Universidad Nacional Autónoma de México

Facultad de Ciencias

Análisis de robustez de una red neuronal bioinspirada

T E S I S

que para obtener el título de:

matemático

P R E S E N T A :

Ian Xul Belaustegui

TUTOR

Alessio Franci

Ciudad Universitaria, CDMX, 2022

«We think in generalities, but we live in details.»

Alfred North Whitehead

Índice general

Introducción

Las redes neuronales artificiales (Artifitial Neural Networks, ANN) son el actual paradigma del aprendizaje de máquinas y la inteligencia artificial. A pesar de originalmente haber tomado inspiración de las redes neuronales de los sistemas nerviosos animales, su desarrollo formal requirió de una serie de enormes simplificaciones y supuestos adicionales (muchas veces sin contraparte biológica clara). A pesar de esta gran simplificación, estos modelos computacionales han mostrado un gran desempeño y flexibilidad en una gran variedad de tareas y problemas con los que metodologías previas tenían problemas. Estos modelos han encontrado aplicaciones en problemas de reconocimiento visual, procesamiento de lenguaje natural, manejo autónomo de máquinas, juegos de mesa y videojuegos, etc [1–3]. Debido a su buen desempeño en diversas tareas, algunas personas se sienten inclinadas a pensar que estos modelos, por más simplificados que sean, de hecho capturan aspectos importantes del funcionamiento cerebral, incluso planteando la hipótesis de que algunos aspectos de las ANN que no están inspiradas en el funcionamiento cerebral deben existir como mecanismos cerebrales, pues si son mecanismos suficientemente eficientes seguramente han surgido evolutivamente (por ejemplo el entrenamiento de pesos sinápticos por *backpropagation* [4]). Es decir, si las ANN realmente modelan de buena forma el funcionamiento cerebral, entonces resultados de la primera pueden inspirar investigación en la segunda (y *viceversa*).

A pesar del gran éxito de estos modelos, las ANN han demostrado tener algunos problemas que sugieren que estas no capturan aspectos importantes del funcionamiento cerebral. Entre estos, un problema muy estudiado recientemente, principalmente por sus implicaciones en la seguridad de implementaciones públicas de ANN, es el problema de los ejemplos adversarios. Esto es, que para todas las implementaciones más eficientes de ANN, casi siempre es posible perturbar la señal de entrada de la red mínimamente (casi imperceptible para un humano) de tal forma que la red súbitamente se muestre completamente confundida y realice mal su tarea. Más allá de todas las implicaciones económicas y de seguridad que esto podría tener en la sociedad (la existencia de ejemplos adversarios implica que es posible engañar a muchos sistemas basados en ANN) [5]; los ejemplos adversarios sacan a la luz un aspecto de las ANN que parece ser fundamentalmente diferente de su contraparte biológica, pues los cerebros parecen ser muy robustos a ruido y pequeñas deformaciones de los estímulos de entrada. Nos sería difícil imaginar una deformación de la imagen clara de algún objeto dado (por ejemplo un venado) de manera que nos llevara a pensar con mucha seguridad que es otro objeto completamente diferente (por ejemplo un

avión) sin que esta deformación fuera bastante dramática. De hecho parecería que por nuestra misma concepción de los objetos, entre más ajenos nos parecen dos de estos, más difícil nos parece que debería ser convertir uno en otro. Sin embargo esto es muy diferente a lo que encontramos en ANN, donde una red cuya tarea es clasificar imágenes puede clasificar correctamente y con gran seguridad una imagen como un venado, y la modificación cuidadosa de un único pixel puede llevarla a clasificar la (casi) misma imagen como un avión con seguridad alta [6]. El anterior es solo un ejemplo de un fenómeno mucho más general, y actualmente existe una gran diversidad de métodos para generar ejemplos adversarios para diferentes tipos de ANN, con diferentes formas de definir una perturbación pequeña. También existen algunas supuestas "defensas" contra los ataques adversarios, aunque estas siempre sacrifican desempeño por robustez [7].

En el contexto de lo anterior uno podría preguntarse qué es lo que falta en las ANN para que tengan la favorable propiedad de la robustez. Aquí se podría proponer una gran cantidad de hipótesis, tomando inspiración de la neurobiología uno podría ir complejizando el modelo hasta que tenga la propiedad deseada, partiendo del supuesto de que el procesamiento cerebral la tiene. Esto se podría hacer en muchas direcciones. En este trabajo exploramos un modelo propuesto por Krotov y Hopfield [8], donde lo que se modifica es la forma en la que las ANN aprenden a partir de experiencia. Este trabajo consiste primero en una breve revisión y comparación de las redes neuronales biológicas y las redes neuronales artificiales en el capítulo 1, con un enfoque sobre las formas de aprendizaje en ambas. En el capítulo 2 se define el modelo de Krotov y Hopfield así como algunos antecedentes importantes. Este modelo se implementó de nuevo a partir de la descripción del artículo y se reprodujeron algunos resultados importantes, y los resultados se presentan en este capítulo. Finalmente el capítulo 3 trata sobre los ejemplos adversarios, su definición formal, posibles explicaciones teóricas, diferentes métodos para generarlos, así como los resultados de algunos experimentos computacionales para poner a prueba la robustez de una red entrenada con el método de Krotov y Hopfield. En este último se encontró que la red entrenada con el método alternativo es notablemente más robustas a perturbaciones adversarias generadas con el método Fast Gradient Sign Method (FGSM) y Projected Gradient Descent (PGD). También se describe que en mi implementación de la arquitectura propuesta por Krotov y Hopfield, la cual usa una capa con un tipo de activación llamado Rectified Polynomial Unit (RePU) ya tiene un tipo de resistencia natural a ejemplos adversarios generados por métodos computacionales, pues tiende a hacer los gradientes tan pequeños que el error de punto flotante los redondea a cero. Estos gradientes son generalmente utilizados en la mayoría de los métodos para generar ejemplos adversarios. Sin embargo esta defensa es muy fácil de evitar haciendo un reescalamiento adecuado del espacio de salida de la red.

Así, este trabajo por un lado corrobora los resultados expuestos en [8]; y por otro sugiere (como se ha sugerido en otros lugares), que el uso de métodos de aprendizaje alternativos, en particular métodos biológicamente plausibles, puede tener un efecto positivo sobre la robustez de redes neuronales artificiales. A pesar de que el método usado no representa una solución completa al problema, apunta a que más investigación en esta área podría ser de gran importancia.

