Máster en Física Avanzada

Redes Neuronales y Complejas

Problemas de Evaluación

Problema 1. Demostrar que la relación entre el campo magnético y la magnetización es:

$$\frac{h}{k_B T} \approx M(1-\tau) + M^3(\tau - \tau^2 + \tau^3/3 + \dots)$$

Solución:

Podemos escribir la ecuación de estado en la teoría de campo medio en la vecindad de T_c ($\tau = \frac{T_c}{T},\ (\tau \approx 1),\ (1-\tau \approx 0)$) como:

$$M = tanh(\frac{h}{k_B T} + M\tau)$$

Usando la identidad trigonométrica $tanh(x+y) = \frac{tanhx + tanhy}{1+(tanhx)(tanhy)}$, tenemos que:

$$M = \frac{tanh(\frac{h}{k_B T}) + tanh(M\tau)}{1 + tanh(\frac{h}{k_B T}) \cdot tanh(M\tau)}$$

que podemos reescribir como

$$tanh(\frac{h}{k_B T}) = \frac{tanhM - tanh(M\tau))}{1 - (tanhM) \cdot tanh(M\tau)}$$

Cerca del punto crítico, esto es $(h, M) \ll 1$ y $\tau \approx 1$, los argumentos de todas las funciones tangente hiperbólicas que aparecen en la ecuación son pequeños. Por lo tanto, podemos usar la expansión en serie de Taylor para $tanhx$:

$$tanhx = x - \frac{x^3}{3} + \frac{2}{15}x^5 + \dots$$

para obtener, en primer lugar

$$\frac{h}{k_B T} \approx \frac{M - M\tau + \frac{1}{3}M^3\tau^3}{1 - M^2\tau}$$

y dado que M es pequeño, podemos expandir el denominador como $1/(1-\delta) \approx 1 + \delta$, de modo que quedándonos solo hasta los términos $O(M^3)$, tenemos:

$$\frac{h}{k_B T} \approx (M - M\tau + \frac{1}{3}M^3\tau^3) \cdot (1 - M^2\tau)$$

$$= M - M\tau + \frac{1}{3}M^3\tau^3 + M^3\tau - M^3\tau^2 + O(M^5)$$

que podemos reescribir como:

$$\frac{h}{k_B T} \approx M(1-\tau) + M^3(\tau - \tau^2 + \tau^3/3) + \dots$$

Problema 2. Hacer el resumen del cap. 1.2 Spike Trains and Firing Rates, P. Dayan, L. F. Abbott, Theoretical Neuroscience - Computational and Mathematical, Modeling of Neural Systems, The MIT Press, 2005.

Solución:

En este resumen abordaremos los conceptos principales del capítulo 1.2 del libro *Theoretical Neuroscience - Computational and Mathematical Modeling of Neural Systems* de P. Dayan y L. F. Abbott. Este capítulo introduce medidas básicas de la actividad neural y explica cómo los potenciales de acción (impulsos o también espigas) se utilizan para transmitir información en el sistema nervioso.

Las neuronas transmiten información a través de secuencias de potenciales de acción o trenes de impulsos (spikes). Aunque los impulsos individuales pueden variar ligeramente en duración, amplitud y forma, en estudios de codificación neural se tratan como eventos estereotipados e idénticos. Si ignoramos la breve duración de un impulso (aproximadamente 1 ms), una secuencia de impulsos se puede caracterizar simplemente por una lista de los tiempos en los que ocurrieron los impulsos. Para n impulsos, estos tiempos se denotan por t_i con $i = 1, 2, \ldots, n$. El ensayo durante el cual se registran los impulsos comienza en el tiempo 0 y termina en el tiempo T, así que $0 \leq t_i \leq T$ para todos los i.

La función de respuesta neural $\rho(t)$ representa un tren de impulsos como una suma de funciones delta de Dirac en los tiempos de impulso t_i:

$$\rho(t) = \sum_{i=1}^{n} \delta(t - t_i)$$

Esto permite expresar sumas sobre impulsos como integrales en el tiempo. Para cualquier función bien comportada $h(t)$, podemos escribir:

$$\sum_{i=1}^{n} h(t - t_i) = \int_{-\infty}^{\infty} d\tau h(\tau)\rho(t - \tau)$$

Dado que la secuencia de potenciales de acción generada por un estímulo específico varía de un ensayo a otro, las respuestas neuronales se tratan típicamente de manera estadística o probabilística. Por ejemplo, pueden caracterizarse por tasas de disparo, en lugar de secuencias específicas de impulsos. La tasa de conteo de impulsos se obtiene contando el número de potenciales de acción que aparecen durante un ensayo y dividiendo por la duración del ensayo. Denotamos la tasa de conteo de impulsos por r:

$$r = \frac{n}{T} = \frac{1}{T} \int_{0}^{T} d\tau \rho(\tau)$$

Una tasa de disparo dependiente del tiempo puede definirse contando impulsos en intervalos de tiempo cortos y promediando sobre múltiples ensayos. La tasa de disparo dependiente del tiempo se da por:

$$r(t) = \frac{1}{\Delta t} \int_{t}^{t+\Delta t} d\tau \langle \rho(\tau) \rangle$$

Esta expresión establece una relación importante entre la función de respuesta neural promedio y la tasa de disparo.

Para medir la tasa de disparo $r(t)$ de manera más precisa, se pueden usar diferentes métodos como el binning o el uso de una ventana deslizante. Por ejemplo, una función de ventana Gaussiana puede proporcionar una estimación suave de la tasa de disparo:

$$w(\tau) = \frac{1}{\sqrt{2\pi}\sigma_w} \exp\left(-\frac{\tau^2}{2\sigma_w^2}\right)$$

En este caso, σ_w controla la resolución temporal de la tasa resultante, jugando un papel análogo a Δt. Una función de ventana continua como la Gaussiana genera una estimación de la tasa de disparo que es una función suave del tiempo.

Tanto las funciones de ventana rectangular como Gaussiana aproximan la tasa de disparo en cualquier tiempo, utilizando impulsos disparadas tanto antes como después de ese tiempo. Una neurona postsináptica que monitorea el tren de impulsos de una célula presináptica solo tiene acceso a los impulsos que han ocurrido previamente. Una aproximación de la tasa de disparo en el tiempo t que depende solo de los impulsos disparadas antes de t puede calcularse utilizando una función de ventana que se anule cuando su argumento es negativo. Tal función de ventana o núcleo se llama causal. Una forma comúnmente utilizada es la función α:

$$w(\tau) = [\alpha^2 \tau \exp(-\alpha\tau)]^+$$

donde $1/\alpha$ determina la resolución temporal de la estimación de la tasa de disparo resultante. La notación $[z]^+$ para cualquier cantidad z representa la operación de rectificación de media onda:

$$[z]^+ = \begin{cases} z & \text{si } z \geq 0 \\ 0 & \text{otro caso} \end{cases}$$

Las respuestas neuronales típicamente dependen de muchas propiedades diferentes de un estímulo. En este capítulo, caracterizamos las respuestas de las neuronas como funciones de solo uno de los atributos del estímulo al que pueden ser sensibles. El valor de este único atributo se denota por s. Una forma simple de caracterizar la respuesta de una neurona es contar el número de potenciales de acción disparados durante la presentación de un estímulo. Este enfoque es más apropiado si el parámetro s que caracteriza el estímulo se mantiene constante durante el ensayo. Si promediamos el número de potenciales de acción disparados sobre (en teoría, un número infinito de) ensayos y dividimos por la duración del ensayo, obtenemos la tasa de disparo promedio, $\langle r \rangle$, definida anteriormente. La tasa de disparo promedio escrita como una función de s, $\langle r \rangle = f(s)$, se llama la curva de sintonización de la respuesta neural. La forma funcional de una curva de sintonización depende del parámetro s utilizado para describir el estímulo. La elección precisa de los parámetros utilizados como argumentos de las funciones de la curva de sintonización es en parte una cuestión de convención. Debido a que las curvas de sintonización corresponden a tasas de disparo, se miden en unidades de impulsos por segundo o Hz.

Un ejemplo es una neurona en la corteza visual primaria (V1) que responde a diferentes orientaciones de una barra de luz. La curva de sintonización puede ajustarse mediante una Gaussiana:

$$f(s) = r_{\max} \exp\left(-\frac{1}{2}\left(\frac{s - s_{\max}}{\sigma_f}\right)^2\right)$$

donde s es el ángulo de orientación de la barra de luz, $s_{\max}$ es el ángulo de orientación que evoca la respuesta promedio máxima $r_{\max}$ (con $s - s_{\max}$ tomado en el rango entre -90° y +90°), y σ_f determina

el ancho de la curva de sintonización. La neurona responde más vigorosamente cuando se presenta un estímulo con $s = s_{\max}$, por lo que llamamos $s_{\max}$ el ángulo de orientación preferido de la neurona.

Otro ejemplo es la corteza motora primaria (M1), donde las neuronas responden a la dirección del movimiento del brazo. La curva de sintonización puede describirse mediante una función coseno:

$$f(s) = r_0 + (r_{\max} - r_0)\cos(s - s_{\max})$$

Para algunas neuronas, la curva de sintonización requiere rectificación de media onda:

$$f(s) = [r_0 + (r_{\max} - r_0)\cos(s - s_{\max})]^+$$

Un último ejemplo es la disparidad retiniana en la corteza visual primaria (V1), que puede describirse mediante una función sigmoidal:

$$f(s) = \frac{r_{\max}}{1 + \exp\left(\frac{s_{1/2} - s}{\Delta s}\right)}$$

Las curvas de sintonización permiten predecir la tasa de disparo promedio, pero no describen cómo varía la tasa de conteo de impulsos r respecto a su valor medio $\langle r \rangle = f(s)$ de un ensayo a otro. Mientras que el mapeo de estímulo a respuesta promedio puede describirse determinísticamente, es probable que las respuestas de ensayos individuales, como las tasas de conteo de impulsos, solo puedan modelarse de manera probabilística. Por ejemplo, los valores de r pueden generarse a partir de una distribución de probabilidad con media $f(s)$. La desviación de ensayo a ensayo de r respecto a $f(s)$ se considera ruido, y tales modelos a menudo se llaman modelos de ruido. La desviación estándar para la distribución del ruido puede ser independiente de $f(s)$, en cuyo caso la variabilidad se llama ruido aditivo, o puede depender de $f(s)$. El ruido multiplicativo corresponde a tener la desviación estándar proporcional a $f(s)$.

La variabilidad en la respuesta se extiende más allá del nivel de los conteos de impulsos a todo el patrón temporal de los potenciales de acción. Un modelo del capítulo discute la respuesta neuronal utilizando un generador de impulsos estocástico para producir variabilidad en la respuesta. Este enfoque toma una estimación determinista de la tasa de disparo $r_{\text{est}}(t)$ y produce un patrón de impulsos estocástico a partir de ella. El generador de impulsos produce números y patrones variables de potenciales de acción, incluso si se utiliza la misma tasa de disparo estimada en cada ensayo.

Problema 2OPT. Simulación del modelo de Hodgkin-Huxley o de Leaky Integrate-and-Fire usando software especifico

Solución:

Introducción

El modelo de Hodgkin-Huxley es un modelo matemático que describe cómo las neuronas inician y propagan los potenciales de acción. Fue desarrollado por Alan Hodgkin y Andrew Huxley en 1952 y se basa en experimentos realizados en el axón gigante de calamar. El modelo describe las variaciones de potencial de membrana de una neurona en función del tiempo mediante un conjunto de ecuaciones diferenciales no lineales que representan los canales de sodio (Na^+) y potasio (K^+), así como la corriente de fuga. Las ecuaciones fundamentales del modelo son:

$$C_m \frac{dV}{dt} = I - \bar{g}_{Na} m^3 h (V - E_{Na}) - \bar{g}_K n^4 (V - E_K) - \bar{g}_L (V - E_L), \tag{1}$$

donde:

- C_m es la capacitancia de la membrana.
- V es el potencial de membrana.
- I es la corriente externa aplicada.
- $\bar{g}_{Na}$ y $\bar{g}_K$ son las conductancias máximas de los canales de sodio y potasio.
- E_{Na} y E_K son los potenciales de equilibrio para el sodio y el potasio.
- $\bar{g}_L$ es la conductancia de fuga y E_L es el potencial de equilibrio de la fuga.
- m, h, y n son variables de compuerta que siguen sus propias ecuaciones diferenciales.

Las derivadas temporales de m, h y n están dadas por:

$$\frac{dm}{dt} = \alpha_m (1 - m) - \beta_m m, \tag{2}$$

$$\frac{dh}{dt} = \alpha_h (1 - h) - \beta_h h, \tag{3}$$

$$\frac{dn}{dt} = \alpha_n (1 - n) - \beta_n n, \tag{4}$$

donde α y β son funciones del potencial de membrana V.

El software utilizado para esta simulación es el *HHsim*, un simulador específico para el modelo de Hodgkin-Huxley desarrollado por David Terman y David Schulz. Este software permite la simulación detallada y visualización de los comportamientos dinámicos de la membrana neuronal bajo diferentes condiciones y parámetros.

Metodología

Para realizar la simulación del modelo de Hodgkin-Huxley, se utilizó el simulador *HHsim*, disponible en `http://www.cs.cmu.edu/~dst/HHsim/`. A continuación, se detallan las principales ventanas y funciones del simulador:

Ventana Principal

La ventana principal del *HHsim* muestra las trazas del potencial de membrana y las variables de compuerta (m, h, y n) en función del tiempo. Aquí se pueden observar los efectos de los estímulos aplicados y ajustar diversos parámetros para analizar los resultados.

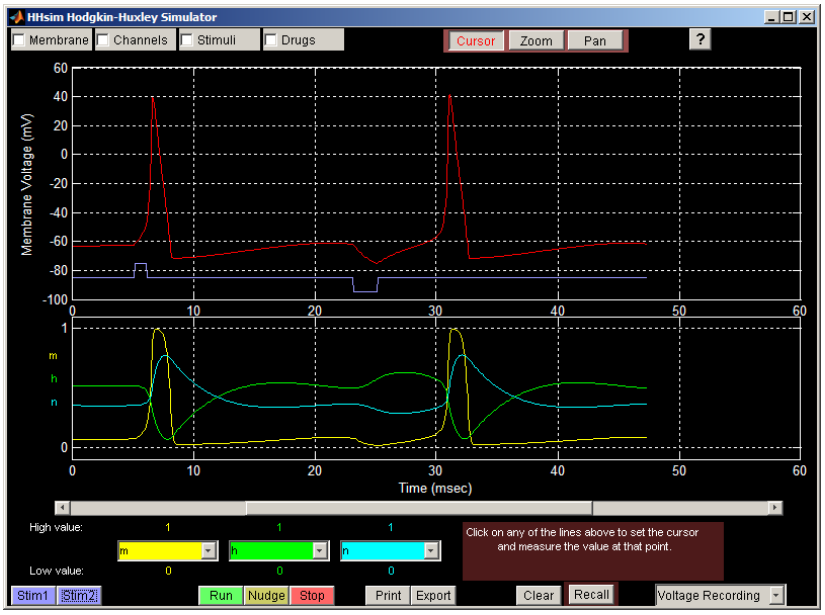


Figura 1: Ventana principal del simulador *HHsim*.

