de MNIST. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

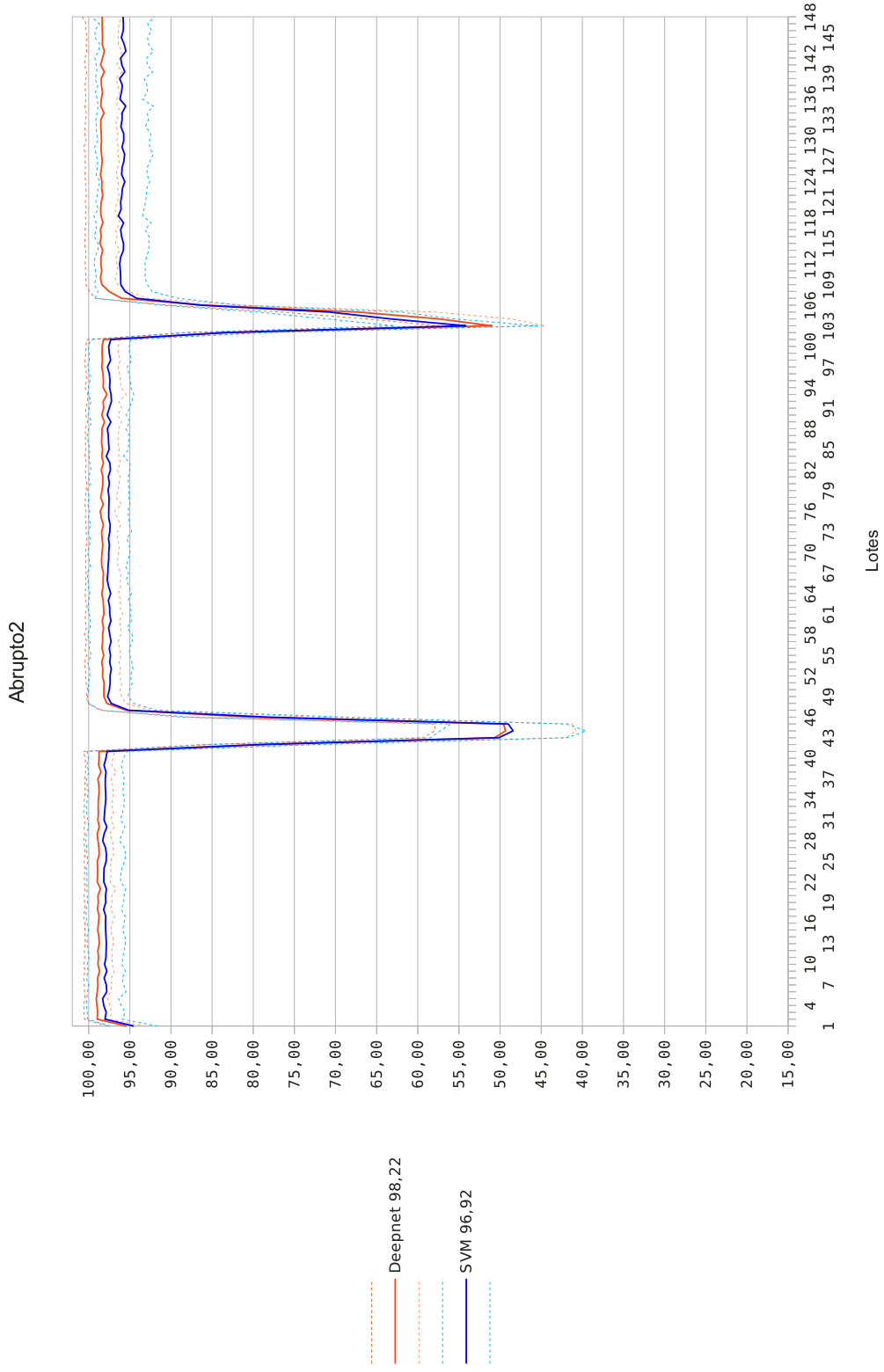

Figura 18: Resultado con dataset Abrupto2 de MNIST. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