1 Aprendizaje Biológico y Artificial

Resumen

Las redes neuronales artificiales, paradigma del aprendizaje de máquina reciente, presumen estar basadas en las redes neuronales biológicas que encontramos en cerebros de mamíferos y otros animales. En este capítulo se resumen algunos aspectos básicos de ambos dominios, con un enfoque en los mecanismos que permiten a estas redes aprender.

1.1. Aprendizaje Hebbiano y Memoria Asociativa

1.1.1. Aprendizaje y Memoria Biológica

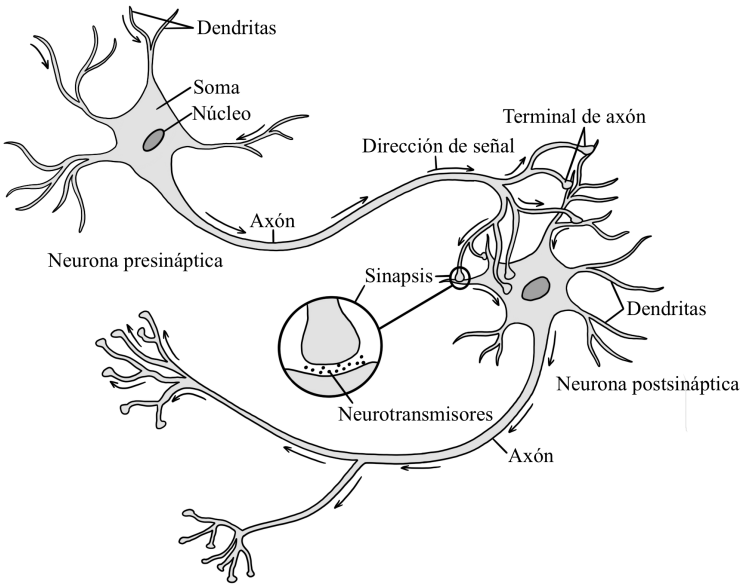
La gran mayoría de los animales contamos con un sistema nervioso que nos permite adaptar nuestros comportamientos y respuestas a los estímulos del ambiente de maneras dinámicas. Además de coordinar nuestros movimientos, la organización de las neuronas en ganglios y cerebros permite procesar de formas muy complejas los estímulos que recibimos del ambiente. La forma en que respondemos a los estímulos depende en gran parte de la secuencia temporal en la que los hemos experimentado. Es decir que los estímulos moldean nuestra forma de responder a ellos. Esta flexibilidad es lo que permite a los animales adaptarse dinámicamente a su entorno, y es lo que se puede definir como aprendizaje. El aprendizaje asociativo es el que lleva al reconocimiento de relaciones contingentes entre eventos [9]. Es de gran importancia para los animales pues les permite predecir la ocurrencia de algunos eventos a partir de otros. Un experimento clásico que ilustra lo anterior es el condicionamiento Pavloviano. Este consiste en presentar a un animal (un perro por ejemplo) con un estímulo relacionado a la biología del organismo (estímulo no condicionado), por ejemplo presentar a un perro con carne, de tal forma que la respuesta al estímulo está condicionada *a priori*. En tal caso esperaríamos que el perro empiece a salivar al ver la carne. Por otro lado se le presenta con un estímulo biológicamente neutro que se busca asociar al estímulo no condicionado, por ejemplo el sonido de una campana. Si se presenta repetidamente el sonido justo antes de presentarle carne, el perro asociará el sonido de la campana con la carne. Eventualmente será suficiente con hacer sonar la campana, sin presentar carne, para generar una respuesta de salivación. Esto se puede interpretar como que los sistemas nerviosos animales tienen la capacidad de aprender a evaluar en cierta medida la probabilidad condicional de un evento dado

otro [9]. Aprendizaje y memoria son conceptos similares, aunque es importante enfatizar la diferencia: el aprendizaje es el proceso por el cual los animales modifican su respuesta futura de acuerdo a los estímulos ambientales a los que son sometidos; por otro lado la memoria es la estructuración interna por medio de la cual se codifica el aprendizaje y se mantiene a través del tiempo.

1.1.2. Aprendizaje en las Neuronas

A escala pequeña, el principal tipo celular que constituye al sistema nervioso de los animales es la neurona. La estructura general y partes básicas de una neurona se muestra en la figura 1.1.

Figura 1.1: Se muestra un esquema de dos neuronas, una pre-sináptica y otra post-sináptica. La dirección del flujo de información se indica con flechas.



Una característica de gran importancia para las neuronas es que mantienen siempre una diferencia de potencial eléctrico entre el interior y exterior de su membrana, gracias a canales iónicos intermembranales, tanto pasivos como activos. Estos canales mantienen una diferencia de potencial eléctrico alrededor de -65mV, pero cuando esta diferencia de potencial se eleva por encima de cierto umbral, canales iónicos regulados por voltaje producen una serie de flujos iónicos altamente coordinados que hacen que la diferencia de potencial se eleve hasta un valor positivo, y vuelva a bajar a un valor ligeramente por debajo del potencial de reposo en un tiempo extremadamente corto, en el orden de un milisegundo. A este salto súbito en el potencial membranal se le conoce como potencial de acción (o *spike*). Lo importante de estos es que la despolarización local que se da al principio del potencial de acción tiene un efecto sobre una pequeña vecindad de la región donde ocurre, lo cual a su vez lleva a que una vecindad se despolarice más allá del umbral, y también lleve a cabo toda la secuencia de despolarización y repolarización. Esto lleva a una reacción en cadena

que va propagando el potencial de acción. De esta forma el potencial de acción puede transmitir información sobre una despolarización local a todo el resto de la neurona. Por otro lado, la comunicación entre neuronas se da en pequeñas estructuras especializadas llamadas sinapsis, donde las membranas de ambas neuronas se encuentran muy próximas (generalmente entre prolongaciones llamadas axones y dendritas, véase 1.1), y en estas se generan una gran variedad de estructuras de señalización. Las sinapsis tienen la propiedad de transmitir información (en la forma de despolarizaciones locales) unidireccionalmente de una neurona pre-sináptica a una post-sináptica (de axón a dendrita). Las sinapsis pueden ser de tipo excitatorio o de tipo inhibitorio, de acuerdo a si su actividad hace más o menos probable la ocurrencia de un potencial de acción respectivamente. Además, las sinapsis pueden ser más o menos efectivas en su acción. A la magnitud de la efectividad de una sinapsis se le conoce como el *peso sináptico*. En este sentido las neuronas forman redes dirigidas con pesos, donde las sinapsis pueden interpretarse como las aristas. En 1894, Santiago Ramón y Cajal propuso que la base material del aprendizaje debía encontrarse en los patrones de conectividad entre neuronas, los cuales, en su visión, se modificaban durante el proceso de aprendizaje [10]. Actualmente, y en concordancia con lo anterior, el proceso de aprendizaje se identifica con el proceso de actualización de los pesos sinápticos.