Configuración de la Membrana

En esta ventana se pueden ajustar las concentraciones intracelulares y extracelulares de los iones, los potenciales de equilibrio y otros parámetros como la temperatura y la resistencia de la membrana.

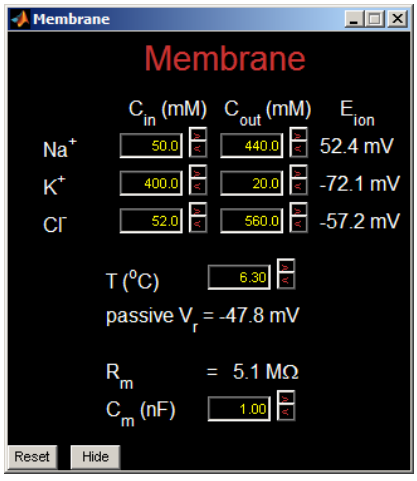


Figura 2: Ventana de configuración de la membrana.

Canales Iónicos

Aquí se pueden activar o desactivar los canales iónicos pasivos y activos, y ajustar sus conductancias. Esto permite analizar cómo cada tipo de canal contribuye a la generación del potencial de acción.

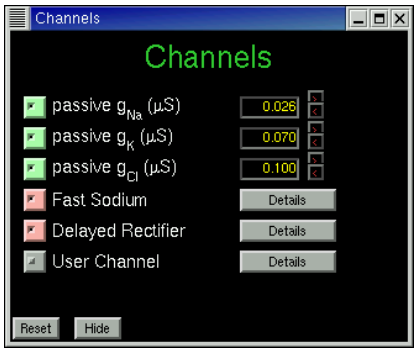


Figura 3: Ventana de configuración de los canales iónicos.

Aplicación de Drogas

El simulador incluye opciones para aplicar drogas que inhiben corrientes de sodio (Na^+) o potasio (K^+), como la tetrodotoxina (TTX) y el tetraetilamonio (TEA). Esto permite estudiar los efectos farmacológicos sobre el potencial de acción.

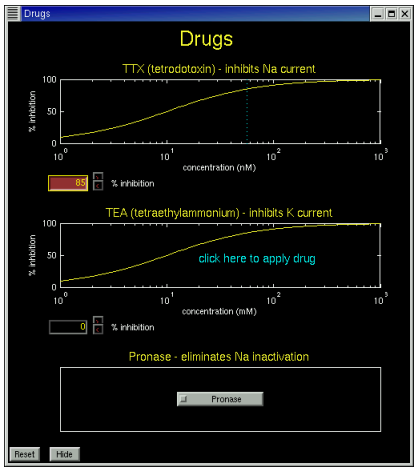


Figura 4: Ventana de aplicación de drogas.

Estímulos

La ventana de estímulos permite definir los patrones de corriente que se aplicarán a la membrana para inducir potenciales de acción. Se pueden configurar múltiples estímulos con diferentes amplitudes y duraciones.

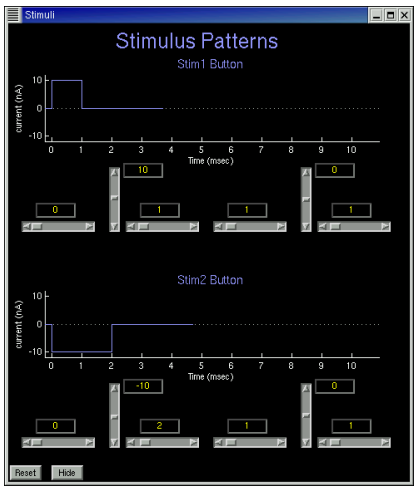


Figura 5: Ventana de configuración de los estímulos.

Resultados

Parte I. Potencial de Equilibrio

Para realizar una simulación con el modelo de Hodgkin-Huxley, primero vamos a realizar un pequeño experimento para analizar el simulador *HHsim*. Asi, vamos a explorar el potencial de equilibrio de una célula con un solo tipo de canal. Para ello, se desactivaron todos los canales excepto el canal pasivo de sodio. El potencial de reposo V_r mostrado en la ventana de la membrana fue igual al potencial de inversión del sodio, $E_{Na} = 52{,}4$ mV.

En primer lugar, se redujo la concentración externa de sodio (C_{out}) a 220 mM. Sabemos que, de acuerdo a la ecuación de Nernst, el potencial de reposo V_r es:

$$V_r = \frac{RT}{zF} \ln\left(\frac{C_{\text{out}}}{C_{\text{in}}}\right), \tag{5}$$

donde:

- R es la constante de los gases (8.314 J/(mol·K)).
- T es la temperatura en Kelvin.
- z es la valencia del ion (para Na^+, $z = 1$).
- F es la constante de Faraday (96485 C/mol).

A una temperatura de 6,3°C (279.45 K):

$$\frac{RT}{zF} = \frac{8{,}314 \times 279{,}45}{96485} \approx 0{,}0241 \text{ V} \tag{6}$$

Así, al reducir C_{out} a 220 mM:

$$E_{Na} = V_r = 0{,}0241 \ln\left(\frac{220}{50}\right) \approx 35{,}7 \text{ mV} \tag{7}$$

Esto se confirmó en el simulador, donde V_r se ajustó a 35.7 mV al cambiar C_{out} a 220 mM.

Después, quisimos hacer otra prueba para encontrar la concentración externa de Na^+ necesaria para alcanzar $V_r = E_{Na} = 55{,}5$ mV. De acuerdo a la ecuación de Nernst esto es:

$$55{,}5 \cdot 10^{-3} = 0{,}0241 \ln\left(\frac{C_{\text{out}}}{50}\right) \tag{8}$$

Despejando C_{out}:

$$C_{\text{out}} = 50 \times e^{\frac{55{,}5}{0{,}0241}} = 500{,}16 \text{ mM} \tag{9}$$

Esto se confirmó en el simulador, donde V_r se ajustó a 55.5 mV al cambiar C_{out} a 500 mM.

Por último, volvimos a poner los parámetros de la concentración a sus valores normales y se comprobó que ocurre al duplicar la temperatura de 6,3°C a 12,6°C. Calculamos el nuevo valor de $\frac{RT}{zF}$:

$$\frac{RT}{zF} = \frac{8{,}314 \times 285{,}75}{96485} \approx 0{,}0246 \text{ V} \tag{10}$$

El nuevo potencial de reposo es:

$$E_{Na} = 0{,}0246 \ln\left(\frac{440}{50}\right) \approx 53{,}5 \text{ mV} \tag{11}$$

Mientras que con el simulador obtubimos 53,6 mV. El potencial de reposo no se duplicó porque el potencial de equilibrio depende logarítmicamente de la relación de concentraciones y linealmente de la temperatura en Kelvin.

Como vemos, el simulador nos permite obtener resultados precisos y coherentes con la teoría.

Parte II. Potencial de Acción

Para analizar y simular el potencial de acción, la célula fue estimulada utilizando el botón *Stim1* en la ventana principal, generando un pico en el potencial de membrana visible en la línea roja. La estimulación con *Stim2* aplicó un pulso hiperpolarizante a la célula, lo que también resultó en un pico de potencial de acción. Esto se debe a que la hiperpolarización inicial aumenta la conductancia de sodio, activando canales de sodio adicionales que llevan a una despolarización suficiente para disparar un potencial de acción.

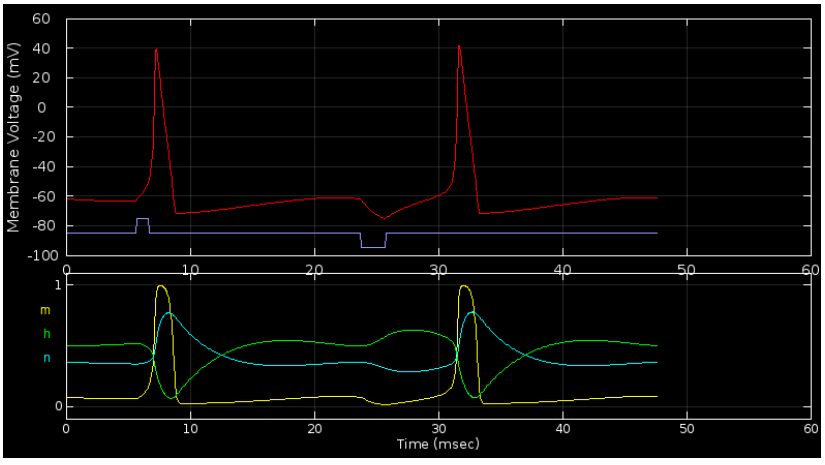


Figura 6: Hiperpolarización causando un pico de potencial de acción.

El siguiente experimento consistió en añadir un segundo pulso de 10 nA a *Stim1*, 5 ms después del primer pulso. No se observaron dos picos separados debido a que el segundo pulso ocurrió durante el período refractario absoluto, cuando las compuertas de los canales no permiten una segunda despolarización. Se determinó con el simulador que el tiempo necesario entre el final del primer pulso y el inicio del segundo para que este último cause un segundo pico es de 9 ms.

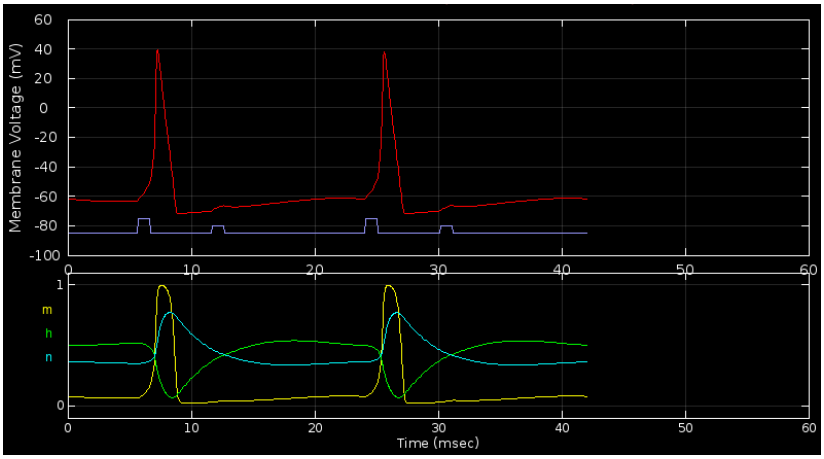


Figura 7: Dos pulsos de 10 nA separados por 5 ms.

Por último, se observó que un pulso negativo ocurrido poco después de un pulso positivo puede prevenir la ocurrencia de un pico de potencial de acción. Se configuró *Stim1* para proporcionar un pulso de por un pulso de -5 nA de 1ms despúes de un pulso de 5 nA de 1 ms. Se encontró usando el simulador que el intervalo de tiempo más largo para que el pulso negativo bloquee el pico es de 0.9 ms.

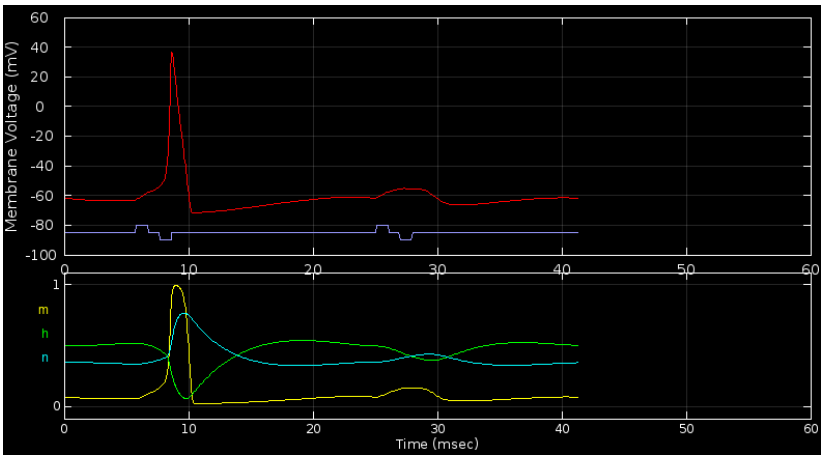


Figura 8: Intervalo de tiempo necesario para que un pulso negativo bloquee un pico.

Problema 3. TAREA de simulación sobre la parte de redes atractoras: Usando la regla de Hebb, estudiar la dinámica de aprendizaje en una red neuronal atractora de tipo Hopfield a la cual se presenta una imagen, por ejemplo, la letra A.

Solución:

Introducción

Las redes atractoras son un tipo especial de redes neuronales que poseen estados estables llamados atractores. Un atractor es un conjunto de estados hacia el cual el sistema evoluciona a partir de una variedad de condiciones iniciales. Las redes atractoras son ampliamente utilizadas para modelar la memoria asociativa y la recuperación de patrones en sistemas biológicos y artificiales.

Una red neuronal atractora de Hopfield es un tipo de red recurrente que se caracteriza por tener una dinámica que converge a uno de varios patrones almacenados. Estas redes consisten en neuronas binarias que pueden tomar valores de $+1$ o -1. La conexión entre las neuronas está determinada por una matriz de pesos sinápticos que se calcula durante la fase de entrenamiento. El funcionamiento de las redes de Hopfield se puede describir mediante la siguiente ecuación de actualización:

$$S_i(t+1) = \text{sign}\left(\sum_j J_{ij} S_j(t)\right)$$

donde $S_i(t)$ es el estado de la neurona i en el tiempo t, y J_{ij} es el peso sináptico entre las neuronas i y j (también se suele llamar W_{ij}). La evolución dinámica de la red la lleva a un estado estable o atractor, que corresponde a un patrón previamente aprendido.