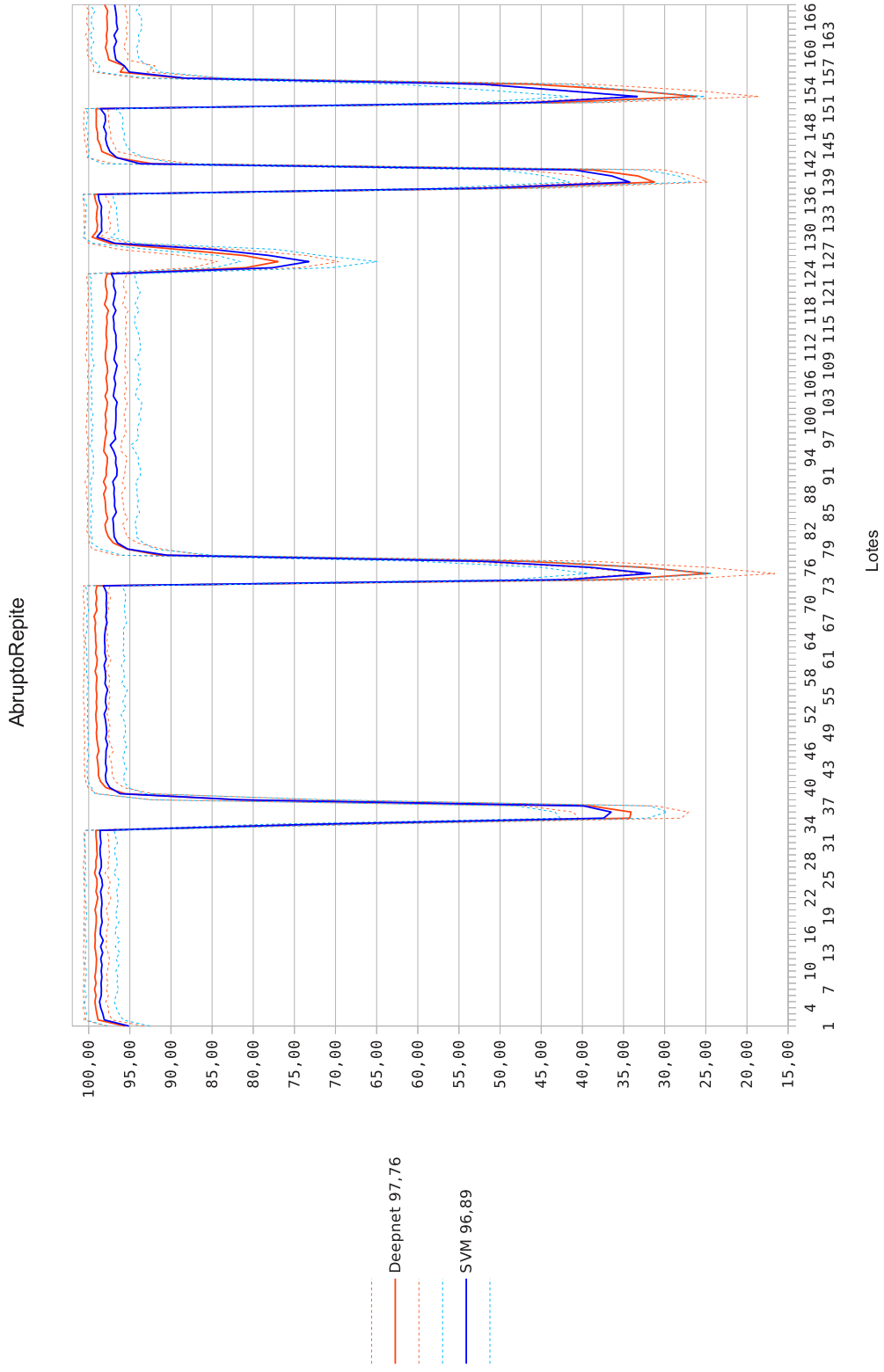

Figura 19: Resultado con dataset AbruptoRepite de MNIST. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

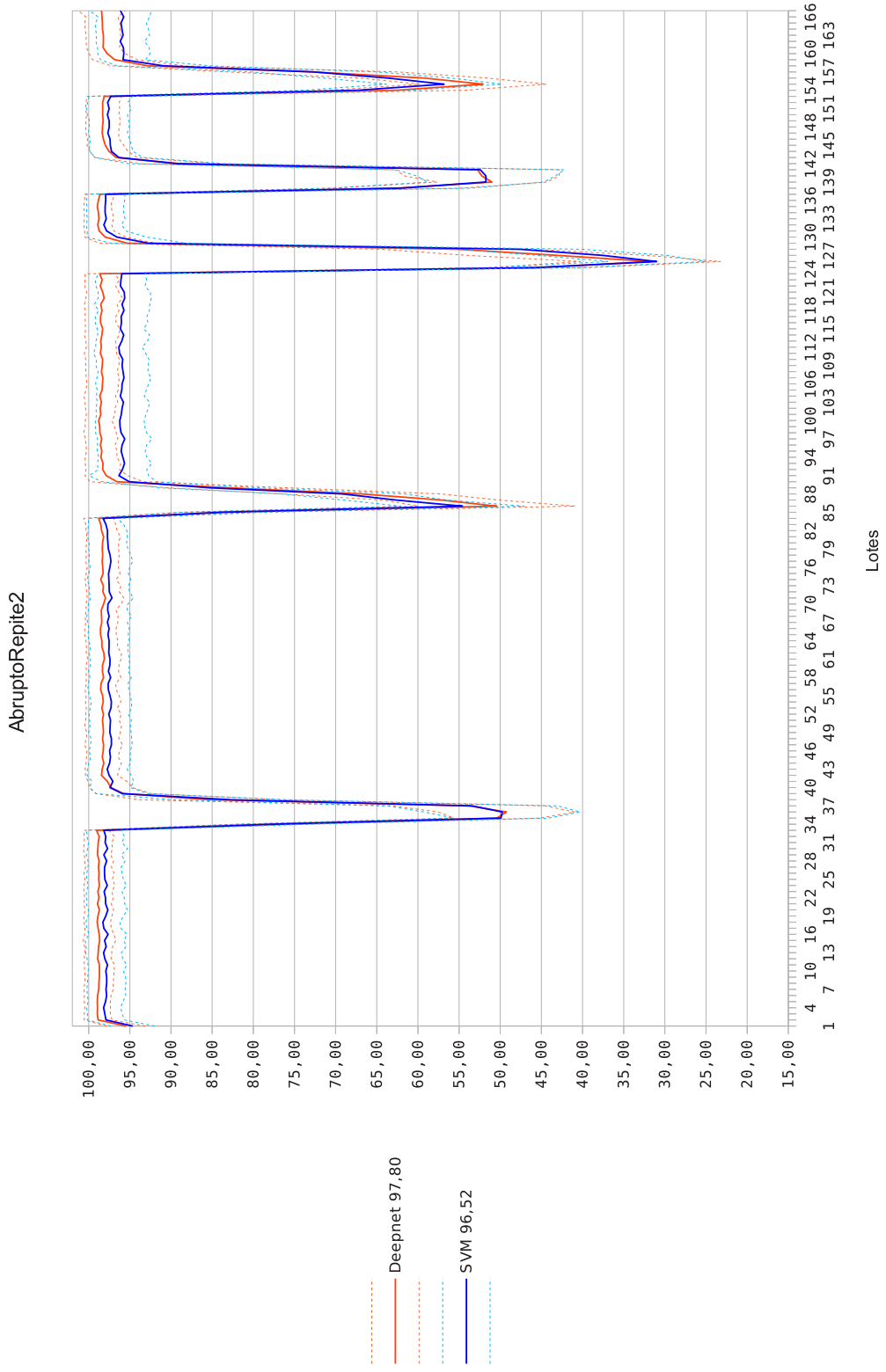

Figura 20: Resultado con dataset AbruptoRepite2 de MNIST. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