1.1.3. Plasticidad Hebbiana

En 1949, Donald Hebb publicó "The Organization of Behaviour", donde expuso, entre otras cosas, una teoría más elaborada del proceso de aprendizaje. En particular definió la famosa regla de Hebb: cuando existe una sinapsis entre una neurona A y una neurona B, y la actividad de la neurona A a menudo provoca o facilita potenciales de acción en la neurona B, entonces algún proceso o cambio metabólico ocurre en una o ambas neuronas de tal manera que el peso sináptico entre ellas crece [11]. Esta regla muchas veces se resume como "cells that wire together fire together". La idea es que neuronas que suelen generar potenciales de acción frente a los mismos estímulos refuerzan su conectividad, formando lo que se llama una *asamblea celular*, es decir un grupo de células que suelen generar spikes frente al mismo estímulo y cuyas conexiones son fuertes. La idea es que entonces basta con que el estímulo se presente parcialmente o de forma ruidosa para activar a la asamblea completa. Por ejemplo si una asamblea se vuelve selectiva a un círculo en el campo visual, será suficiente una línea punteada delimitando la misma figura del círculo para evocar la respuesta completa de la asamblea. La propiedad anterior tiene además como consecuencia que la respuesta de las asambleas a un estímulo resiste la pérdida de unas pocas neuronas particulares [12]. Para Hebb el aprendizaje se da gracias a la formación de estas asambleas celulares. Este mecanismo teórico da cuenta del aprendizaje asociativo, pues parejas de asambleas celulares pueden asociarse a su vez cuando la correlación entre sus activaciones es alta, generando conjuntos neuronales que reconocen patrones cada vez más complejos.

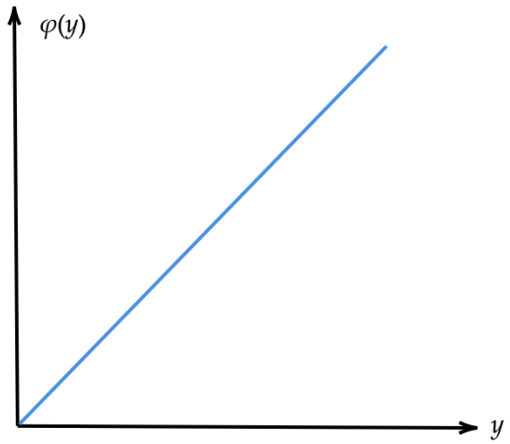
Tiempo después se descubrió un mecanismo molecular correspondiente a la regla de Hebb, conocido como Long Term Potentiation (LTP). Al activar repetidamente ciertas sinapsis excitatorias del hipocampo se obtuvo una potenciación de la sinapsis

que podía durar desde horas hasta días [13]. Con el tiempo se ha visto que más que un único mecanismo, existe una familia de mecanismos que corresponden a la idea original de Hebb; estos suelen involucrar al receptor NMDA y señalización por Ca^{2+} [14]. Podemos modelar este mecanismo con una función de la siguiente forma:

$$\Delta m \propto \varphi(y)x$$

donde Δm es la modificación al peso sináptico, y es la frecuencia de spikes de la neurona post-sináptica, x es la frecuencia de spikes de la neurona pre-sináptica, y φ es una función que especifica la relación entre la frecuencia post-sináptica, la pre-sináptica y la modificación. En el caso Hebbiano φ se puede suponer lineal como en la figura 1.2. A pesar de que claramente la frecuencia de spikes de una neurona es siempre positiva, si en teoría se consideran parejas de neuronas excitadoras e inhibidoras como una sola, se pueden modelar las frecuencias de spikes para incluir el caso de valores negativos [15], aunque por ahora nos limitaremos a describir el caso positivo.

Figura 1.2: Función φ correspondiente a un modelo Hebbiano.

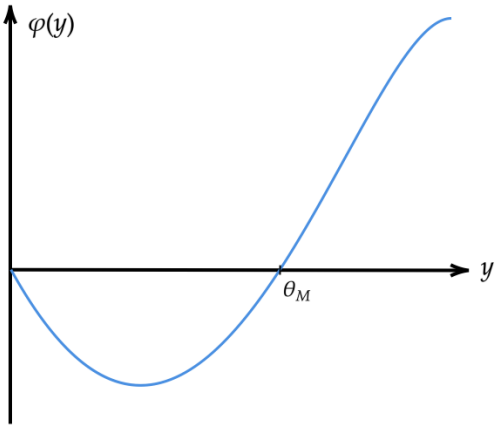


1.1.4. Plasticidad Anti-Hebbiana

Un problema inmediatamente surge del modelo de arriba, y es que si todas las modificaciones sinápticas de largo plazo fueran potenciadoras, los pesos sinápticos tenderían a crecer indefinidamente, en cualquier caso saturando la red y perdiendo todo su potencial para procesar información. Debido a esto, es necesario un mecanismo opuesto, que reduzca los pesos sinápticos de neuronas cuyas activaciones están mal correlacionadas. A mecanismos de esta forma se les conoce como Anti-Hebbianos [16]. También se han encontrado mecanismos moleculares correspondientes, los cuales se conocen como Long-Term Depression (LTD), primero descrito en sinapsis del hipocampo. Se ha visto que casi todas las sinapsis que presentan LTP también presentan alguna forma de LTD [14]. Así, la mayoría de las sinapsis son bidireccionalmente modificables. Una forma de modelar plasticidad Hebbiana + Anti-Hebbiana es como en la teoría CLO (Cooper, Liberman & Oja) [17], donde la ecuación y las variables

representan lo mismo que en la ecuación Hebbiana, pero donde φ es una función continua que es negativa para valores menores a una constante θ_M (comúnmente llamada la constante de modificación) y positiva para valores mayores, como en la figura 1.3. Se puede mostrar que este modelo permite el desarrollo de especificidad en redes neuronales, particularmente para estímulos visuales. Es decir, en un modelo del cortex visual, esta regla de aprendizaje permite el desarrollo de neuronas que se activan frente a ciertos patrones específicos como también se ha visto en cerebros de animales [17].