La regla de Hebb, propuesta por Donald Hebb en 1949, es una regla de aprendizaje que se basa en la idea de que las conexiones entre dos neuronas se fortalecen si ambas están activas simultáneamente. En términos matemáticos, si las activaciones de dos neuronas i y j están correlacionadas, el peso sináptico W_{ij} entre ellas aumenta. La regla de Hebb se puede formular como:

$$J_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu$$

donde ξ_i^μ es el valor del patrón μ en la neurona i, N es el número de neuronas, y p es el número de patrones almacenados. Esta fórmula asegura que los patrones almacenados sean atractores del sistema, y que la red pueda recuperar estos patrones a partir de estados iniciales ruidosos o incompletos.

Las redes de Hopfield tienen aplicaciones en diversas áreas, incluyendo la memoria asociativa, el reconocimiento de patrones, y la optimización combinatoria. La capacidad de una red de Hopfield para recuperar un patrón almacenado a partir de un estado ruidoso o incompleto es una propiedad clave que emula la memoria humana, donde podemos reconocer y recordar información a partir de fragmentos o versiones distorsionadas.

La energía de la red de Hopfield se define como:

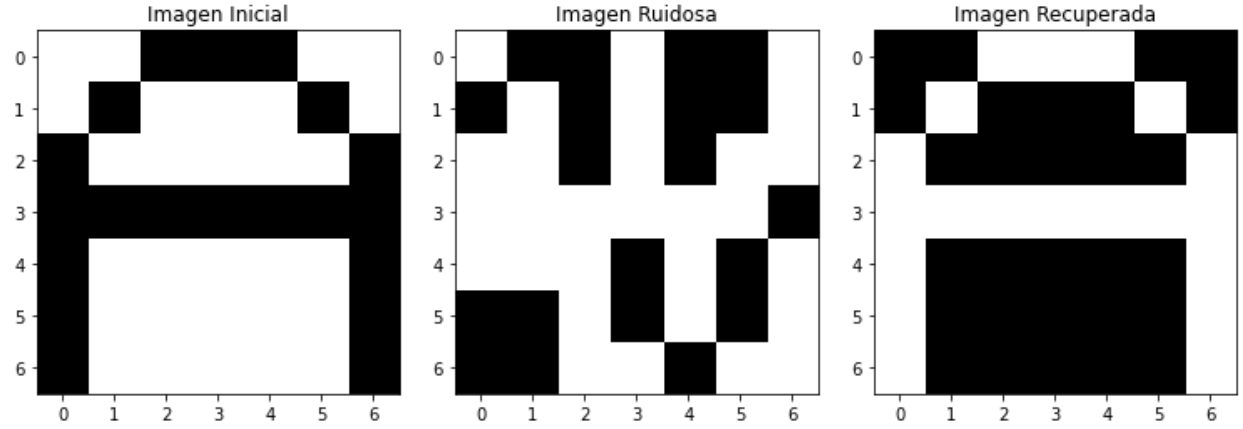
$$H = -\frac{1}{2} \sum_{i \neq j} J_{ij} S_i S_j$$

La dinámica de la red busca minimizar esta energía, llevando al sistema a uno de los atractores correspondientes a los patrones almacenados.

Resultados

Para demostrar la dinámica de aprendizaje en una red neuronal de tipo Hopfield, se presenta la imagen de la letra A. Primero, se dibuja la letra A en una matriz de 7x7, que luego se aplana a un vector de longitud 49. Posteriormente, se calcula la matriz de pesos sinápticos inicializada y entrenada usando la regla de Hebb. Se genera una versión ruidosa aleatoria de la letra A y se actualiza iterativamente para recuperar la imagen original.

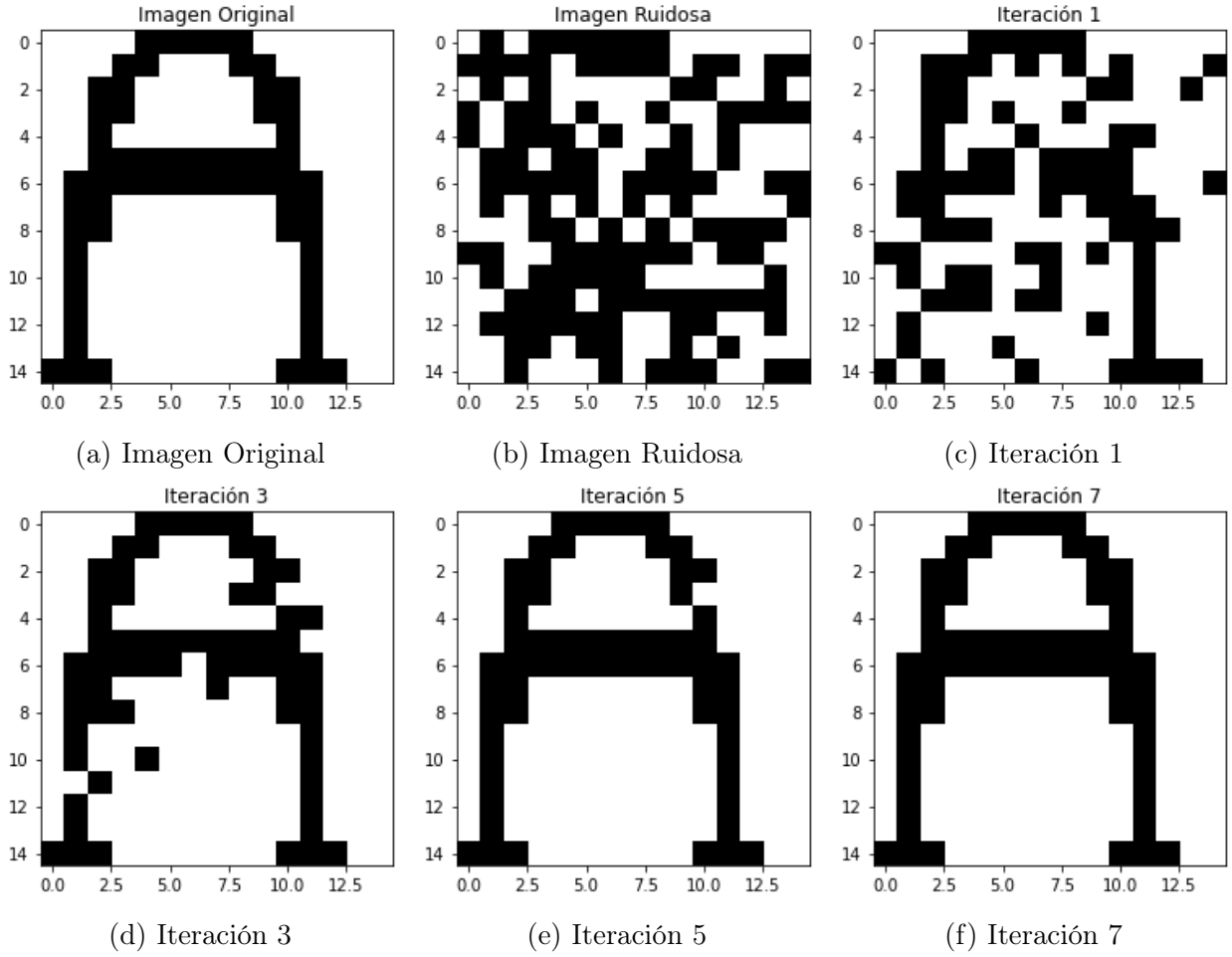
Tras un barrido de todos los elementos de la red (una iteración), se ha recuperado la imagen original (la letra A), como se muestra en la Figura 9. Es importante notar que la imagen recuperada se ha recuperado como el opuesto o negativo de la original debido a la naturaleza binaria de las neuronas en la red de Hopfield.



(a) Imagen Inicial (b) Imagen Ruidosa (c) Imagen Recuperada

Figura 9: Recuperación de la letra A en una red de Hopfield

Para aumentar la dificultad, se incrementó el tamaño de la matriz a 15x15 y se actualizó el estado de la red de forma asincrónica, actualizando una neurona aleatoria a la vez en cada iteración. En este caso, se observó cómo la imagen se va recuperando en cada iteración, obteniendo la imagen original en 7 iteraciones, como se puede observar en la Figura 10.



(a) Imagen Original (b) Imagen Ruidosa (c) Iteración 1

(d) Iteración 3 (e) Iteración 5 (f) Iteración 7

Figura 10: Recuperación de la letra A en una red de Hopfield asincrónica

Problema 4. TAREA OPCIONAL de simulación sobre la parte de redes atractoras segunda parte: Una vez estudiada la dinámica de aprendizaje en la tarea 3, repetir el análisis en el caso de una red neuronal diluida y ver qué efecto tiene la dilución en la recuperación de la imagen.

Solución:

Red Neuronal Diluida

En esta sección, repetimos el análisis de la dinámica de aprendizaje utilizando una red neuronal diluida. Una red neuronal diluida es aquella en la que un porcentaje de las conexiones sinápticas han sido eliminadas, reduciendo así la conectividad entre las neuronas. Esto nos permite estudiar el efecto de la dilución en la recuperación de la imagen. Podemos definir el peso sináptico diluido $\hat{J}_{ij}$ como

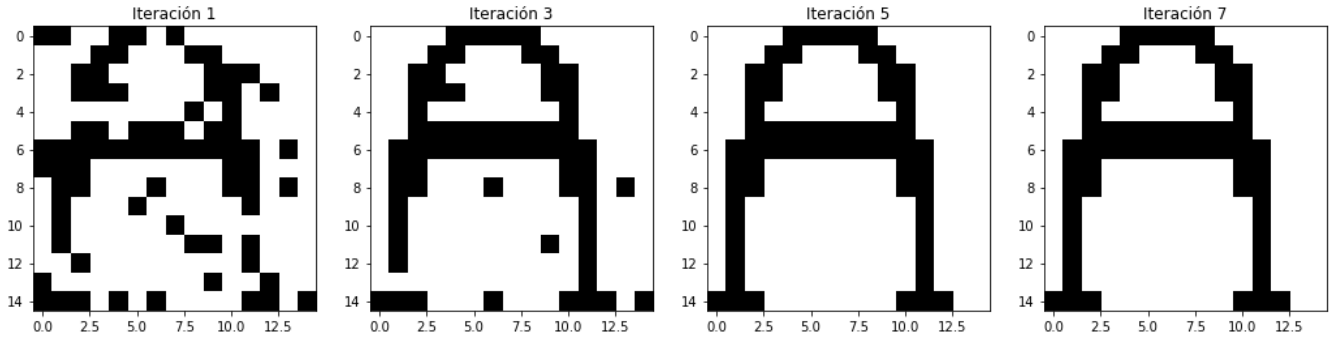
$$\hat{J}_{ij} = \begin{cases} 0 & \text{con } P(d) \\ J_{ij} & \text{con } 1 - P(d) \end{cases}$$

donde $P(d)$ es la tasa de dilución. Para implementar la red diluida, primero aplicamos la regla de Hebb para calcular la matriz de pesos sinápticos J_{ij}. Luego, introducimos la dilución eliminando un porcentaje específico de las conexiones de acuedo a la tasa de dilución $P(d)$.

Para estudiar el efecto de la dilución en la recuperación de la imagen, hemos realizado el análisis para diluciones del 10 %, 50 %, 90 % y 98 %. Siguiendo el mismo procedimiento que en el ejercicio anterior, partimos de la imagen original, generamos una imagen completamente ruidosa y actualizamos el estado de la red de forma asincrónica para ver cómo se recupera la imagen original. Se estudia la dinámica de recuperación durante 7 iteraciones.

Dilución del 10 %

La Figura 11 muestra los resultados obtenidos para una dilución del 10 %. Se observa que, incluso con una dilución baja, la imagen se recupera progresivamente en cada iteración.

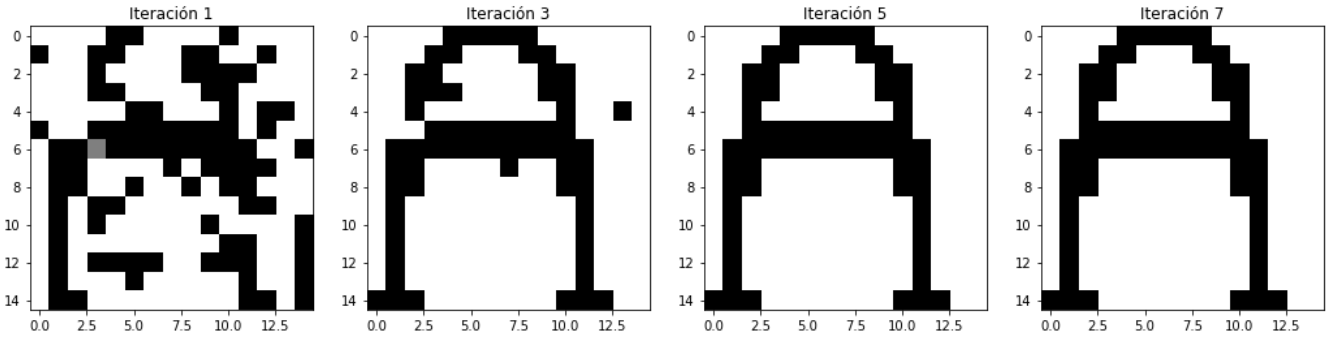


(a) Iteración 1 (b) Iteración 3 (c) Iteración 5 (d) Iteración 7

Figura 11: Recuperación de la letra A con una dilución del 10 %.

Dilución del 50 %

La Figura 12 muestra los resultados obtenidos para una dilución del 50 %. Con una dilución moderada, se observa que la red aún es capaz de recuperar la imagen, aunque con mayor dificultad.

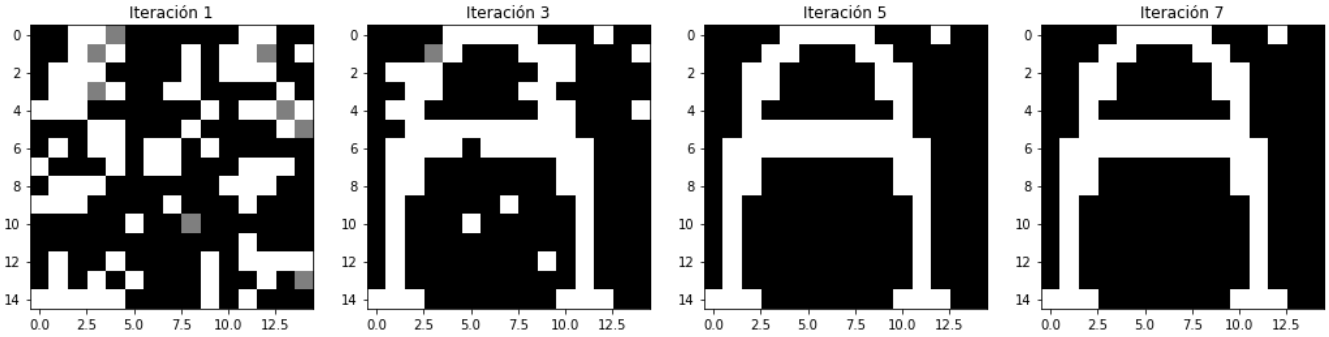


(a) Iteración 1 (b) Iteración 3 (c) Iteración 5 (d) Iteración 7

Figura 12: Recuperación de la letra Acon una dilución del 50 %.