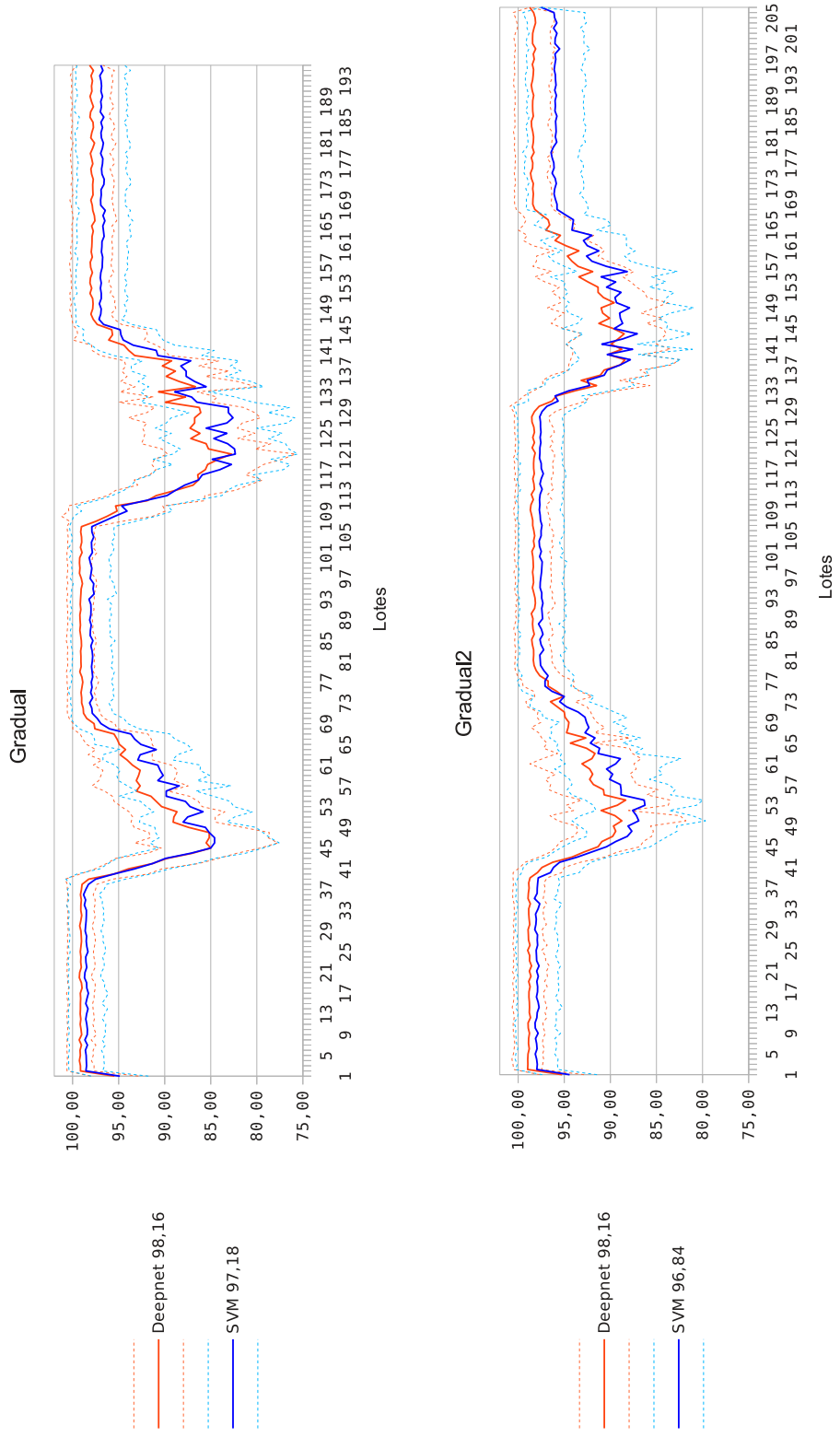

Figura 21: Resultados con datasets Gradual y Gradual2 de MNIST. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

6. Resultados para *ne*-NORB

Las Figuras 22, 23 y 24 muestran los resultados de las pruebas realizadas sobre los datasets *ne*-NORB.

Para simplificar las gráficas, en Abrupto y AbruptoRepite, cada punto de las mismas corresponden al promedio de los valores de 5 lotes consecutivos (es decir, 200 ejemplos), excepto durante cambios y períodos de adaptación, en donde hay un punto por cada lote para una mejor apreciación. En la gráfica Gradual, todos los puntos corresponden al promedio de 2 lotes consecutivos.

Las líneas continuas y de trazos tienen el mismo significado que con *ne*-MNIST pero con 10 variantes de los datasets en lugar de 30.