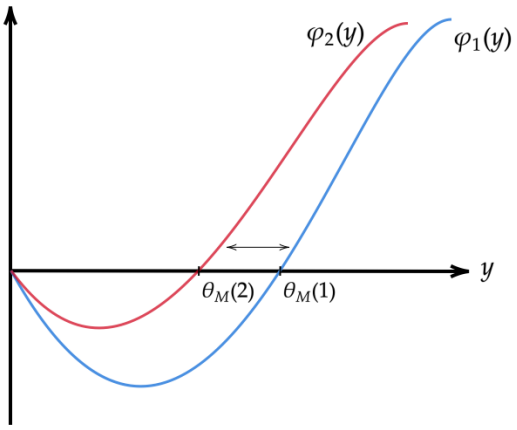
Figura 1.3: Función φ correspondiente a un modelo tipo CLO.



1.1.5. Metaplasticidad y Teoría BCM

Lo anterior funciona siempre y cuando se mantenga los pesos sinápticos dentro de ciertas cotas, pues a pesar de que la introducción de plasticidad anti-Hebbiana resuelve el problema de que todos los pesos se saturen, y por lo tanto permite la codificación de información, no resuelve el problema de que los pesos crezcan monótonamente, sin cota o hacia una cota superior fija. Lo anterior da como resultado problemas en cuanto al desarrollo de especificidad en las neuronas [15]. Existen al menos dos soluciones al problema anterior: la metaplasticidad y el escalamiento sináptico [16]. La metaplasticidad se puede entender como un cambio en la constante de modificación como función de la historia de actividad de la neurona post-sináptica. El mecanismo fue originalmente propuesto teóricamente como parte de la teoría BCM (Bienenstock, Cooper & Munro), y más tarde corroborado experimentalmente [16]. La idea es que cuando la actividad post-sináptica promedio es alta, θ_M crece, y cuando es baja decrece. De esta forma, cuando la neurona post-sináptica es activada mucho por las pre-sinápticas, se vuelve más difícil que un peso sináptico crezca, y más fácil que decrezca; y *vice-versa* [15]. Dar una forma matemática para este modelo es un poco más complejo que para los anteriores, pero la principal diferencia con respecto a la anterior es que aquí la función φ es a su vez una función del tiempo (figura 1.4), así que la podríamos escribir como φ_t, y para calcularla sería necesario integrar sobre la historia de actividad de la neurona post-sináptica. Por otro lado el escalamiento

Figura 1.4: Función φ correspondiente a un modelo tipo BCM.



sináptico es un escalamiento normalizante de los pesos sinápticos (de todas las sinapsis hacia una neurona) que los mantiene dentro de ciertas cotas, y también se ha demostrado experimentalmente [16].

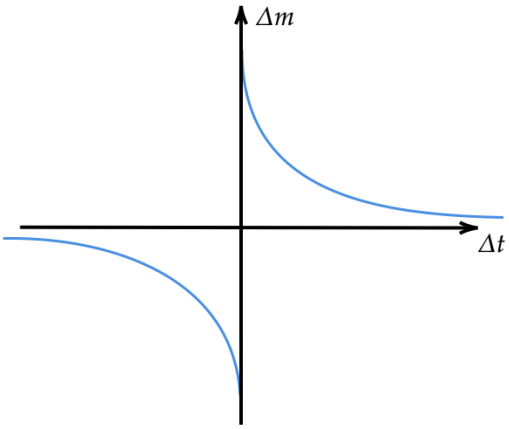
En ocasiones la regla de Hebb se interpreta como: cuando la correlación entre los spikes de neuronas conectadas es alta, la conexión entre ellas se refuerza (por ejemplo [18]). Sin embargo lo que dice la regla de Hebb, si la interpretamos estrictamente, es que el peso sináptico de A a B se refuerza cuando A participa en la generación de potenciales de acción en B. Es decir, que hay un orden causal: primero es el potencial de acción en A y luego el potencial de acción en B. Esta relación causal se preserva en el fenómeno conocido como Spike Time Dependent Potentiation (STDP) [19]. Se ha observado que se pueden inducir potenciaciones y depresiones en los pesos sinápticos con parejas de spikes pre y post-sinápticos discretos, y el tipo de modificación depende del intervalo de tiempo y orden entre ellos. El régimen más común de STDP es uno en el que la magnitud del cambio es mayor entre menor sea el intervalo entre spikes, y la modificación es potenciadora cuando el spike pre-sináptico precede al post-sináptico (pre→post), y depresiva si post→pre (como en la figura 1.5). Sin embargo existen muchas variaciones de esto, y los fenómenos de arriba se pueden explicar a partir de mecanismos STDP, incluyendo la metaplasticidad tipo BCM [16, 19, 20].

1.2. Aprendizaje Artificial con Backpropagation

1.2.1. Redes Neuronales Artificiales

Matemáticamente, se suele modelar a las redes neuronales biológicas de la siguiente manera: definimos una *red neuronal artificial* (*artificial neural network*, ANN) como una red dirigida con pesos, donde cada nodo representa una neurona, cada arista una conexión entre neuronas (la existencia de una sinapsis), los pesos de las aristas (con valores en $\mathbb{R}$) representan los pesos sinápticos, y los pesos de los nodos (sesgos) representan la excitabilidad intrínseca de las neuronas (estos no siempre se implementan

Figura 1.5: Cambio en peso sináptico (Δm) como función de la diferencia temporal entre la activación pre y post-sináptica (Δt) en un modelo STDP.