Dilución del 90 %

La Figura 13 muestra los resultados obtenidos para una dilución del 90 %. Con una alta dilución, la red tiene dificultades significativas para recuperar la imagen original.

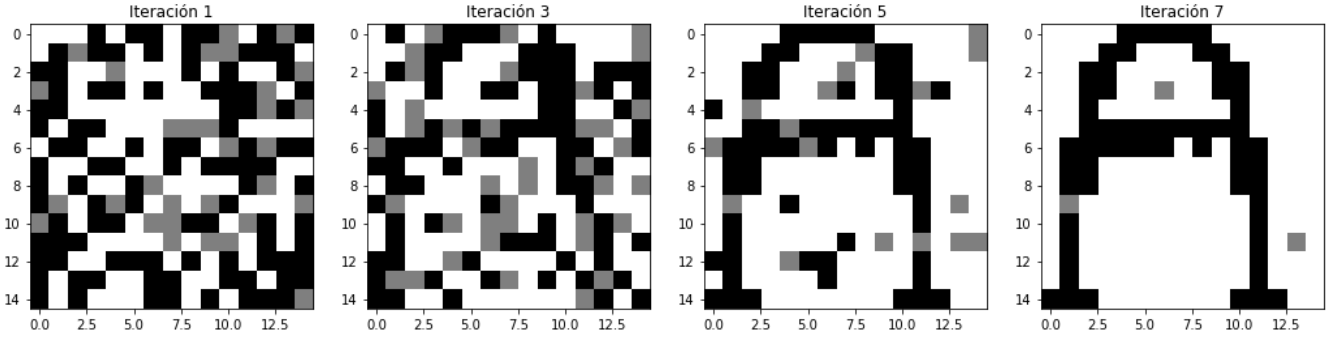


(a) Iteración 1 (b) Iteración 3 (c) Iteración 5 (d) Iteración 7

Figura 13: Recuperación de la letra A con una dilución del 90 %.

Dilución del 98 %

La Figura 14 muestra los resultados obtenidos para una dilución del 98 %. Con una dilución muy alta, la capacidad de la red para recuperar la imagen es muy limitada.



(a) Iteración 1 (b) Iteración 3 (c) Iteración 5 (d) Iteración 7

Figura 14: Recuperación de la letra A con una dilución del 98 %.

El análisis muestra que, a medida que aumenta el grado de dilución, la red neuronal de Hopfield tiene más dificultades para recuperar la imagen original. Con una dilución del 10 %, la red puede recuperar la imagen completamente después de 7 iteraciones, sin mucha diferencia con la del ejercicio anterior sin dilución. Sin embargo, con una dilución del 50 %, la recuperación es algo más dificultosa para las primera iteraciones, aunque se recupera la imagen original tras las 7 iteraciones. Con una dilución del 90 %, observamos que, salvo por un pixel, se recupera la imagen tras las 7 iteraciones, aunque se nota más dificultad en las primeras iteraciones. Por último, con 98 % de tasa, observamos que la red es incapaz de recuperar completamente la imagen original de manera efectiva, aunque se queda muy cerca de conseguirlo. Es impresionante observar como a pesar de diluir tanto la red neuronal, esta es capaz de recuperar una buena parte de la imagen original en unas cuantas iteraciones, mostrando además que la conectividad entre las neuronas es crucial para la capacidad de recuperación de la red.

Problema 5. Demostrar la expresión por el método de la inducción para C(P,N) - el problema de Cover.

Solución:

Queremos demostrar la siguiente expresión para $C(P, N)$ utilizando el método de inducción:

$$C(P,N) = \begin{cases} 2^P & \text{para } P \leq N \\ 2\sum_{i=0}^{N-1} \binom{P-1}{i} & \text{para } P > N \end{cases}$$

donde $C(P, N)$ representa el número de dicotomías linealmente separables (coloraciones que pueden ser definidas por un hiperplano a través del origen que separa puntos negros y blancos). Vamos a usar los siguientes pasos para demostrar esta ecuación.

Paso 1: Base de la inducción

Verificamos la expresión para $P = 1$.

Para $P = 1$ y cualquier $N \geq 1$:

$$C(1, N) = 2$$

Esto coincide con la expresión dada ya que $2^1 = 2$.

Paso 2: Hipótesis de inducción

Supongamos que la fórmula es correcta para algún P. Es decir, suponemos que la expresión:

$$C(P,N) = \begin{cases} 2^P & \text{para } P \leq N \\ 2\sum_{i=0}^{N-1} \binom{P-1}{i} & \text{para } P > N \end{cases}$$

es correcta.

Paso 3: Paso inductivo

Queremos demostrar que la fórmula es correcta para $P + 1$. Tenemos los siguientes casos:

Caso 1: $P + 1 \leq N$

Para este caso, entonces la hipótesis de inducción es:

$$C(P, N) = 2^P$$

Debemos mostrar que:

$$C(P + 1, N) = 2^{P+1}$$

Usamos la relación de recurrencia:

$$C(P + 1, N) = C(P, N) + C(P, N - 1)$$

Dado que $P + 1 \leq N$, también ocurre que $P \leq N$ y $P \leq N - 1$, así que:

$$C(P, N) = 2^P$$

$$C(P, N-1) = 2^P$$

Por lo tanto:

$$C(P+1, N) = 2^P + 2^P = 2 \cdot 2^P = 2^{P+1}$$

Luego para este caso queda demostrada.

Caso 2: $P + 1 > N$ Para este otra caso, tenemos que según la hipótesis de inducción:

$$C(P, N) = 2 \sum_{i=0}^{N-1} \binom{P-1}{i}$$

Debemos demostrar que:

$$C(P+1, N) = 2 \sum_{i=0}^{N-1} \binom{P}{i}$$

De nuevo usamos la relación de recurrencia:

$$C(P+1, N) = C(P, N) + C(P, N-1)$$

Dado que $P + 1 > N$, también se cumple que $P > N - 1$, que según la hipótesis de inducción es:

$$C(P, N-1) = 2 \sum_{i=0}^{N-2} \binom{P-1}{i}$$

Por lo tanto, tenemos que según la relación de recurrencia se cumple:

$$C(P+1, N) = 2 \sum_{i=0}^{N-1} \binom{P-1}{i} + 2 \sum_{i=0}^{N-2} \binom{P-1}{i} = 2 \left(\sum_{i=0}^{N-1} \binom{P-1}{i} + \sum_{i=0}^{N-2} \binom{P-1}{i} \right)$$

Ahora, usamos la propiedad de los coeficientes binomiales:

$$\binom{P}{i} = \binom{P-1}{i} + \binom{P-1}{i-1}$$

Entonces, notando que $\sum_{i=0}^{N-2} \binom{P-1}{i} = \sum_{i=1}^{N-1} \binom{P-1}{i-1}$, tenemos:

$$\sum_{i=0}^{N-1} \binom{P}{i} = \binom{P}{0} + \binom{P}{1} + \cdots + \binom{P}{N-1} = \sum_{i=0}^{N-1} \binom{P-1}{i} + \sum_{i=1}^{N-1} \binom{P-1}{i-1}$$

Por lo tanto, se cumple que:

$$\sum_{i=0}^{N-1} \binom{P}{i} = \sum_{i=0}^{N-1} \binom{P-1}{i} + \sum_{i=1}^{N-1} \binom{P-1}{i-1}$$

Y finalmente, queda demostrado que:

$$C(P+1, N) = 2 \sum_{i=0}^{N-1} \binom{P}{i}$$

Por lo tanto, mediante el método de inducción, hemos demostrado que la expresión $C(P, N)$ es correcta para todos los P.

Problema 6. Obtener la expresion para el error de generalización.

Solución:

Queremos demostrar que, sea

$$P(x,y) = \frac{1}{2\pi\sqrt{Q-R^2}} exp\left[-\frac{1}{2}\frac{x^2+Qy^2-2Rxy}{Q-R^2}\right]$$

el error de generalización es

$$\varepsilon_g = \left(\int_0^\infty dx \int_{-\infty}^0 dy + \int_{-\infty}^0 dx \int_0^\infty dy\right) P(x,y) = \frac{1}{\pi} arccos\left(\frac{R}{\sqrt{Q}}\right)$$

Para simplificar la integral, transformamos a coordenadas polares. En coordenadas polares, $x = r\cos\theta$ y $y = r\sin\theta$, donde $r \geq 0$ y θ está en el rango $[0, 2\pi]$. El diferencial de área en coordenadas polares es:

$$dx\, dy = r\, dr\, d\theta$$

Las regiones donde x y y tienen signos opuestos corresponden a $\pi/2 < \theta < \pi$ y $3\pi/2 < \theta < 2\pi$. Por lo tanto, la integral se puede escribir como:

$$\varepsilon_g = \int_{\pi/2}^{\pi} d\theta \int_0^\infty P(r,\theta)\, r\, dr + \int_{3\pi/2}^{2\pi} d\theta \int_0^\infty P(r,\theta)\, r\, dr$$

En coordenadas polares, la densidad conjunta $P(x,y)$ se transforma utilizando $x = r\cos\theta$ y $y = r\sin\theta$:

$$x^2 + Qy^2 - 2Rxy = r^2(\cos^2\theta + Q\sin^2\theta - 2R\cos\theta\sin\theta)$$

que se puede simplificar a:

$$x^2 + Qy^2 - 2Rxy = r^2\left(1 + (Q-1)\sin^2\theta - 2R\cos\theta\sin\theta\right)$$

La densidad conjunta $P(x,y)$ en coordenadas polares es entonces:

$$P(r,\theta) = \frac{1}{2\pi\sqrt{Q-R^2}} \exp\left[-\frac{r^2}{2}\frac{1+(Q-1)\sin^2\theta - 2R\cos\theta\sin\theta}{Q-R^2}\right]$$

En primer lugar evaluamos la integral radial, esto es

$$\int_0^\infty r\exp\left[-ar^2\right] dr$$

donde $a = \frac{1+(Q-1)\sin^2\theta - 2R\cos\theta\sin\theta}{2(Q-R^2)}$. Esta es una integral gaussiana estándar y se evalúa como:

$$\int_0^\infty r\exp\left[-ar^2\right] dr = \frac{1}{2a}$$

Sustituyendo a tenemos que

$$\int_0^\infty r \exp\left[-\frac{r^2}{2}\frac{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}{Q-R^2}\right]dr = \frac{(Q-R^2)}{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}$$

Ahora evaluamos la integral angular:

$$\varepsilon_g = \frac{1}{2\pi\sqrt{Q-R^2}}\cdot\left(\int_{\pi/2}^{\pi}\frac{Q-R^2}{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}d\theta+\int_{3\pi/2}^{2\pi}\frac{Q-R^2}{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}d\theta\right)$$

Simplificando:

$$\varepsilon_g = \frac{\sqrt{Q-R^2}}{2\pi}\left(\int_{\pi/2}^{\pi}\frac{1}{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}d\theta+\int_{3\pi/2}^{2\pi}\frac{1}{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}d\theta\right)$$

Tenemos que, según cálculo simbólico, esta integral es

$$\int\frac{1}{1+(Q-1)\sin^2\theta-2R\cos\theta\sin\theta}d\theta = \frac{\arctan\left(\frac{Q\tan(\theta)-R}{\sqrt{Q-R^2}}\right)}{\sqrt{Q-R^2}}+C$$

Para el primer intervalo $\left[\frac{\pi}{2},\pi\right]$, tenemos que $\tan(\pi)=0$ y $\tan(\pi/2)$ tiende a ∞, por lo tanto:

$$\arctan\left(\frac{Q\tan(\pi)-R}{\sqrt{Q-R^2}}\right) = \arctan\left(\frac{0-R}{\sqrt{Q-R^2}}\right) = \arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right)$$

y

$$\arctan\left(\frac{Q\tan(\pi/2)-R}{\sqrt{Q-R^2}}\right) \approx \arctan(\infty) = \frac{\pi}{2}$$

Entonces:

$$\left.\frac{\arctan\left(\frac{Q\tan(\theta)-R}{\sqrt{Q-R^2}}\right)}{\sqrt{Q-R^2}}\right|_{\pi/2}^{\pi} = \frac{1}{\sqrt{Q-R^2}}\left(\arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right)-\frac{\pi}{2}\right)$$

Para el segundo intervalo $\left[\frac{3\pi}{2},2\pi\right]$, tenemos que $\tan(2\pi)=0$ y $\tan(3\pi/2)$ tiende a $-\infty$, por lo tanto:

$$\arctan\left(\frac{Q\tan(2\pi)-R}{\sqrt{Q-R^2}}\right) = \arctan\left(\frac{0-R}{\sqrt{Q-R^2}}\right) = \arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right)$$

y

$$\arctan\left(\frac{Q\tan(3\pi/2)-R}{\sqrt{Q-R^2}}\right) \approx \arctan(-\infty) = -\frac{\pi}{2}$$

Entonces:

$$\left.\frac{\arctan\left(\frac{Q\tan(\theta)-R}{\sqrt{Q-R^2}}\right)}{\sqrt{Q-R^2}}\right|_{3\pi/2}^{2\pi} = \frac{1}{\sqrt{Q-R^2}}\left(\arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right) - \left(-\frac{\pi}{2}\right)\right)$$

Sumando ambas contribuciones, obtenemos:

$$\varepsilon_g = \frac{\sqrt{Q-R^2}}{2\pi}\left(\frac{1}{\sqrt{Q-R^2}}\left[\arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right) - \frac{\pi}{2}\right] + \frac{1}{\sqrt{Q-R^2}}\left[\arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right) + \frac{\pi}{2}\right]\right)$$

Simplificando:

$$\varepsilon_g = \frac{\sqrt{Q-R^2}}{2\pi}\cdot\frac{1}{\sqrt{Q-R^2}}\left(2\arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right)\right)$$

$$\varepsilon_g = \frac{1}{\pi}\arctan\left(\frac{-R}{\sqrt{Q-R^2}}\right)$$

Usando la propiedad $\arctan(-x) = -\arctan(x)$, tenemos:

$$\varepsilon_g = \frac{1}{\pi}\left(-\arctan\left(\frac{R}{\sqrt{Q-R^2}}\right)\right)$$

Finalmente, usamos la relación entre arctan, arcsin y arc cos, que dice que

$$\arctan\left(\frac{x}{\sqrt{1-x^2}}\right) = \arcsin x = \frac{\pi}{2} - \arccos x$$

Por lo que podemos escribir que

$$\arctan\left(\frac{R}{\sqrt{Q-R^2}}\right) = \arccos\left(\frac{R}{\sqrt{Q}}\right) - \frac{\pi}{2}$$

Por lo tanto, llegamos a:

$$\varepsilon_g = \frac{1}{\pi}\arccos\left(\frac{R}{\sqrt{Q}}\right) - \frac{1}{2}$$

Es muy probable que haya algún error de cálculo y por eso tenemos ese término $-\frac{1}{2}$, ya que debería salir que

$$\varepsilon_g = \frac{1}{\pi}\arccos\left(\frac{R}{\sqrt{Q}}\right)$$

Problema 7A. Aplicaciones del Algoritmo de Back Propagation en Problemas Reales.