El **mejor rendimiento** tanto con SRBM como con SVM, se obtuvo en el dataset con el que sus meta-parámetros fueron optimizados: *Abrupto*. En todos los datasets siempre fue superior SEA & SVM. Los resultados pueden verse en la Tabla 5. Si no se tuvieran en cuenta los lotes con los que se obtuvieron bajos rendimientos (luego de cambios abruptos y durante cambios graduales), estos porcentajes ascenderían entre 1 y 3,5 puntos aproximadamente, para ambos clasificadores.

Dataset	Acierto % ($\pm\sigma$)	
	SEA & SRBM	SEA & SVM
Abrupto	72,6 (0,5)	79,1 (0,3)
AbruptoRepite	71,3 (0,5)	78,1 (0,3)
Gradual	70,1 (0,5)	76,5 (0,5)

Tabla 5: Resultados con *ne*-NORB sobre las 10 ejecuciones. El porcentaje de acierto mostrado es el promedio de las 10 ejecuciones y entre paréntesis se ve la desviación estándar correspondiente a cada uno.

Los resultados de las 10 variantes fueron mucho más dispersos que los obtenidos con *ne*-MNIST. Las desviaciones estándar en este caso, están un orden de magnitud por arriba de aquellas mostradas en la Tabla 4.

El hecho mencionado de no haber realizado una optimización completa de SEA & SRBM, se hace evidente en la diferencia de rendimiento de unos 6–7 puntos por debajo de SEA & SVM.

En comparación con *ne*-MNIST, este problema es evidentemente más difícil. Algunos factores pueden ser el mayor nivel de detalle en las imágenes y la pérdida de información útil al escalarlas de 96×96 a 28×28.

En general, los intervalos de confianza fueron más grandes que en *ne*-MNIST, y en este caso apenas mayor con SRBM.

Respecto de la **capacidad de SEA en adaptarse** a los cambios abruptos, se observó un comportamiento similar al obtenido con *ne*-MNIST: fuerte caída al producirse un cambio (nuevo o recurrente) seguida de una recuperación tras no menos de 5 lotes (tamaño máximo del ensamble).

Del mismo modo que con *ne*-MNIST, cuando se dieron los **cambios recurrentes**, el sistema volvió a perder rendimiento y a recuperarse durante los próximos 5 a 7 lotes. También fue similar la caída luego de la primera y segunda aparición de una distribución y los niveles de performance tras las recuperaciones se mantuvieron en el mismo nivel en ambas apariciones.

Sin embargo, cabe destacar una particularidad que se dio en los cambios alrededor de los lotes 57, 90 y 122, en donde la caída fue mucho menor que en los otros cambios. Las distribuciones implicadas en cada uno son $D_0 \rightarrow D_3$, $D_1 \rightarrow D_4$ y $D_2 \rightarrow D_5$ respectivamente. Los azimut en D_0 son 0, 20 y 40°, y en D_3, 180, 200 y 220°, es decir, los objetos son vistos del lado opuesto (diferencia de 180°), esto se repite con los otros dos pares de distribuciones. Entonces, que la caída haya sido *leve* puede deberse a que algunos objetos son lo suficientemente simétricos respecto de algún plano, y a que para ciertos pares de azimut y su opuesto, estos objetos reflejan dicha simetría. De este modo, dos imágenes compartirán más características similares facilitando la clasificación de los expertos que en ese momento estén en el ensamble.

Los **cambios graduales** se dan durante 38 lotes. Durante estos cambios la caída del rendimiento también fue menor que en los cambios abruptos. La performance más baja del sistema se dio aproximadamente cuando la cantidad de muestras de las dos distribuciones involucradas en cada cambio fue similar, luego de lo cual comenzó a recuperarse.

Tanto en los períodos estables, para los tres datasets, como durante los cambios graduales, el rendimiento usando SVM fue superior. Sin embargo, la adaptación a las nuevas distribuciones fue similar con ambos clasificadores.