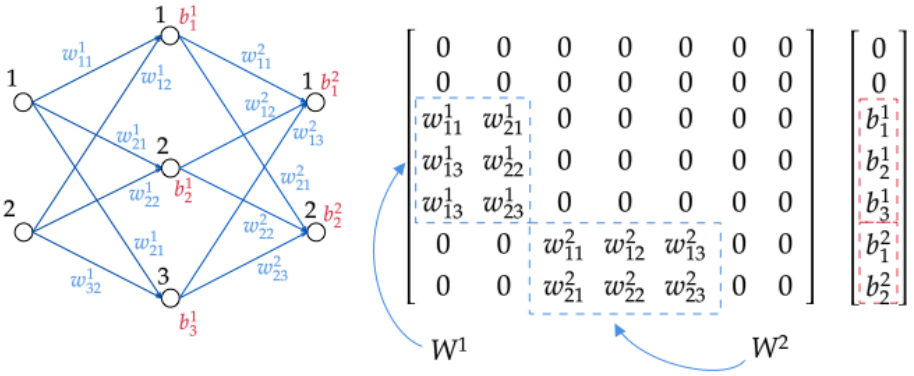
en los modelos). Por simplicidad siempre tomamos las ANN con todas las conexiones, así podemos representar cualquier ANN como una matriz cuyas entradas son los pesos de las aristas numeradas y un vector que representa los sesgos. Simplemente dejamos fuera las aristas con pesos iguales a cero al hacer su diagrama. Lo más común es dotar a estas redes de un poco más de estructura. La más simple es la red neuronal feed-forward, donde los nodos se organizan en *capas* ordenadas de forma lineal, de manera que todos los pesos entre nodos de cada capa y entre nodos de capas que no son adyacentes en el orden se fijan a cero (véase fig. 1.6). De esta forma podemos partir la matriz de pesos en un conjunto de matrices más pequeñas, cada una de las cuales representa el valor de los pesos entre las neuronas de dos capas adyacentes; y al vector de sesgos lo podemos dividir en un conjunto de vectores consistente con las capas. En adelante nos referiremos a este tipo de redes cuando hablemos de una ANN. A la primera capa de una ANN le diremos la *capa de entrada*, a la última capa la llamaremos *capa de salida*. Por convención denotaremos por N al número de capas de la red. Dada una numeración interna de las neuronas en cada capa y $\ell \in \{1, ..., N\}$, denotamos por W^ℓ a la matriz de pesos entre nodos de la ℓ-ésima capa y la $(\ell+1)$-ésima, b^ℓ es el vector de sesgos, y tomamos $n(\ell)$ igual al número de nodos en la capa número ℓ.

Modelamos el nivel de excitación o la frecuencia de potenciales de acción de una neurona como un valor numérico (activación) asignado a cada nodo de la red. De manera similar a las redes biológicas, las activaciones de cada neurona en una ANN dependen de las activaciones de las neuronas conectadas a ella. Como las neuronas de la primera capa no tienen ninguna otra neurona conectada a ellas, sus activaciones se inicializan en algún vector de valores iniciales $x \in \mathbb{R}^{n(1)}$. Para cada $\ell \in \{1, ..., N-1\}$, definimos la función de activación:

$$\alpha(a^\ell) := \vec{\sigma}\left(W^\ell \cdot a^\ell + b^\ell\right)$$

donde $\sigma : \mathbb{R} \to \mathbb{R}$ es alguna función creciente (podría ser la identidad incluso, aunque es importante que sea no lineal para que la red tenga la propiedad de universalidad

Figura 1.6: Estructura de una red neuronal artificial, con su representación como gráfica dirigida con pesos (izquierda), y como matriz de pesos y vector de sesgos (derecha). En este caso una ANN de 3 capas neuronales, donde la capa de entrada tiene 2 neurona (izquierda), la segunda capa 3 (en medio) y la capa de salida 2 (derecha).



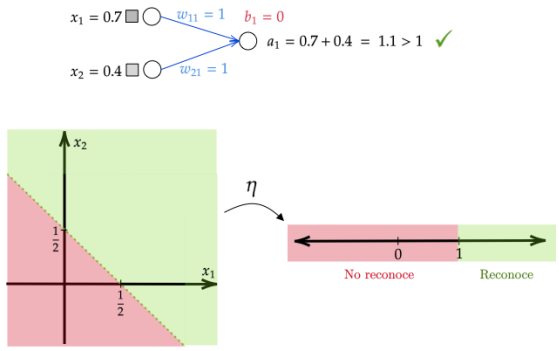
[21]), y $\vec{\sigma}$ representa la operación de aplicar σ entrada a entrada sobre un vector. La función de activación puede tomar muchas formas dependiendo de la función σ, y la idea detrás de esta es simular la forma en la que la actividad de las neuronas presinápticas determina la actividad de las neuronas post-sinápticas. Luego, calculamos para cada capa un *vector de activación* de manera recursiva: $a^1 := x$, y para cada $\ell \in \{1, ..., N\}$ tomamos $a^{\ell+1} := \alpha(a^\ell)$. De esta forma se propaga una activación en la capa de entrada hacia adelante al resto de las capas (*ergo* el nombre feed-forward). Visto de forma general, una ANN define una función $\eta : \mathbb{R}^{n(1)} \to \mathbb{R}^{n(N)}$ dada por $\eta(x) := a^N$ la activación de la capa de salida resultante de propagar el vector x por la red.

1.2.2. ANN como Clasificadores

En la práctica, las ANN se utilizan para reconocer patrones en datos, por ejemplo la presencia de cierto objeto en una imagen. Cualquier archivo de imagen se puede representar como un vector en un espacio euclidiano (posiblemente de dimensión muy alta), y por lo tanto es susceptible de usarse como argumento de una ANN (vista como función). Luego, podemos segmentar el espacio de dimensión el número de neuronas de la última capa en regiones que representen ciertos patrones; de tal manera que si el resultado de aplicar la ANN a la imagen cae en una región correspondiente a algún patrón, decimos que la ANN *reconoce dicho patrón en la imagen*. Un ejemplo sencillo de esto sería una red que reconoce, para una imagen compuesta por dos pixeles con valores entre 0 y 1, si el promedio del valor de los pixeles es mayor o menor a un medio. Para esto tomamos una red con dos capas, donde la primera capa consiste de dos neuronas (correspondientes a los pixeles) y la segunda capa es una única neurona que mide la respuesta. Podemos decidir que queremos que la red reconozca el patrón cuando la activación de la neurona de salida sea mayor o igual a 1, y que en otro caso

no. Es decir, dividimos el espacio de respuesta (la recta real) en regiones $(-\infty, 1)$ y $[1, \infty)$, correspondientes a patrones "el valor promedio de los pixeles no es mayor a 0.5" y "el valor promedio de los pixeles es mayor que 0.5". De hecho, dadas las condiciones de este ejemplo, podemos encontrar pesos y sesgos que hacen que la red reconozca el patrón si y solo si de hecho ocurre en la imagen: esto se logra asignando pesos de 1 a cada arista y sesgos de 0 [22].