Solución:

Introducción

El algoritmo de Backpropagation (BP) es una técnica ampliamente utilizada en el entrenamiento de redes neuronales artificiales. Su capacidad para ajustar pesos y minimizar errores lo hace ideal para una variedad de aplicaciones prácticas. En este ejercicio, presentaremos un análisis detallado del estado de la aplicación de BP, enfocándonos en el ejemplo de NETtalk, un sistema diseñado para pronunciar texto en inglés. NETtalk es un modelo de red neuronal desarrollado por Terrence Sejnowski y Charles Rosenberg en los años 80. El objetivo principal del proyecto fue construir un sistema que pudiera aprender a pronunciar texto en inglés mediante la presentación de ejemplos de texto y sus transcripciones fonéticas correspondientes.

Arquitectura y entrenamiento de NETtalk

La red NETtalk está compuesta por tres capas: una capa de entrada, una capa oculta y una capa de salida. La red tiene 18,629 pesos ajustables, lo cual era considerable para la época.

- **Entradas**: 203 unidades divididas en 7 grupos de 29 unidades, representando un carácter por unidad en codificación "one-hot".
- **Capa Oculta**: 80 unidades.
- **Salidas**: 26 unidades que codifican características articulatorias de los fonemas y límites de sílabas.

El entrenamiento de NETtalk implicó los siguientes pasos:

1. **Inicialización de Pesos**: Los pesos de la red se inicializan con valores pequeños y aleatorios.
2. **Propagación hacia Adelante**: Los caracteres de entrada se procesan a través de la red, produciendo una salida fonética.
3. **Cálculo del Error**: La salida de la red se compara con la salida deseada.
4. **Retropropagación del Error**: El error se propaga hacia atrás, ajustando los pesos utilizando la regla de gradiente descendente.
5. **Actualización de Pesos**: Los pesos se actualizan para minimizar el error.
6. **Iteración**: Este proceso se repite para cada patrón en el conjunto de entrenamiento hasta que la red esté adecuadamente entrenada.

Tras ser entrenada con un conjunto de datos de 20,000 palabras, NETtalk logró una precisión del 78 % en la pronunciación de nuevas palabras. Este alto nivel de precisión demostró la capacidad de las redes neuronales para generalizar a partir de ejemplos de entrenamiento y aplicar este conocimiento a nuevas situaciones. NETtalk fue un avance significativo en el uso de redes neuronales para el procesamiento del lenguaje natural. Sin embargo, presentaba algunas limitaciones, como la falta de modelado de las etapas de procesamiento de imágenes y reconocimiento de letras en la corteza visual. A pesar de estas limitaciones, NETtalk demostró que es posible aprender asociaciones fonéticas complejas a partir del contexto de letras circundantes.

Problema 7B. Entrenar una red neuronal con el algoritmo de BP. Ver qué ocurre cuando uno aumenta el número de capas ocultas haciendo la red más profunda.

Solución:

Reconocimiento de dígitos con una Red Neuronal

Introducción

El algoritmo de retropropagación del error, conocido comúnmente como Backpropagation, es una técnica fundamental utilizada para entrenar redes neuronales artificiales. Este algoritmo se ha convertido en un componente esencial para ajustar los pesos de las conexiones en una red neuronal multicapa, con el objetivo de minimizar el error en la salida y mejorar el rendimiento del modelo. Una red neuronal típica consta de varias capas: una capa de entrada, una o más capas ocultas y una capa de salida. Cada capa está compuesta por múltiples neuronas, cada una de las cuales aplica una función de activación que introduce no linealidad al modelo, permitiendo que la red aprenda y represente relaciones complejas en los datos. A continuación, explicamos brevemente en qué consiste cada capa:

Capa de Entrada: La capa de entrada es la primera capa de la red neuronal y su función principal es recibir los datos de entrada. Cada neurona en esta capa representa una característica del dato de entrada. Para nuestro caso, en una imagen de dígitos escritos a mano de 28x28 píxeles, habría 784 neuronas en la capa de entrada, una por cada píxel de la imagen. Estas neuronas no realizan ningún procesamiento complejo, simplemente transmiten los datos a la siguiente capa.

Capas Ocultas: Las capas ocultas se sitúan entre la capa de entrada y la capa de salida. Su propósito es procesar las señales recibidas de la capa de entrada a través de múltiples transformaciones no lineales, permitiendo que la red aprenda patrones complejos y representaciones de los datos. Cada neurona en una capa oculta recibe entradas de todas las neuronas de la capa anterior y transmite su salida a todas las neuronas de la siguiente capa. En la red neuronal usada en este trabajo, cada capa oculta contiene 64 neuronas.

Capa de Salida: La capa de salida es la última capa de la red y su función es producir la predicción final. El número de neuronas en la capa de salida corresponde al número de clases en un problema de clasificación o al número de valores que queremos predecir. En el caso de la clasificación de dígitos, hay 10 neuronas en la capa de salida, cada una representando una clase posible (dígitos del 0 al 9).

El proceso de entrenamiento de una red neuronal con Backpropagation comienza con la propagación hacia adelante. En esta fase, se presenta un conjunto de entradas a la capa de entrada, y las señales se propagan a través de las capas de la red. Cada neurona en una capa recibe las entradas, las procesa aplicando una función de activación, y envía la salida a las neuronas de la siguiente capa. Este proceso continúa hasta que las señales alcanzan la capa de salida, generando una predicción, tal y como se ha explicado anteriormente. Una vez obtenida la predicción, se calcula el error comparando la salida de la red con la salida deseada utilizando una función de costo, como el error cuadrático medio o la entropía cruzada. Esta función de costo cuantifica la diferencia entre la predicción de la red y el valor real, proporcionando una medida del rendimiento del modelo.

La fase crucial del algoritmo de Backpropagation es la propagación hacia atrás, cuyo objetivo es ajustar los pesos de las conexiones para minimizar el error. Este ajuste se realiza mediante el método de gradiente descendente, que actualiza los pesos en función del gradiente de la función de

costo con respecto a cada peso. El cálculo del gradiente implica dos pasos importantes: primero, se calcula el gradiente de la función de costo con respecto a los pesos que conectan la capa de salida y la capa oculta; luego, el error se propaga hacia atrás a través de la red, capa por capa, calculando los gradientes para cada peso.

En términos más técnicos, el algoritmo de Backpropagation se puede describir de la siguiente manera:

Sea la función de costo cuadrático de la forma:

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2,$$

donde n es el número total de ejemplos de entrenamiento; la suma se realiza sobre los ejemplos de entrenamiento individuales x; $y = y(x)$ es la salida deseada correspondiente; L denota el número de capas en la red; y $a^L = a^L(x)$ es el vector de activaciones que la red produce cuando se le da la entrada x.

Entonces el algoritmo realiza lo siguiente:

- Se presenta una entrada x y se establece la activación correspondiente a^1 para la capa de entrada.
- Durante la propagación hacia adelante, para cada capa $l = 2, 3, \ldots, L$, se calculan

$$z^l = w^l a^{l-1} + b^l \quad \text{y} \quad a^l = \sigma(z^l)$$

donde σ es la función de activación.

- El error de salida δ^L se calcula como

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

- La retropropagación del error se realiza para cada capa $l = L-1, L-2, \ldots, 2$, calculando

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

- Finalmente, el gradiente de la función de costo se obtiene mediante

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{y} \quad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

El nombre "retropropagación" proviene del hecho de que los vectores de error δ^l se calculan hacia atrás, comenzando desde la capa final. Este enfoque hacia atrás es necesario porque la función de costo depende de las salidas de la red. Para comprender cómo varía el costo con respecto a los pesos y sesgos anteriores, es necesario aplicar repetidamente la regla de la cadena, trabajando hacia atrás a través de las capas para obtener expresiones utilizables. Este proceso de ajuste iterativo es fundamental para el aprendizaje de la red neuronal, permitiendo que la red mejore su rendimiento con cada ciclo de entrenamiento.

Metodología

En esta sección, describimos las herramientas y técnicas utilizadas para el desarrollo del proyecto, incluyendo TensorFlow, Keras y el dataset MNIST. Además, explicamos las funciones utilizadas en el código y algunos conceptos clave como ReLU, Softmax, Adam, Crossentropy y épocas.

TensorFlow, Keras y MNIST

TensorFlow es una biblioteca de código abierto desarrollada por Google para el aprendizaje automático y el deep learning (aprendizaje profundo). Proporciona un entorno flexible y eficiente para el cálculo numérico, utilizando grafos de flujo de datos. Por otro lado, Keras es una API de alto nivel que se ejecuta sobre TensorFlow. Keras simplifica la construcción y el entrenamiento de modelos de deep learning mediante una interfaz intuitiva y fácil de usar. Encontramos conveniente usar Keras ya que en este, el algoritmo de retropropagación no se escribe explícitamente como una función o método separado. En su lugar, está implementado como parte del proceso de entrenamiento cuando utilizamos `model.fit()`. El algoritmo de retropropagación es manejado por las características de diferenciación automática y optimización de TensorFlow.

El dataset MNIST es considerado el "Hello World"de la visión artificial. Contiene un conjunto de entrenamiento de 60,000 imágenes de dígitos manuscritos (de 0 a 9) y otro conjunto de pruebas con 10,000 muestras adicionales. Este dataset es ampliamente utilizado para entrenar y evaluar modelos de reconocimiento de patrones en el campo del deep learning.

Funciones y Conceptos Clave

Las funciones utilizadas en el código son las siguientes:

- `Sequential`: Es una forma de construir modelos en Keras donde las capas se apilan secuencialmente.
- `Dense`: Añade una capa densamente conectada (fully connected).
- `Flatten`: Convierte una entrada de varias dimensiones en una sola dimensión.
- `compile`: Configura el modelo para el entrenamiento. Aquí se especifica el optimizador, la función de pérdida y las métricas de evaluación.
- `fit`: Entrena el modelo usando los datos de entrenamiento. Durante este proceso, la retropropagación se ejecuta automáticamente para ajustar los pesos de la red.
- `evaluate`: Evalúa el rendimiento del modelo en el conjunto de prueba.
- `predict`: Genera predicciones a partir de nuevas entradas.

Otros conceptos claves necesarios para usar estas funciones son:

ReLU (Rectified Linear Unit): Es una función de activación que se define como $\text{ReLU}(x) = \text{máx}(0, x)$. Introduce no linealidad en el modelo y es ampliamente utilizada debido a su simplicidad y eficiencia.

Softmax: Es una función de activación utilizada en la capa de salida para problemas de clasificación multiclase. Convierte las salidas en probabilidades que suman 1, facilitando la interpretación de la predicción.

Adam (Adaptive Moment Estimation): Es un optimizador que combina las ventajas del Gradiente Descendente Estocástico (SGD) con las del algoritmo RMSProp. Adam ajusta dinámicamente la tasa de aprendizaje utilizando momentos de primer y segundo orden. Esto permite un entrenamiento más eficiente y estable. Adam se deriva de dos optimizadores: - **Momentum**: Ayuda a reducir el ruido y acelera el aprendizaje. - **RMSProp**: Ajusta la tasa de aprendizaje dinámicamente para cada parámetro.

Crossentropy (Entropía Cruzada): Es una función de pérdida utilizada para medir la diferencia entre dos distribuciones de probabilidad. En problemas de clasificación, la entropía cruzada es una métrica efectiva para evaluar el rendimiento del modelo.

Épocas: Una época es un ciclo completo a través del conjunto de datos de entrenamiento. Cuando entrenamos un modelo con un conjunto de datos grande, no es eficiente (y a veces ni siquiera es posible) procesar todas las muestras de entrenamiento de una vez. Por lo tanto, los datos se dividen en pequeños lotes o "batches". Así, *batch_ size* determina cuántas muestras hay en cada lote. En cada iteración de entrenamiento, el modelo procesa un lote de datos, calcula la función de pérdida (error) y ajusta los parámetros (pesos y sesgos) del modelo basándose en los gradientes calculados a partir de ese lote. Después de procesar cada lote, los parámetros del modelo se actualizan. Este proceso se repite hasta que todos los lotes del conjunto de datos se han procesado. Un ciclo completo a través de todos los lotes es lo que llamamos una época.

Resultados

En primer lugar, hemos empezado usando funciones de activación sigmoides en las capas ocultas y en la capa de salida, y como optimizador el gradiente descendiente estocástico (SGD). Usando estas funciones de activación y optimizador, hemos obtenido estos resultados de la precisión de la red entrenada durante 100 épocas, como se muestra en la Figura 15.