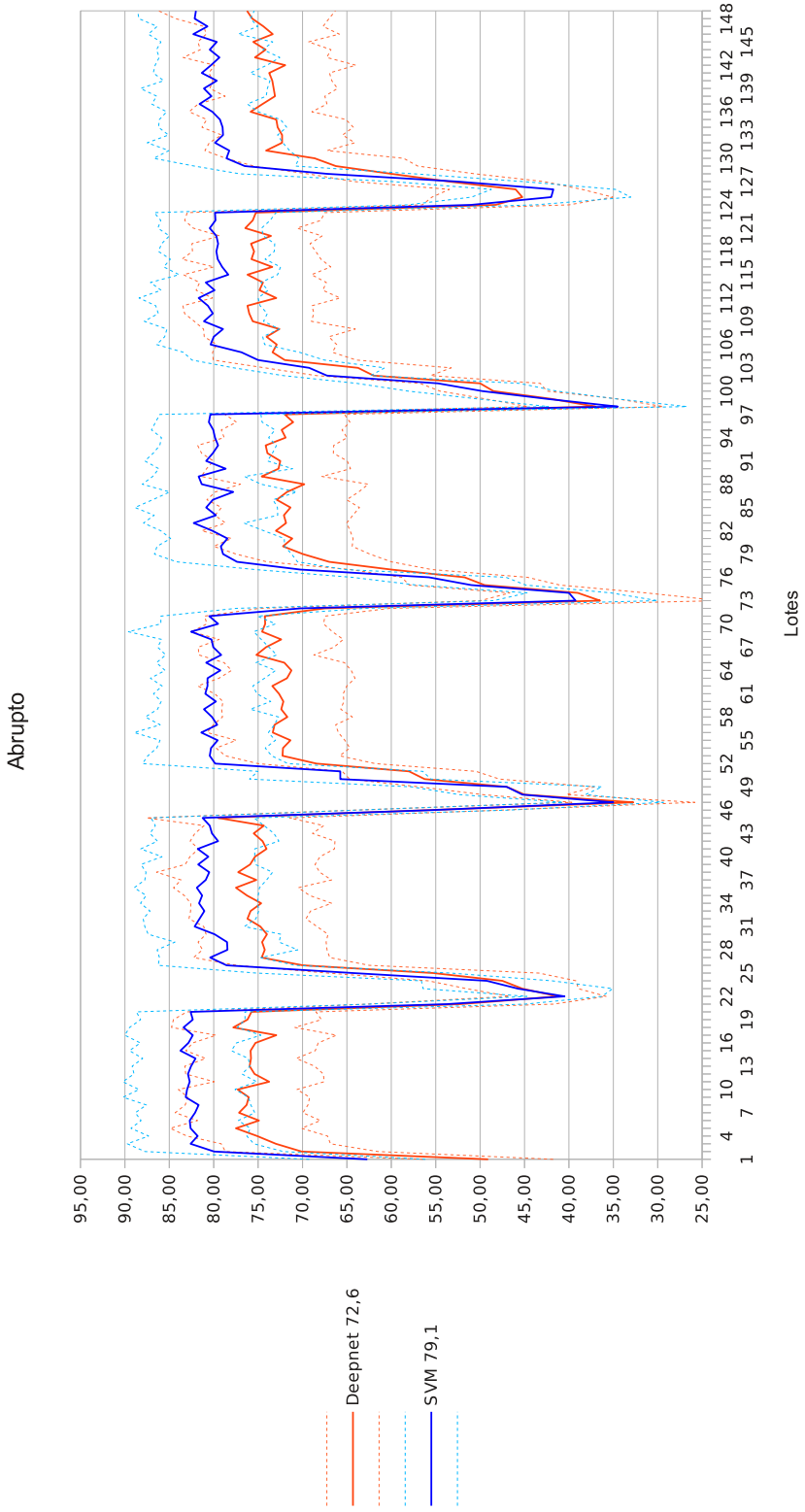

Figura 22: Resultado con dataset Abrupto de NORB. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

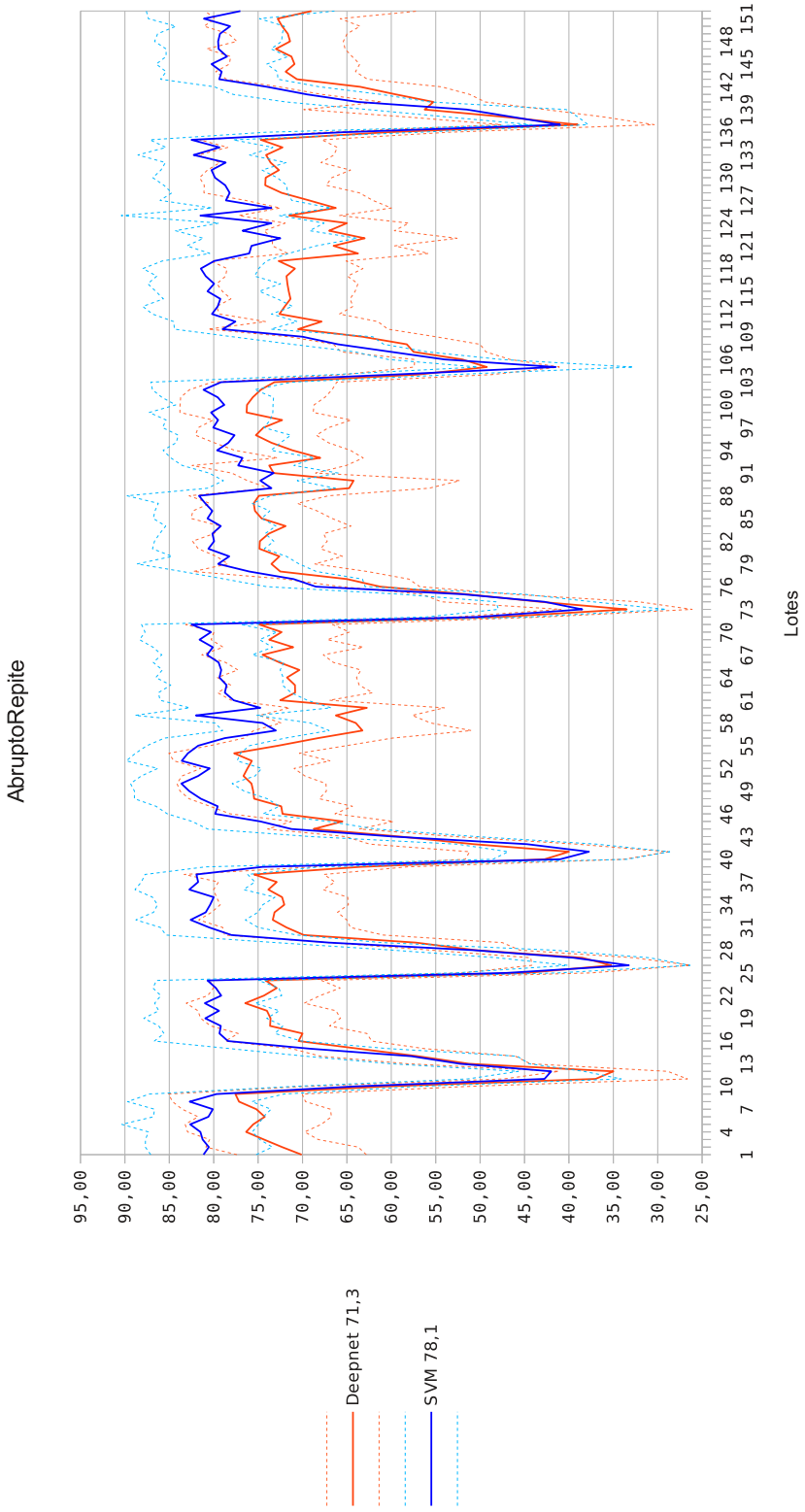

Figura 23: Resultado con dataset AbruptoRepite de NORB. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