Figura 1.7: Una ANN muy simple que reconoce cuando el promedio de los valores de las dos neuronas en la capa de entrada es mayor o igual a 0.5.



1.2.3. Aprendizaje en ANNs

Un ejemplo más complejo sería el problema de tomar imágenes de ciertas dimensiones con un dígito escrito, y que la red reconozca el dígito que está escrito correctamente. Este problema, a diferencia del anterior, no puede ser resuelto de manera analítica, y cualquier solución es necesariamente una aproximación. Uno de los problemas centrales de las redes neuronales artificiales es partir de una estructura de nodos y aristas fija y un criterio de reconocimiento, y encontrar los pesos y sesgos que mejor aproximen la solución de un problema para un tipo de entrada. A este problema lo podemos nombrar *el problema del aprendizaje*. Las respuestas a este problema se pueden clasificar en dos tipos: aprendizaje supervisado y no supervisado. El aprendizaje supervisado es el que requiere de muestras de datos etiquetados con su clase correcta, de manera que la ANN pueda actualizar sus pesos y sesgos sabiendo si la respuesta que da a los ejemplos es correcta o incorrecta. Es decir que se busca un algoritmo para maximizar el ajuste de la ANN a los datos etiquetados, bajo el supuesto que si la muestra es suficientemente general, la ANN se desempeñará razonablemente bien con cualquier entrada. Por otro lado, los métodos de aprendizaje no supervisado buscan reglas de actualización de los pesos que no dependan de que los datos estén etiquetados [23, 24]. La ventaja de estos métodos es que no requieren de grandes conjuntos de datos etiquetados. Por otro lado, la desventaja es que la ANN no aprende a etiquetar los patrones que reconoce, simplemente a distinguirlos o a generar representaciones eficientes de estos [24]. Por supuesto también es posible desarrollar métodos de aprendizaje que sean una combinación de los dos tipos mencionados arriba.

1.2.4. Backpropagation

El método de aprendizaje supervisado más usado actualmente es el que se conoce como "backpropagation". Llamaremos a redes entrenadas de esta forma ANN-BP. La idea se basa en definir, para cada vector de entrada, una *función de costo*, esto es, una función diferenciable del espacio de salida a un intervalo de los reales, cuyo valor sea mayor entre más diferente sea el vector de salida del vector "correcto" que se esperaría dado el vector de entrada. Un ejemplo de una función de este tipo es la suma de los cuadrados de las diferencias de las entradas de los vectores. Si tomamos x un vector de entrada, $\eta(x)$ el vector de salida de la ANN, y $y(x)$ el vector respuesta, podemos definir una función de costo como:

$$C(x) = \sum_i (y_i(x) - a_i(x))^2 = \|y(x) - \eta(x)\|^2$$

Definido esto, el problema de encontrar los pesos y sesgos que maximicen el poder predictivo de la ANN se puede ver como un problema de minimizar la función de costo para todas las entradas de los ejemplos. Suponiendo que la función σ de la función de activación es diferenciable, podemos calcular el gradiente de la función de costo respecto a los pesos o a los sesgos de cada capa. Gracias a la regla de la cadena para funciones de varias variables y a la estructura de las redes feed-forward, es posible obtener expresiones relativamente simples de estos gradientes como función de los gradientes en la siguiente capa. De ahí el nombre backpropagation, pues cada gradiente se calcula recursivamente a partir del de la siguiente capa, propagando el error en sentido inverso a cómo se propaga la activación. Luego, como el gradiente apunta en la dirección de máximo crecimiento del costo como función de los pesos o de los sesgos, modificar los pesos o sesgos con pequeños pasos en la dirección opuesta al gradiente siempre nos mueve hacia un mínimo local de la superficie de costo (suponiendo que exista). Al procedimiento anterior se le conoce como *descenso del gradiente*. Aplicando este procedimiento suficientes veces podemos aproximar un mínimo de la función de costo.

Podemos resumir las expresiones necesarias para realizar el descenso del gradiente por backpropagation en cuatro ecuaciones. La primera nos dice la derivada del costo con respecto a cada una de las activaciones de la última capa (es decir que tenemos el gradiente del costo con respecto a las activaciones de la última capa).

$$\frac{\partial C}{\partial a_i^N} = 2(y_i(x) - a_i^N) \tag{BP1}$$

La segunda nos da el gradiente del costo respecto a las activaciones de una capa ℓ cualquiera en términos del gradiente respecto a la siguiente capa. A partir de esto podemos calcular todos los gradientes de manera recursiva.

$$\nabla_{a^\ell} C = \nabla_{a^{\ell+1}} C \cdot W^\ell \tag{BP2}$$

Finalmente tenemos ecuaciones que nos dan la derivada del costo respecto a un peso o un sesgo cualesquiera en términos de las derivadas parciales respecto a las

activaciones, la derivada de la función σ, y las activaciones (todos los cuales están dados en las ecuaciones de arriba).

$$\frac{\partial C}{\partial W_{ij}^{\ell}} = \frac{\partial C}{\partial a_i^{\ell+1}} \sigma'(u_i) a_j^{\ell} \tag{BP3}$$

$$\frac{\partial C}{\partial b_i^{\ell}} = \frac{\partial C}{\partial a_i^{\ell+1}} \sigma'(u_i) \tag{BP4}$$