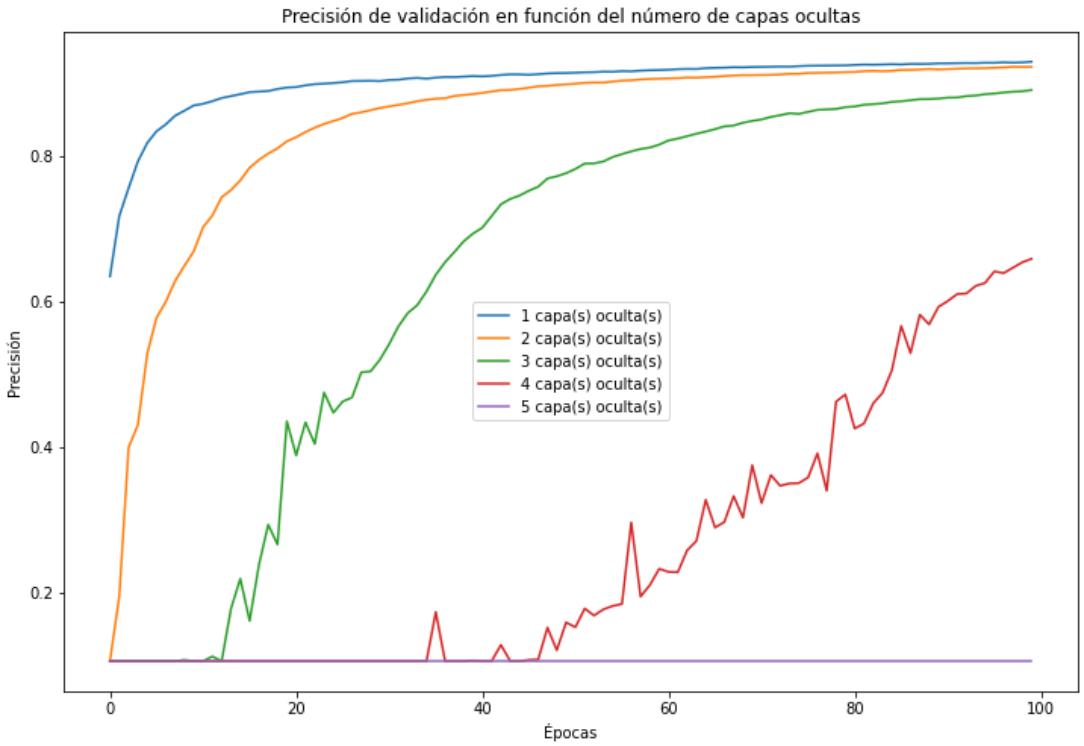


Figura 15: Precisión de la red entrenada con funciones de activación sigmoides y optimizador SGD durante 100 épocas.

La precisión máxima obtenida en el conjunto de prueba para diferentes profundidades de red es la siguiente tabla 1:

Como se puede ver en la Figura 15, a medida que añadimos capas, la precisión disminuye y por cada capa extra necesitamos más épocas para estabilizar la precisión de la red. Como veremos en el ejercicio 8, esto se debe al problema de la desaparición del gradiente (*vanishing gradient problem*).

Número de Capas Ocultas	**Precisión Máxima**
1 capa(s) oculta(s)	0.9281
2 capa(s) oculta(s)	0.9183
3 capa(s) oculta(s)	0.8892
4 capa(s) oculta(s)	0.6448
5 capa(s) oculta(s)	0.1135

Cuadro 1: Precisión máxima en el conjunto de prueba usando funciones de activación sigmoides y optimizador SGD.

Este problema se puede mitigar si usamos funciones de activación más adecuadas y un optimizador más potente.

Para ello, hemos usado la función ReLU en las capas ocultas, la función softmax en las capas de salida y el optimizador Adam. Una vez hechas estas modificaciones, hemos estudiado la precisión del modelo en función del número de capas ocultas durante 10 épocas (10 veces menos que antes), para diferentes profundidades de red: $N = 1, 5, 10, 15, 25, 35, 45$. Los resultados obtenidos se muestran en la Figura 16.

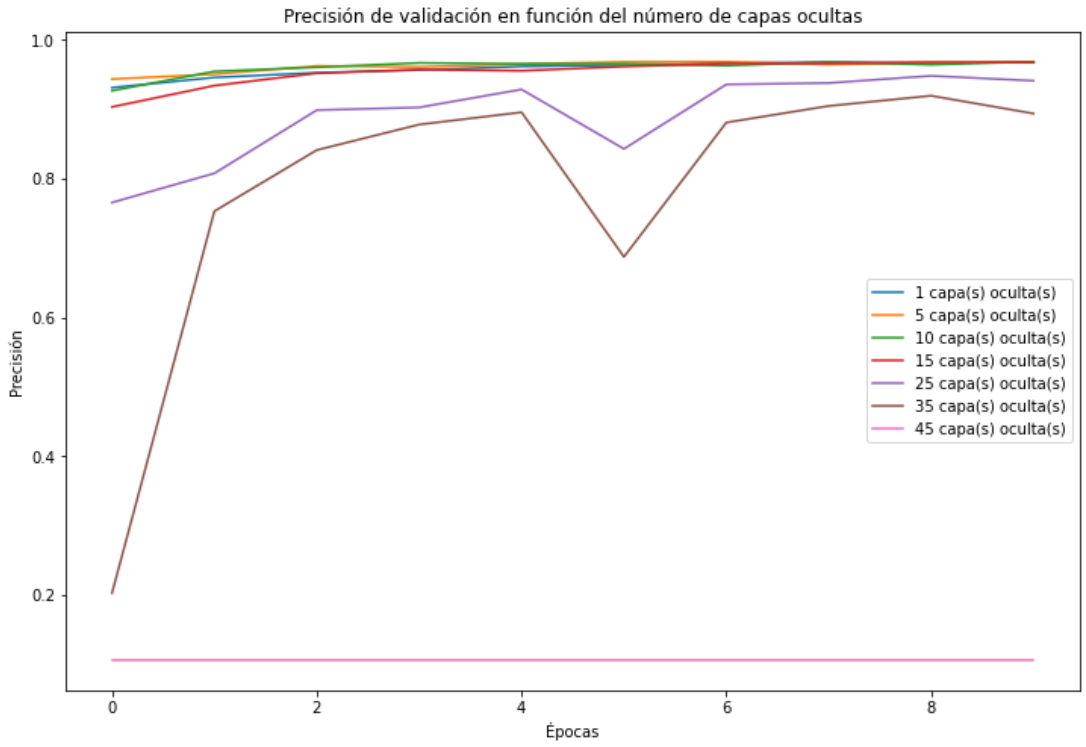


Figura 16: Precisión de la red entrenada con funciones de activación ReLU y softmax y optimizador Adam durante 10 épocas.

La precisión máxima obtenida en el conjunto de prueba para diferentes profundidades de red es la siguiente (2):

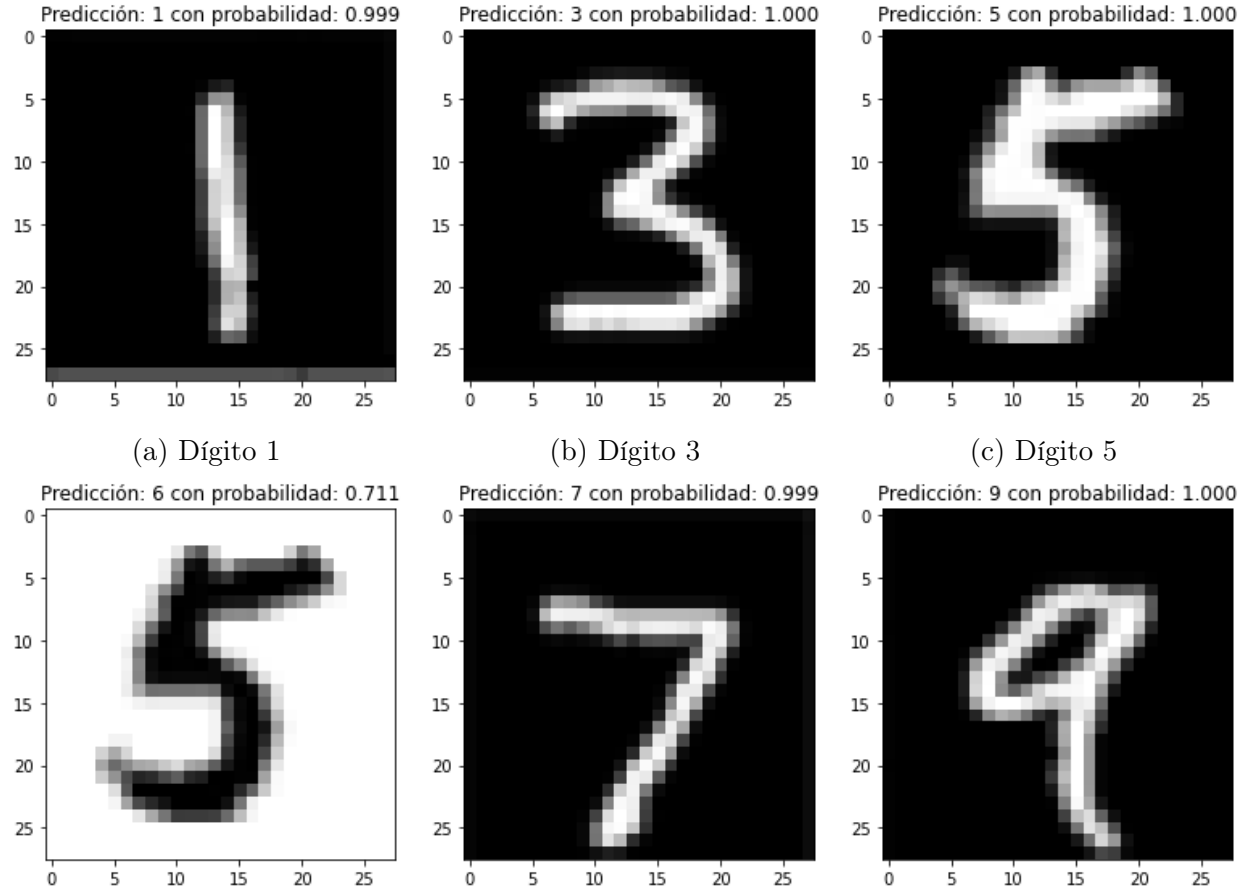
Como vemos en estos resultados, al aplicar estas modificaciones hemos conseguido mitigar el problema, además de mejorar significativamente la precisión de la red. No obstante, también se observa que la precisión decae con el aumento de la cantidad de capas ocultas, aunque mucho más lentamente que con la configuración inicial de la red con funciones de activación sigmoides, pudiendo llegar a tener una red con 25 capas ocultas y una precisión del 94 %, una precisión más que aceptable

Número de Capas Ocultas	**Precisión Máxima**
1 capa(s) oculta(s)	0.9710
5 capa(s) oculta(s)	0.9700
10 capa(s) oculta(s)	0.9696
15 capa(s) oculta(s)	0.9688
25 capa(s) oculta(s)	0.9425
35 capa(s) oculta(s)	0.8946
45 capa(s) oculta(s)	0.1135

Cuadro 2: Precisión máxima en el conjunto de prueba usando funciones de activación ReLU y softmax y optimizador Adam.

(e incluso más alta que las precisiones obtenidas con las funciones sigmoides). Además, usar Adam, ReLU y softmax ha disminuido el tiempo de entrenamiento ya que tan solo hemos necesitado 10 épocas para obtener estos resultados.

Ya que tenemos la red neuronal entrenada, vamos a ver cómo se comporta cuando le mostramos distintos dígitos escritos a mano para un modelo con 3 capas ocultas y otro con 35 capas ocultas. Para ponérselo un poco más difícil, hemos elegido dígitos que en principio tienen cierto parecido entre sí: el 1, 7 y 9, y el 3 y 5. Además de la predicción del modelo, hemos estudiado con qué probabilidad hace la predicción. Para el modelo con 3 capas obtenemos los siguientes resultados, como se muestra en la Figura 17.



(a) Dígito 1 (b) Dígito 3 (c) Dígito 5

(d) Dígito 5 fondo blanco (e) Dígito 7 (f) Dígito 9

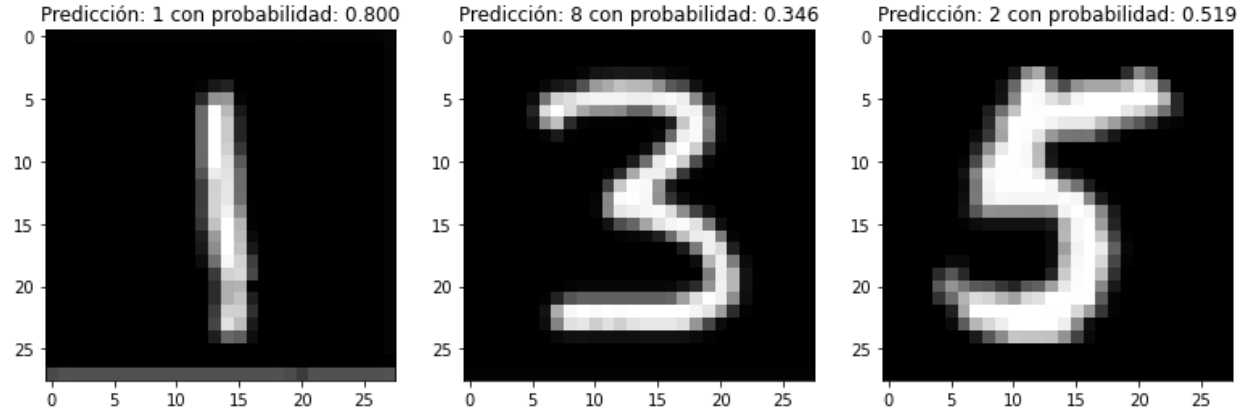
Figura 17: Predicciones del modelo con 3 capas ocultas para los dígitos 1, 3, 5, 7 y 9.

Como vemos, el modelo con 3 capas ocultas acierta en todos los casos con la predicción y además lo hace con una probabilidad cercana al 100 %. Como anécdota, comentar que el modelo únicamente reconoce los dígitos cuando son blancos con un fondo negro. Cuando los dígitos son negros con el fondo blanco, el modelo falla en la predicción, como se muestra en la Figura 17.