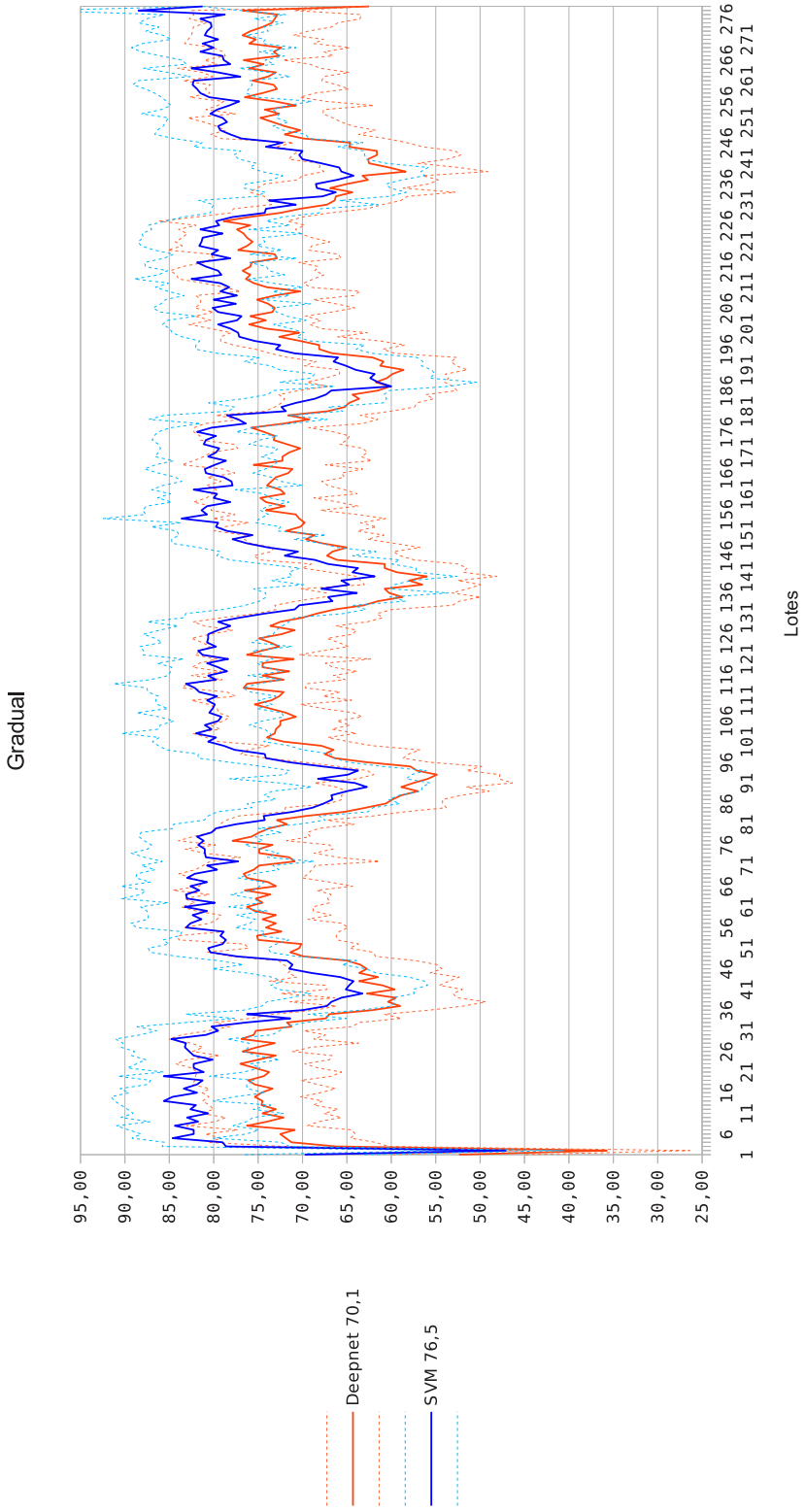

Figura 24: Resultado con dataset Gradual de NORB. Las líneas de trazos representan una medida de dispersión de los valores basada en sus desviaciones estándar.

7. Conclusiones

Para este trabajo se implementaron los algoritmos de clasificación Redes Profundas (SRBM), Naïve Bayes (NB), C4.5 y SVM[5], un algoritmo listo para probar SRBM y SVM, SEA (con SRBM, C4.5 y SVM), DWM (con NB y SVM) y GF (con NB y SVM).

7.1. *ne*-MNIST

En los datasets *ne*-MNIST pudo comprobarse que las SRBM aportan algo más de efectividad que las SVM aunque el tiempo requerido es muy alto. Si para un problema determinado el tiempo no es tan importante, vale la pena considerar el uso de redes profundas.

Las SVM dentro de un ensamble no tienen en cuenta datos de entrenamiento que no hayan sido usados para entrenarlas. En cambio, cada nueva SRBM tiene acumulada información de todos los lotes vistos hasta el momento, en la red base (mediante aprendizaje no supervisado). Con estos datasets, el hecho de contar con esta *historia*, al parecer no tuvo consecuencias negativas. Queda determinar si pudo haber agilizado la búsqueda de un buen mínimo en las etapas supervisadas. Ver párrafo "Acumular información" en sección 8. Es probable que este dataset de dígitos sea relativamente fácil y entonces no pueda verse el efecto de acumular información, ver párrafo "NORB" en sección 8.