1.2.5. Deep Learning

El tipo de red neuronal entrenado por backpropagation más exitoso en años recientes es la red neuronal profunda (deep neural network, DNN). Estas redes enfatizan el uso de varias capas neuronales escondidas. La idea de esto es que cada capa neuronal añade capacidad de abstracción en las representaciones de la red, haciendo que los patrones que reconoce la red estén menos sujetos a detalles contingentes de los mismos. Además de la arquitectura profunda, estas redes implementan una forma de activación especial conocida como capas convolucionales. El principio de estas capas está dado por la idea de filtros, que son pequeñas matrices cuadradas que se recorren por una capa (vista como matriz de dos dimensiones) calculando el producto punto de cada región de la capa y el filtro, y obteniendo de esta manera las activaciones de la siguiente capa. La siguiente matriz obtenida de esta manera se conoce como la convolución de la capa de input con el filtro. Dado que el resultado de la convolución de una capa con estructura 2 dimensional y un filtro es de nuevo una capa con estructura 2 dimensional (generalmente más pequeña), se puede aplicar el procedimiento de convolución sucesivamente para cierto número de las capas iniciales. En estas capas lo que se aprende son las entradas de los filtros. Estas redes tienen varias ventajas, particularmente en el problema de reconocimiento de imagenes de pixeles. Por un lado son computacionalmente muy eficientes [25], pues permiten la reutilización de ciertos patrones locales primitivos a lo largo de toda la imagen, evitando la necesidad de que las primeras capas de la red tengan que optimizar todos los pesos, los cuales para una imagen grande pueden ser una cantidad enorme. También reducen el problema de overfitting, es decir, evitan que la red aprenda una solución demasiado particular al conjunto de datos de entrenamiento pero sin capacidad de generalizar a muestras fuera de la distribución de entrenamiento.

Las DNN son el paradigma actual del reconocimiento visual artificial. Pues aunque las ANN-BP se desempeñan muy bien en problemas simples como el reconocimiento de dígitos, las DNN pueden escalarse a problemas muy complejos que requieren un alto nivel de abstracción de una manera computacionalmente viable, en donde una ANN-BP clásica sería inviable [26].

1.3. ANN-BP y Redes Biológicas

Claramente muchos aspectos de la enorme complejidad funcional de las redes biológicas no queda representada en las redes artificiales (de la forma que se definieron

arriba). Sin embargo, el propósito de las redes artificiales (en el contexto del aprendizaje de máquinas) no es simular todos los aspectos de los sistemas biológicos en los que se basan, sino únicamente aquellos aspectos de importancia para realizar tareas particulares. En este sentido se ha visto que con modelos relativamente simples (comparados con la complejidad del cerebro) se pueden realizar tareas cognitivas como las que realiza el cerebro mamífero, por ejemplo el procesamiento de imágenes y lenguaje naturales [1], o el aprendizaje de juegos de mesa y videojuegos a niveles iguales o mayores a los que puede alcanzar un humano [2, 3]. En las secciones anteriores hemos hablado sobre algunos aspectos del aprendizaje en redes neuronales biológicas, y al problema de encontrar los parámetros específicos de una ANN que mejor aproximen alguna función (o realicen una tarea) lo hemos denominado "el problema del aprendizaje". Hemos hablado de una solución particular, el descenso del gradiente por backpropagation, la cual es el paradigma actual del campo. La pregunta natural es si esta forma de aprendizaje artificial tiene algo en común con el aprendizaje biológico. Hasta ahora no ha surgido evidencia empírica contundente de que exista algún mecanismo análogo al BP en el cerebro [24, 27, 28], aunque si existen modelos teóricos biológicamente plausibles que llevan a cabo un aprendizaje equivalente al descenso del gradiente, y se hipotetiza que dada la gran efectividad de este método, se esperaría que alguno de estos mecanismos se haya fijado evolutivamente en los cerebros biológicos [4, 24]. En general cuando decimos que el backpropagation no es biológicamente plausible no estamos afirmando que el cerebro no cuente con mecanismos que minimicen alguna función de costo, tal vez incluso por medio de un mecanismo comparable a la propagación de errores en sentido feedback [4]. De hecho incluso varios mecanismos de aprendizaje no supervisado de tipo Hebbiano se pueden ver como si realizaran una función análoga a la de minimizar el error de reconstrucción de las entradas [29]. Se puede argumentar también que todo el aprendizaje efectivo se puede ver como supervisado a algún nivel [23]. Más bien lo que no se ha observado es un mecanismo equivalente en el sentido de una supervisión a larga distancia [8]. Pues las sinapsis modifican sus pesos de acuerdo únicamente a información local, por lo que un modelo biológicamente plausible del backpropagation debería representar los errores (que se propagan en el BP) de forma local [24], otro problema es que en el backpropagation las activaciones se propagan hacia adelante y los errores hacia atrás de forma perfectamente simétrica, lo cual implicaría que cada sinapsis debería tener una pareja exactamente equivalente a ella pero en la dirección opuesta, cuyo peso se mantiene alineado con el primero a lo largo del tiempo, conocido como el problema del transporte de pesos [24, 30]. Se ha visto que para algunos problemas simples, este problema se puede resolver simplemente asignando pesos hacia atrás de forma aleatoria, manteniendo un nivel de aprendizaje comparable con el método BP [31]. Sin embargo para tareas más complejas esta técnica se ha mostrado insuficiente [32]. Otro problema deviene simplemente del hecho de que el modelo de ANN usado comúnmente toma las activaciones como números reales, lo cual permite el cálculo directo de derivadas parciales; lo cual constrasta con las redes biológicas donde las activaciones se dan en la forma de trenes de potenciales de acción [4, 24].

En resumen no existe evidencia en redes biológicas de que una señal guía basada en un error respecto a un resultado esperado, varias capas adelante en la red, coordine de

forma específica la actualización simultánea de los pesos de la red. Aunque si existen modelos en ciertos sentidos equivalentes que evitan algunos (pero no todos) estos problemas [24]. Preguntas de este tipo, como si el cerebro realiza o no minimización de funciones de costo, siguen abiertas y son un área de investigación actual [4].