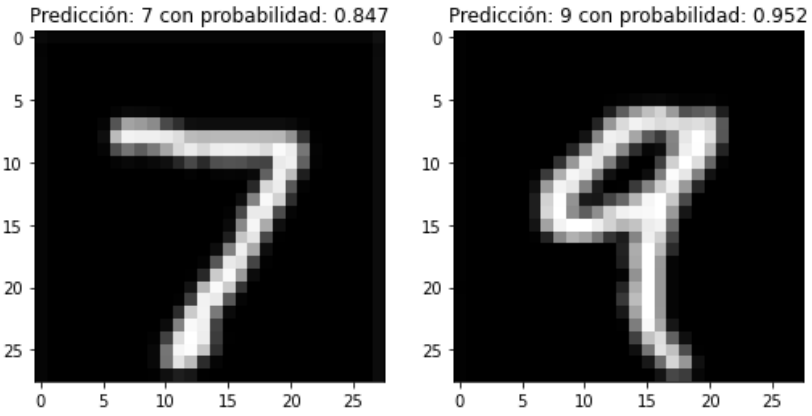
Para el modelo con 35 capas ocultas obtenemos los siguientes resultados, como se muestra en la Figura 18. Como vemos, este modelo falla en 2 de las 5 predicciones, y además, las que acierta, las probabilidades son algo más bajas que las del modelo con 3 capas ocultas.

En resumen, aunque un modelo con más capas ocultas tiene la capacidad teórica de capturar patrones más complejos, en la práctica parece sufrir de problemas como el sobreajuste y la inestabilidad del entrenamiento. En nuestro caso, el modelo con 3 capas ocultas no sólo logró una precisión alta, sino que también mostró una mayor probabilidad en sus predicciones, mientras que el modelo con 35 capas ocultas tuvo un rendimiento inferior y menor probabilidad en las predicciones correctas.

Estos resultados parecen extraños. Intuitivamente, se esperaría que al agregar capas ocultas adicionales se permitiría a la red aprender funciones de clasificación más complejas, mejorando así su desempeño. En teoría, no debería empeorar, ya que las capas adicionales, en el peor de los casos, podrían no hacer nada. Sin embargo, eso no es lo que observamos aquí. En el siguiente ejercicio, explicaremos qué está ocurriendo realmente, haciendo un análisis crítico sobre las dificultades de entrenamiento de las redes profundas y cómo afecta la velocidad de aprendizaje al aumentar la profundidad de la red.



(a) Dígito 1 (b) Dígito 3 (c) Dígito 5



(d) Dígito 7 (e) Dígito 9

Figura 18: Predicciones del modelo con 35 capas ocultas para los dígitos 1, 3, 5, 7 y 9.

Problema 8. Análisis crítico sobre las dificultades de entrenamiento de las redes Deep y qué ocurre con la velocidad de aprendizaje aumentado la red.

Solución:

Como vimos en el ejercicio anterior, al aumentar la cantidad de capas ocultas en una red neuronal que utiliza funciones de activación sigmoides, la precisión de la red entrenada disminuía. Estos resultados parecen extraños. Intuitivamente, se esperaría que al agregar capas ocultas adicionales se permitiría a la red aprender funciones de clasificación más complejas, mejorando así su desempeño. En teoría, no debería empeorar, ya que las capas adicionales, en el peor de los casos, podrían no hacer nada. Sin embargo, eso no es lo que observamos aquí. En este ejercicio, intentaremos explicar qué está ocurriendo realmente, haciendo un análisis crítico sobre las dificultades de entrenamiento de las redes profundas y cómo afecta la velocidad de aprendizaje al aumentar la profundidad de la red. Veamos qué está ocurriendo.

Supongamos que las capas ocultas adicionales podrían ayudar en principio, y que el problema radica en que nuestro algoritmo de aprendizaje no está encontrando los pesos y sesgos correctos. Queremos descubrir qué está fallando en nuestro algoritmo de aprendizaje y cómo mejorarlo. Para ello, denotemos el gradiente como $\delta^l_j = \frac{\partial C}{\partial b^l_j}$, es decir, el gradiente para la j-ésima neurona en la l-ésima capa. Podemos pensar en el gradiente δ^1 como un vector cuyas entradas determinan qué tan rápido aprende la primera capa oculta, y δ^2 como un vector cuyas entradas determinan qué tan rápido aprende la segunda capa oculta. Utilizaremos entonces las longitudes de estos vectores como

medidas globales (aproximadas) de la velocidad a la que aprenden las capas. Así, la longitud $\|\delta^l\|$ mide la velocidad a la que está aprendiendo la l-ésima capa oculta.

Modificando ligeramente el código usado en el ejercicio anterior, hemos calculado la velocidad de aprendizaje de cada capa de la red neuronal para la configuración del ejercicio anterior en la que usamos funciones de activación sigmoides, 10 capas ocultas, con 30 épocas, obteniendo los siguientes resultados mostrados en la Figura 19. Como vemos, las últimas capas aprenden mucho más rápido que las primeras.

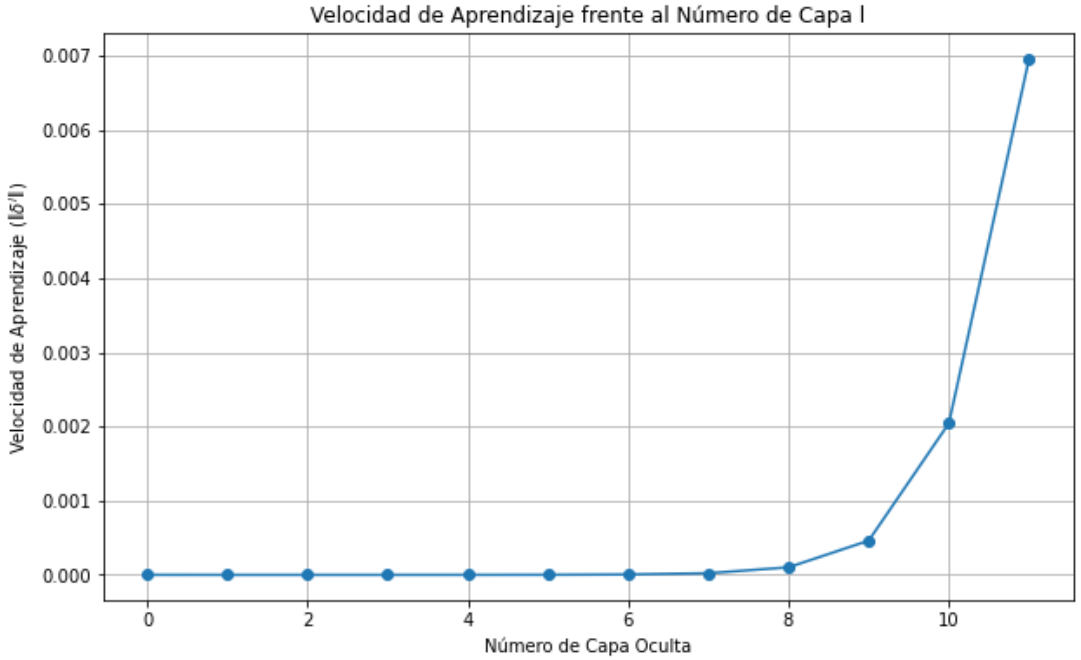


Figura 19: Velocidad de Aprendizaje frente al Número de Capas Ocultas. Los índices 0 y 11 se corresponden con las capas de entrada y de salida, respectivamente.

Aquí podemos hacer una observación importante: en al menos algunas redes neuronales profundas, el gradiente tiende a disminuir a medida que retrocedemos a través de las capas ocultas. Esto significa que las neuronas en las capas iniciales aprenden mucho más lentamente que las neuronas en las capas posteriores. Aunque hemos observado esto en una sola red, hay razones fundamentales por las cuales esto ocurre en muchas redes neuronales. Este fenómeno es conocido como el problema de la desaparición del gradiente.

Problema de la Desaparición del Gradiente

El problema de la desaparición del gradiente fue identificado en los primeros años del desarrollo de redes neuronales y es especialmente prevalente en redes que utilizan funciones de activación como la sigmoide o la tangente hiperbólica (tanh). Estas funciones comprimen su entrada a un rango pequeño de valores (0 a 1 para sigmoide y -1 a 1 para tanh), lo que puede resultar en derivadas muy pequeñas. Durante el proceso de retropropagación, estas pequeñas derivadas se multiplican repetidamente a través de las capas, lo que puede llevar a gradientes extremadamente pequeños y, por lo tanto, a una actualización de pesos insignificante.

Más generalmente, el gradiente en redes neuronales profundas es inestable, tendiendo a explotar o desaparecer en las capas iniciales. Esta inestabilidad es un problema fundamental para el aprendizaje basado en gradientes en redes neuronales profundas.

Problema del Gradiente Explosivo

El problema del gradiente explosivo es el fenómeno opuesto, donde los gradientes pueden crecer exponencialmente durante la retropropagación, lo que resulta en actualizaciones de pesos extremadamente grandes y, por lo tanto, en oscilaciones y divergencias durante el entrenamiento. En resumen, el verdadero problema aquí es que las redes neuronales sufren de un problema de gradiente inestable. Como resultado, si usamos técnicas estándar de aprendizaje basadas en gradientes, las diferentes capas de la red tenderán a aprender a velocidades extremadamente diferentes.

Soluciones al Problema del gradiente inestable

Para mitigar el problema de la desaparición del gradiente, se han desarrollado varias técnicas y arquitecturas:

- **Funciones de Activación ReLU:** La función de activación *Rectified Linear Unit* (ReLU) ayuda a superar el problema de la desaparición del gradiente al evitar la saturación del gradiente. ReLU reemplaza valores negativos con cero, asegurando que los gradientes que fluyen hacia atrás permanezcan no nulos y no desaparezcan. Esto promueve un mejor flujo del gradiente y permite un aprendizaje efectivo en redes neuronales profundas. Como vimos en el ejercicio anterior, usar este tipo de funciones de activación nos permitió tener redes más profundas que además daban muy buenos resultados.
- **Inicialización Adecuada de Pesos:** La inicialización adecuada de los pesos puede ayudar a mantener los gradientes en un rango razonable. Métodos como la inicialización de He o la inicialización de Xavier (o Glorot) están diseñados para mantener la varianza de los activaciones relativamente constante a través de las capas.
- **Redes Residuales:** Las Redes Residuales (ResNets) son un tipo popular de red neuronal que ha mostrado un rendimiento notable en el deep learning. La arquitectura de las ResNets introduce conexiones de salto, donde la salida de una capa se suma a la entrada de otra capa, facilitando un mejor flujo de gradientes y evitando el problema de la desaparición del gradiente. Las conexiones de salto también permiten que la red aprenda funciones residuales (o de error residual), lo que facilita el entrenamiento de modelos más profundos. Estas ResNets se han utilizado para lograr un buen rendimiento en distintas aplicaciones, como el reconocimiento de imágenes, la detección de objetos y el procesamiento del lenguaje natural. Las conexiones de salto en las ResNets han demostrado aliviar eficazmente el problema de la desaparición del gradiente y hacer que el entrenamiento sea más fácil.

En conclusión, el problema de la desaparición del gradiente es un desafío significativo en el entrenamiento de redes neuronales profundas. Sin embargo, mediante el uso de técnicas adecuadas como funciones de activación ReLU, inicialización de pesos o usar redes residuales, es posible mitigar este problema y entrenar redes profundas de manera efectiva. De hecho, los gradientes inestables son solo un obstáculo para el deep learning, aunque es un obstáculo fundamental importante. Gran parte de la investigación en curso tiene como objetivo comprender mejor los desafíos que pueden surgir al entrenar redes profundas.

Problema 9. Usando datos abiertos de redes en Internet y software especifico propio o abierto, elegir y analizar una red compleja.

Solución:

Para realizar este ejercicio, en primer lugar hemos buscado una fuente de datos pública. La fuente seleccionada ha sido Network Repository. Ahí hemos navegado en el repositorio de "Brain Networks" y hemos buscado un conjunto de datos con un tamaño de nodos y conexiones adecuado. Por este motivo, hemos escogido los datos del cortex visual del ratón disponibles en: MOUSE-VISUAL-CORTEX-1.

Con estos datos, hemos desarrollado un código en Python utilizando la librería `networkx` para representar y analizar la red compleja. A continuación se presenta el código utilizado:

```
import networkx as nx
import matplotlib.pyplot as plt

file_path = 'bn-mouse_visual-cortex_1.edges'

# Leer el archivo de red
G = nx.read_edgelist(file_path, comments='#', delimiter=' ', create_using=nx.Graph())

# Información básica de la red
num_nodes = G.number_of_nodes()
num_edges = G.number_of_edges()
print(f'Número de nodos: {num_nodes}')
print(f'Número de aristas: {num_edges}')

# Análisis básico de la red
degree_centrality = nx.degree_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)

# Mostrar los primeros 5 nodos con mayor centralidad de grado
sorted_degree = sorted(degree_centrality.items(), key=lambda x: x[1], reverse=True)
print("Top 5 nodos con mayor centralidad de grado:")
for node, centrality in sorted_degree[:5]:
    print(f"Nodo: {node}, Centralidad de grado: {centrality:.4f}")

# Tamaño de los nodos basado en el grado
node_size = [5000 * nx.degree(G, node) / max(degree_centrality.values()) for node in G.nod

# Visualización de la red
plt.figure(figsize=(12, 12))
pos = nx.spring_layout(G, seed=42)
nx.draw_networkx_nodes(G, pos, node_size=node_size)
nx.draw_networkx_edges(G, pos, alpha=0.5)
plt.title("Red del cortex visual de ratón con tamaño de nodos según su grado")
plt.show()
```

La figura 20 muestra la red compleja del cortex visual del ratón. En esta representación, los nodos

con más conexiones se dibujan más grandes, lo que permite identificar visualmente los nodos más importantes en términos de centralidad de grado.