Puede decirse que la política de inclusión de SRBM al SEA fue satisfactoria ya que el rendimiento fue superior al que se obtuvo con el uso de uno de los mejores métodos conocidos (SVM), y a la hora de adaptarse a los cambios, el comportamiento con ambos clasificadores fue similar.

7.2. *ne*-NORB

Según las pruebas realizadas con otros métodos sobre el dataset estacionario NORB[6], el error es significativamente menor que el que se obtuvo en este trabajo con SEA & SRBM y versiones no estacionarias del mismo. Queda buscar una mejor combinación de meta-parámetros para SRBM de modo de achicar el error obtenido, es de esperar que tenga un rendimiento cercano (e incluso superior) al de SVM para un mismo tamaño de imagen, ya sea el original o uno más chico como el usado en este trabajo.

7.3. General

El algoritmo SEA, con la actual configuración de meta-parámetros –en especial, el tamaño máximo de ensamble: EMS–, presentó caídas importantes frente a los cambios y recuperaciones rápidas debido a la elección de un EMS pequeño por el que, para estos datasets, sólo se tiene en cuenta una porción relativamente corta de los lotes anteriores. Esto hace difícil pensar que ante un cambio recurrente el sistema

[5] C4.5 y SVM vía bibliotecas.

[6] http://www.cs.nyu.edu/~yann/research/norb/

cuente todavía con algún experto en el ensamble con información antigua pero circunstancialmente útil, y el hecho de incluir estos datos antiguos en el entrenamiento de las SRBM tampoco mostró una mejoría notable en comparación con el rendimiento para los primeros lotes, ya que MNIST parece ser demasiado fácil como para determinarlo. De todos modos, el rendimiento general en los datasets *ne*-MNIST (estudiados más en detalle) fue superior con SRBM que el obtenido con el uso de uno de los mejores métodos conocidos (SVM), lo que es bueno teniendo en cuenta que el rendimiento para esta versión no estacionaria del problema fue equiparable al esperado en su versión estacionaria.

8. Trabajos futuros

En esta última sección se listan algunas cuestiones que quedaron pendientes, que iban surgiendo durante el trabajo o que escapaban del tema central.

Detectar abruptos Ver la posibilidad de detectar el comienzo de un cambio abrupto, y darle mayor importancia a los nuevos expertos para recuperarse más rápido. Esto podría realizarse pesando los miembros del ensamble de manera similar a como se hace en DWM.

Ordenando Ver cuánto y cómo afecta el orden de los dígitos. Se probaron dos, pero aún con tres distribuciones de dígitos, las combinaciones son muchas.

Acumular información Realizar pruebas con SEA & SRBM sin una red base, entrenando una red nueva con cada lote nuevo, para evaluar si realmente existe una mejora con la versión que usa una red base acumulando toda la información previa. Posiblemente sea necesario hacerlo con otros datos ya que MNIST parece ser demasiado fácil como para determinar esto.

Fine tuning Con el fin de acelerar el entrenamiento, ¿qué pasaría si el fine tuning sólo se hiciera a la última capa (en lugar de a toda la red) dejando intactas las capas ocultas luego del entrenamiento no supervisado? Y por otro lado, ¿si luego de esto (es decir, entrenar sólo la última capa, esperando que la red quede así mejor inicializada) se hiciera además el fine tuning a la red completa?

NORB Ampliar las pruebas con *ne*-NORB y otros datasets más difíciles que MNIST.

DWM con Deepnets Adaptar el algoritmo para usarlo con redes. Recordar el problema del tiempo de entrenamiento de todas las redes del ensamble para cada lote.

GF con Deepnets Buscar la manera de añadir una política de Gradual Forgetting a las redes profundas.

9. Bibliografía

[1] **R. Adams, H. Wallach, Z. Ghahramani.** *Learning the structure of deep sparse graphical models.* JMLR Workshop and Confference Proceedings 9: Proceedings of the 13^{th} International Workshop on Artificial Intelligence and Statistics, pp. 1-8, 2010.

[2] **S. Bochkanov and V. Bystritsky.** *ALGLIB.* http://www.alglib.net/

[3] **C. Sanderson.** *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical Report, NICTA, 2010.

[4] **P. Bartlett, S. Ben-David, S. Kulkarni.** *Learning changing concepts by exploiting the structure of change.* Machine Learning 41(2), pp. 153-174, 2000.

[5] **Y. Bengio, Y. LeCun.** *Scaling learning algorithms towards AI.* Large scale kernel machines, MIT Press, 2007.

[6] **Y. Bengio.** *Learning Deep Architectures for AI.* Journal of Foundations and Trends in Machine Learning 2(1), pp. 1-127, 2009.

[7] **F. Camci, R.B. Chinnam.** *General support vector representation machine for one-class classification on non-stationary classes.* Pattern Recognition 41, pp. 3021-3034, 2008.

[8] **F. Desobry, M Davy, C. Doncarli.** *An online kernel change detection algorithm.* IEEE Transactions on Signal Processing 53(8), pp. 2961-2974, 2005.

[9] **R.O. Duda, P.E. Hart, D.G. Stork.** *Pattern Classification.* Second Edition, John Wiley & Sons, 2000.

[10] **D. Erhan, Y. Bengio, A. Courville, P. Manzagol, P. Vincent, S. Bengio.** *Why does unsupervised pre-training help deep learning?* Journal of Machine Learning Research 11, pp. 625-660, 2010.

[11] **G.L. Grinblat.** *Máquinas de Vectores Soporte para Problemas No Estacionarios.* Tesis Doctoral en Informática, UNR. 2010.

[12] **G.L. Grinblat, L.C. Uzal, H.A. Ceccatto, P.M. Granitto.** *Solving non-stationary classification problems with coupled support vector machines.* IEEE Transactions on Neural Networks 22(1), pp. 37-52, 2011.

[13] **J. Hastad, M. Goldmann.** *On the power of small-depth threshold circuits.* Computational Complexity, vol. 1, pp. 113-129, 1991.

[14] **D.P. Helmbold, P.M. Long.** *Tracking drifting concepts by minimizing disagreements.* Machine Learning 15(3), pp. 279-298. 1994.

[15] **G.E. Hinton.** *Training products of experts by minimizing contrastive divergence.* Neural Computation 14(8), pp. 1711-1800. 2002.

[16] **G.E. Hinton, S. Osindero, Y. Teh.** *A fast learning algorithm for deep belief nets.* Neural Computation 18, pp. 1527-1554. 2006.

[17] **G.E. Hinton.** *Boltzmann machine.* Scholarpedia, 2(5):1668. 2007.

[18] **G.E. Hinton.** *A practical guide to training restricted Boltzmann machines.* Technical Report UTML-TR-2010-003, Department of Computer Science, University of Toronto, Canada, 2010.

[19] **J.J. Hopfield.** *Neural networks and physical systems with emergent collective computational abilities.* Proceedings of the National Academy of Sciences 79, pp. 2554-2558, 1982.

[20] **R. Klinkenberg, T. Joachims.** *Detecting concept drift with support vector machines.* Proceedings of the 17^{th} International Conference on Machine Learning, pp. 487-494, 2000.

[21] **R. Klinkenberg.** *Learning drifting concepts: example selection vs. example weighting.* Intelligent Data Analysis 8, pp. 281-300, 2004.

[22] **J. Kolter, M. Maloof.** *Dynamic Weighted Majority: An ensemble method for drifting concepts.* Journal of Machine Learning Research 8, pp. 2755-2790, 2007.

[23] **I. Koychev.** *Gradual Forgetting for adaptation to concept drift.* Proceedings of ECAI 2000 Workshop, Current Issues in Spatio-Temporal Reasoning, pp. 101-106, 2000.

[24] **A. Kuh, T. Petsche, R.L. Rivest.** *Learning time-variyng concepts.* NIPS, pp. 183-190, 1991.

[25] **H. Larochelle, Y. Bengio, J. Louradour, P. Lamblin.** *Exploring strategies for training deep neural networks.* Journal of Machine Learning Research 10, pp. 1-40, 2009.

[26] **H. Lee, R. Grosse, R. Ranganath, A. Ng.** *Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations.* Proceedings of the 26^{th} International Conference on Machine Learning, pp. 609-616, 2009.

[27] **Chih-Chung Chang and Chih-Jen Lin.** *LIBSVM: A library for support vector machines.* ACM Transactions on Intelligent Systems and Technology 2, pp. 27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[28] **T. Mitchell.** *Machine Learning.* McGraw-Hill, 1997.

[29] **Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.** *Gradient-based learning applied to document recognition.* Proceedings of the IEEE 86(11), pp. 2278-2324, November 1998.

[30] **W. Nick Street, Y. Kim.** *A Streaming Ensemble Algorithm (SEA) for large-scale classification.* Proceedings of the 7^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 377-382, 2001.

[31] **R. Quinlan.** *C4.5* http://www.rulequest.com/Personal/

[32] **M. Ranzato, A. Krizhevsky, G. Hinton.** *Factored 3-way restricted Boltzmann machines for modeling natural images.* JMLR Workshop and Confference Proceedings 9: Proceedings of the 13^{th} International Workshop on Artificial Intelligence and Statistics, pp. 621-628, 2010.

[33] **R. Salakhutdinov.** *Learning deep Boltzmann machines using adaptive MCMC.* Proceedings of the 27^{th} International Conference on Machine Learning, pp. 943-950, 2010.

[34] **J.C. Schlimmer, R.H. Granger Jr.** *Incremental learning from noisy data.* Machine Learning 1, pp. 317-354, 1986.

[35] **Y. LeCun, F.J. Huang, L. Bottou.** *Learning methods for generic object recognition with invariance to pose and lighting.* IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), 2004.

[36] **P. Smolensky.** *Information processing in dynamical systems: Foundations of Harmony Theory.* En D. E. Rumelhart, J. L. McClelland and the PDP Research Group, Parallel Distributed Processing 1: Foundations, pp. 194-281. MIT Press, Cambridge, MA, 1986.

[37] **K.O. Stanley.** *Learning Concept Drift with a Committee of Decision Trees.* Technical Report UTAI-TR-03-302, Department of Computer Sciences, University of Texas at Austin, USA, 2003.

[38] **Y. Tang, C. Eliasmith.** *Deep networks for robust visual recognition.* Proceedings of the 27^{th} International Conference on Machine Learning, pp. 1055-1062, 2010.

[39] **A. Tsymbal.** *The problem of concept drift: definitions and related work.* Technical Report TCD-CS-2004-15, Department of Computer Science, Trinity College Dublin, Ireland, 2004.

[40] **J. Weston, F. Ratle, R. Collobert.** *Deep learning via semi-supervised embedding.* Proceedings of the 25^{th} International Conference on Machine Learning, pp. 1168-1176, 2008.

[41] **G. Widmer, M. Kubat.** *Learning in the presence of concept drift and hidden contexts.* Machine Learning 23, pp. 69-101, 1996.

[42] **I. Žliobaitė.** *Learning under Concept Drift: an Overview.* Technical Report arXiv:1010.4784, Faculty of Mathematics and Informatics, Vilnius University, Lithuania, 2009.