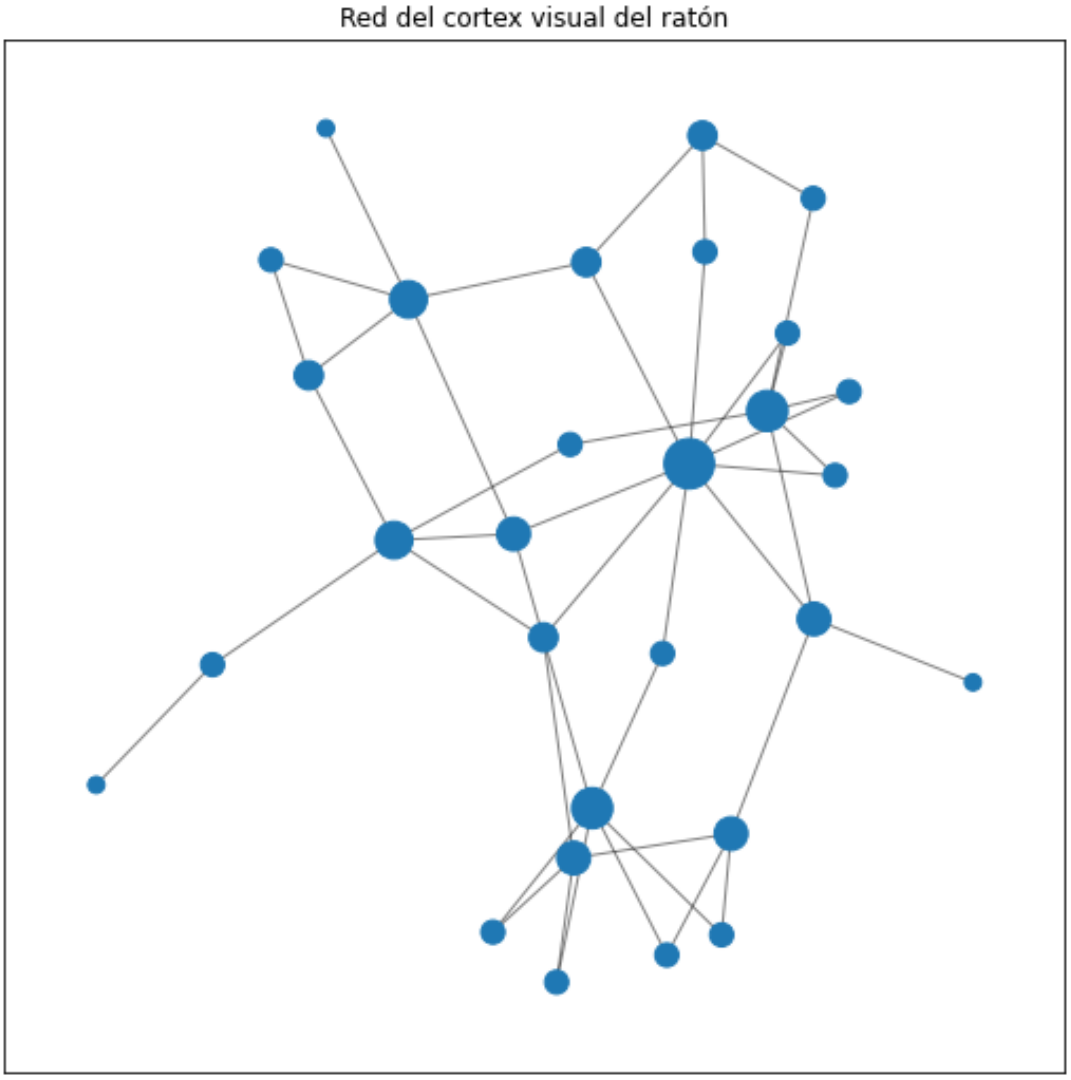


Figura 20: Red del cortex visual del ratón con tamaño de nodos según su grado.

La visualización de la red compleja del cortex visual del ratón nos permite hacernos una idea de cómo es la estructura y las propiedades que podría tener esta red biológica. Los nodos más grandes indican las neuronas con más conexiones, lo cual podría ser de interés para estudios neurocientíficos.

Anexo: Códigos Usados en los Ejercicios

Ejercicio 3

```
import numpy as np
import matplotlib.pyplot as plt

# Función para mostrar la imagen
def plot_image(pattern, title, size=(15, 15)):
    plt.imshow(pattern.reshape(size), cmap='binary')
    plt.title(title)
    plt.show()

# Representación de la letra A en una matriz 15x15
A = np.array([
    [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0]
])
A = A.reshape(225)
A[A == 0] = -1  # Cambiar 0 a -1 para la red de Hopfield

# Inicializar la matriz de pesos W
n_neurons = len(A)
W = np.zeros((n_neurons, n_neurons))

# Aplicar la regla de Hebb para entrenar la red
for i in range(n_neurons):
    for j in range(n_neurons):
        if i != j:
            W[i, j] = A[i] * A[j]

# Función para actualizar la red de manera asincrónica
def update_network_async(state, W):
    for _ in range(len(state)):
        i = np.random.randint(0, len(state))
```

```
        net_input = np.dot(W[i, :], state)
        state[i] = np.sign(net_input)
    return state

# Generar una imagen ruidosa completamente aleatoria
noisy_A = np.random.choice([-1, 1], size=n_neurons)

# Mostrar la imagen ruidosa
plot_image(noisy_A, "Imagen Ruidosa", size=(15, 15))

# Simulación de la red
state = noisy_A
iterations = 20  # Número total de iteraciones
for i in range(iterations):
    state = update_network_async(state, W)
    if i % 2 == 0 or i == iterations - 1:  # Mostrar imagen cada 2 iteraciones y al final
        plot_image(state, f"Iteración {i+1}", size=(15, 15))
```

Ejercicio 4 OPT

```
import numpy as np
import matplotlib.pyplot as plt

# Función para mostrar la imagen
def plot_image(pattern, title, size=(15, 15)):
    plt.imshow(pattern.reshape(size), cmap='binary')
    plt.title(title)
    plt.show()

# Representación de la letra A en una matriz 15x15
A = np.array([
    [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0]
])
```

```
A = A.reshape(225)
A[A == 0] = -1  # Cambiar 0 a -1 para la red de Hopfield

# Inicializar la matriz de pesos W
n_neurons = len(A)
W = np.zeros((n_neurons, n_neurons))

# Aplicar la regla de Hebb para entrenar la red
for i in range(n_neurons):
    for j in range(n_neurons):
        if i != j:
            W[i, j] = A[i] * A[j]

# Función para introducir dilución en la red
def dilute_network(W, dilution_rate):
    mask = np.random.rand(*W.shape) > dilution_rate
    W_diluted = W * mask
    return W_diluted

# Diluir la red con un % de las conexiones eliminadas
dilution_rate = 0.98
W_diluted = dilute_network(W, dilution_rate)

# Función para actualizar la red de manera asincrónica
def update_network_async(state, W):
    for _ in range(len(state)):
        i = np.random.randint(0, len(state))
        net_input = np.dot(W[i, :], state)
        state[i] = np.sign(net_input)
    return state

# Generar una imagen ruidosa completamente aleatoria
noisy_A = np.random.choice([-1, 1], size=n_neurons)

# Mostrar la imagen ruidosa
plot_image(noisy_A, "Imagen Ruidosa", size=(15, 15))

# Simulación de la red diluida
state = noisy_A
iterations = 20  # Número total de iteraciones
for i in range(iterations):
    state = update_network_async(state, W_diluted)
    if i % 2 == 0 or i == iterations - 1:  # Mostrar imagen cada 2 iteraciones y al final
        plot_image(state, f"Iteración {i+1}", size=(15, 15))
```

Ejercicio 7B

Código para Entrenar la Red y calcular la precisión

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Cargar el conjunto de datos MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocesar los datos
x_train = x_train.reshape((x_train.shape[0], 28 * 28)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28 * 28)).astype('float32') / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Función para construir y entrenar la red neuronal
def build_and_train_model(layers):
    model = Sequential()
    model.add(Flatten(input_shape=(28 * 28,)))

    for _ in range(layers):
        model.add(Dense(64, activation='relu')) #cambiar por sigmoid

    model.add(Dense(10, activation='softmax')) #cambiar por sigmoid

    # Compilación del modelo (aquí se define el optimizador y la función de pérdida)
    model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy']) #cambiar adam por sgd

    # Entrenamiento del modelo (aquí se realiza la retropropagación automáticamente)
    history = model.fit(x_train, y_train, epochs=10, batch_size=128,
    validation_split=0.2, verbose=1)

    # Evaluación del modelo en el conjunto de prueba
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)

    return history, test_acc

# Entrenar y evaluar modelos con diferentes capas ocultas
layers_list = [1,5,10,15,25, 35,45]
histories = []
test_accuracies = []

for layers in layers_list:
```

```
    print(f"Entrenando modelo con {layers} capa(s) oculta(s)...")
    history, test_acc = build_and_train_model(layers)
    histories.append(history)
    test_accuracies.append(test_acc)

# Visualizar los resultados
plt.figure(figsize=(12, 8))
for i, history in enumerate(histories):
    plt.plot(history.history['val_accuracy'], label=f'{layers_list[i]} capa(s) oculta(s)')
plt.title('Precisión de validación en función del número de capas ocultas')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.show()

print("Precisión en el conjunto de prueba para diferentes profundidades de red:")
for i, acc in enumerate(test_accuracies):
    print(f"{layers_list[i]} capa(s) oculta(s): {acc:.4f}")
```

Código para Entrenar la Red y generar modelo

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Cargar el conjunto de datos MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocesar los datos
x_train = x_train.reshape((x_train.shape[0], 28 * 28)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28 * 28)).astype('float32') / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Función para construir y entrenar la red neuronal
def build_and_train_model(layers):
    model = Sequential()
    model.add(Flatten(input_shape=(28 * 28,)))

    for _ in range(layers):
        model.add(Dense(64, activation='relu'))

    model.add(Dense(10, activation='softmax'))

    # Compilación del modelo (aquí se define el optimizador y la función de pérdida)
    model.compile(optimizer='adam', loss='categorical_crossentropy',
```

```
    metrics=['accuracy'])

    # Entrenamiento del modelo (aquí se realiza la retropropagación automáticamente)
    model.fit(x_train, y_train, epochs=10, batch_size=128,
    validation_split=0.2, verbose=1)

    # Guardar el modelo entrenado
    model.save('mnist_model_35layers.keras')

    # Evaluación del modelo en el conjunto de prueba
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
    print(f"Precisión en el conjunto de prueba: {test_acc:.4f}")

    return model

# Entrenar y guardar el modelo con una configuración de capas adecuada
model = build_and_train_model(layers=35)
```

Código para cargar el modelo y hacer predicción

```
import tensorflow as tf
from tensorflow.keras.models import load_model
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Cargar el modelo guardado
model = load_model('mnist_model_35layers.keras')

# Función para preprocesar la imagen de entrada
def preprocess_image(image_path):
    img = Image.open(image_path).convert('L')  # Convertir a escala de grises
    img = img.resize((28, 28))  # Redimensionar a 28x28 píxeles
    img = np.array(img).astype('float32') / 255  # Normalizar los valores de los píxeles
    img = img.reshape(1, 28 * 28)  # Aplanar la imagen a un vector de 784 elementos
    return img

# Cargar y preprocesar la imagen de entrada
image_path = '9.jpg'  # Ruta a la imagen que deseas predecir
input_image = preprocess_image(image_path)

# Hacer una predicción
predictions = model.predict(input_image)
predicted_class = np.argmax(predictions)
predicted_probability = np.max(predictions)

# Mostrar la imagen y la predicción con probabilidad
plt.imshow(input_image.reshape(28, 28), cmap='gray')
```

```
plt.title(f'Predicción: {predicted_class} con probabilidad: {predicted_probability:.3f}')
plt.show()
```

Código para estimar la velocidad de la red

```
    import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np

# Cargar el conjunto de datos MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocesar los datos
x_train = x_train.reshape((x_train.shape[0], 28 * 28)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28 * 28)).astype('float32') / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Función para construir y entrenar la red neuronal
def build_and_train_model(layers, epochs=50):
    model = Sequential()
    model.add(Flatten(input_shape=(28 * 28,)))

    for _ in range(layers):
        model.add(Dense(64 , activation='relu'))

    model.add(Dense(10, activation='sigmoid'))

    # Compilación del modelo (aquí se define el optimizador y la función de pérdida)
    model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])

    # Callback para keras para capturar los gradientes
    class GradientCallback(tf.keras.callbacks.Callback):
        def __init__(self):
            super().__init__()
            self.gradients = []

        def on_epoch_end(self, epoch, logs=None):
            # Obtener los gradientes de las capas ocultas
            with tf.GradientTape() as tape:
                y_pred = model(x_train, training=True)
                loss = model.compiled_loss(y_train, y_pred)
            grads = tape.gradient(loss, model.trainable_weights)
            grads = [grads[0]] + grads[1:-1:2] + [grads[-1]]
            # Calcular la norma de los gradientes de cada capa
```

```
            grads_norms = [np.linalg.norm(g.numpy()) for g in grads]
            self.gradients.append(grads_norms)

    gradient_callback = GradientCallback()

    # Entrenamiento del modelo (aquí se realiza la retropropagación automáticamente)
    model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2, verbo

    return model, gradient_callback.gradients

# Entrenar el modelo con una configuración de capas y obtener los gradientes
layers = 10  # Número de capas ocultas
epochs = 30  # Número de épocas
model, gradients = build_and_train_model(layers=layers, epochs=epochs)



# Promediar los gradientes de cada capa oculta a través de las épocas
mean_gradients = np.mean(gradients, axis=0)

# Graficar la velocidad de aprendizaje frente al número de capa oculta
plt.figure(figsize=(10, 6))
plt.plot(range(0, layers+2), mean_gradients, marker='o')
plt.xlabel('Número de Capa Oculta')
plt.ylabel(r'Velocidad de Aprendizaje ($\left \| \delta ^l \right \|$)')
plt.title('Velocidad de Aprendizaje frente al Número de Capa l')
plt.grid(True)
plt.show()
```

Referencias

[1] H. Eugene Stanley. *Introduction to Phase Transitions and Critical Phenomena.* Oxford Science Publications, 1987.

[2] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* The MIT Press, 2005.

[3] David S. Touretzky, Mark V. Albert, Nathaniel D. Daw, Alok Ladsariya, and Mahtiyar Bonakdarpour. Hhsim: Hodgkin-huxley simulator. `http://www.cs.cmu.edu/~dst/HHsim/`, 2024.

[4] Elka Radoslavova Koroutcheva. Tutoriales y apuntes de la asignatura redes neuronales y complejas, 2024. Material de curso.

[5] John Hertz, Anders Krogh, and Richard Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.

[6] Michael A. Nielsen. Neural networks and deep learning. `http://neuralnetworksanddeeplearning.com/index.html`, 2024.

[7] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1(1):145–168, 1987.

[8] Wikipedia. Nettalk (artificial neural network). `https://en.wikipedia.org/wiki/NETtalk_(artificial_neural_network)`.

[9] Interactive Chaos. El dataset mnist. `https://interactivechaos.com/es/manual/tutorial-de-deep-learning/el-dataset-mnist`.

[10] Abhishek Jain. Story about optimizers: From gradient descent to adam optimizer. `https://medium.com/@abhishekjainindore24/story-about-optimizers-from-gradient-descent-to-adam-optimizer-94804f5a0614`.

[11] Keras Documentation. Mnist dataset. `https://keras.io/api/datasets/mnist/`.

[12] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.