Por otro lado, reglas de aprendizaje tipo Hebbiano y anti-Hebbiano han encontrado mucho apoyo experimental a lo largo de los años, y existen diversos modelos matemáticos para estas [16]. Una dirección interesante de trabajo en redes artificiales es la implementación de este tipo de reglas en ANN y el estudio de las propiedades que se derivan de ello [8]. Una posible interrogante, que tiene relación con este trabajo, es si una red entrenada con un régimen más biológicamente plausible que el BP tendrá una mejor respuesta a señales modificadas de maneras específicas para engañar a la red. A la resistencia de una red a ser confundida o engañada por una señal manipulada (con conocimiento total de la red) le diremos la *robustez* de la red (ver capítulo 3). Cabe destacar que esta hipótesis no es comprobable directamente. Es decir, no podemos decir que las redes biológicas sean más robustas que las redes ANN-BP, pues de las segundas tenemos la ventaja de poder acceder a todos los parámetros de la red, y por lo tanto de generar señales muy específicas que engañen a esa red en particular. De las redes biológicas sería casi imposible obtener incluso una muestra representativa de los pesos sinápticos [26]. Aún así, sabemos que el sistema visual de los primates es sensible a ciertas imágenes capaces de generar ilusiones ópticas [26]. A partir de redes entrenadas con regímenes más biológicos podemos aproximarnos a la pregunta de si las redes biológicas son más robustas que las redes feedforward entrenadas con backpropagation. Además de su valor para la neurociencia, responder la pregunta anterior podría tener un gran valor para la disciplina del aprendizaje de máquinas, pues es muy deseable que los métodos de esta lleven a cabo su función de la forma más robusta posible.

Además de lo que mencionamos arriba, cabe resaltar que las ANN difieren de las redes biológicas en algunas otras formas importantes. Por ejemplo, el tipo de red artificial más común es la red feedforward, aunque en el cerebro las conexiones recurrentes y laterales son muy comunes y probablemente tienen una función importante [26]. Otro aspecto curioso de las redes artificiales es que suelen comenzar a entrenarse desde pesos aleatorios, mientras que en las redes biológicas los pesos iniciales suelen estar determinados genéticamente y afinados por la evolución [33]. En general muchas preguntas interesantes y direcciones de trabajo futuro surgen de contrastar las ANN y las redes biológicas, y es claro que el dialogo entre las dos promete un gran enriquecimiento mutuo.

2 Aprendizaje Artificial Bioinspirado

Resumen

Para que las redes neuronales artificiales se asemejen más a las redes biológicas se han propuesto varios métodos de aprendizaje como alternativas al backpropagation, el cual no tiene correlato biológico directo. En este capítulo se presenta la regla de Oja [34], una regla hebbiana para el aprendizaje de componentes principales. Después se presenta una generalización de la anterior dada por Krotov & Hopfield [8], la cual aprende detectores de rasgos que guardan un gran parecido con los elementos de la distribución de entrada. Esta última fue implementada en este trabajo y se presentan aquí los resultados de esta implementación. Se logró reproducir, a grandes rasgos, los resultados del artículo original.

2.1. El Supuesto de Localidad y la Regla de Oja

2.1.1. La Regla de Oja Encuentra Componentes Principales

Como se mencionó en la sección anterior una diferencia entre las redes BP y las redes biológicas es que en las primeras el aprendizaje supone que la actualización de los pesos sinápticos de cada neurona está informada por el estado de todas las demás neuronas de la red. Mientras que en las redes biológicas las neuronas tienen acceso únicamente al estado propio y a la información que reciben de las neuronas conectadas a ellas, además de la información disponible en el medio general. Lo anterior apunta a que debe ser posible generar redes neuronales artificiales con la capacidad de generalización entrenadas con una regla de aprendizaje local, en el sentido de que la regla de actualización de pesos entrantes a cada neurona sea una función únicamente de las activaciones de las neuronas conectadas a ella.

Existen varias reglas de aprendizaje con esta propiedad. Una de las primeras fue la *regla de Oja* [34]. Esta regla se definió originalmente para una ANN con solo una capa de entrada y una de salida, y donde la capa de salida cuenta con una única neurona. Se mostró que dada esta regla de aprendizaje los pesos sinápticos de la red convergen a un vector de norma 1 en la dirección del componente principal de los datos de entrada usados para entrenarla. Esta regla está dada (en su forma discreta) de la siguiente manera:

$$w(t+1) = w(t) + \gamma[\eta(x)x - \eta^2(x)w(t)]$$

Donde $w(t)$ es el vector de pesos en el tiempo t, γ es una velocidad de entrenamiento, x la entrada, y $\eta(x)$ el valor de salida de la red dado x. Lo que vemos es que el primer término en los corchetes de la derecha, $\eta(x)x$, representa el aprendizaje Hebbiano, pues cuando la respuesta de la neurona a una entrada x es grande, el vector de pesos es empujado en la dirección de ese patrón. Por otro lado, el segundo término de los corchetes aproxima la operación normalizar el vector, evitando que crezca indefinidamente y asegurando que se mantenga cercano a una norma de 1 [22].

2.1.2. Análisis de Componentes Independientes y Reglas Anti-Hebbianas

Un problema con este modelo es que no admite una generalización natural a redes de mayor tamaño. Si se usara la misma regla de aprendizaje en una red con más de una neurona en la capa escondida, lo más probable es que todos los pesos de cada neurona en la capa oculta tiendan al mismo vector, que es la componente principal de la distribución de entrada. Para evitar esto, lo que se necesita es introducir asimetría en la regla de aprendizaje o competencia entre las neuronas (una regla anti-Hebbiana) [35]. Con esto se puede asegurar que los pesos de las neuronas converjan a componentes independientes de la distribución.

Cuando se habla de reglas anti-Hebbianas en redes neuronales artificiales, normalmente a lo que se refiere es a alguna forma de conectividad lateral dentro de las capas de manera que se descorrelacionen sus activaciones. Esto se logra introduciendo conexiones laterales inhibidoras entre las neuronas de la misma capa. La descorrelación de las activaciones promueve que la mayor cantidad posible de información se preserve en las activaciones de una capa [35]. Esto, aunado a una regla como la de Oja, tiene el efecto de que la red neuronal funcione como un analizador de componentes principales independientes.

Un modelo muy simple para lograr lo anterior se obtiene tomando la siguiente regla de aprendizaje de pesos laterales:

$$\Delta L_{ij} = -\rho \cdot a_i a_j$$

donde L_{ij} es el peso lateral entre las neuronas i y j de la capa de salida, ρ es una constante de aprendizaje, y a_i es la i-ésima activación de la capa de salida. Esta regla hace que entre mayor sea la correlación entre las activaciones de dos neuronas de la capa de salida, mayor sea la inhibición mutua de sus actividades. Haciendo que la dinámica de aprendizaje tienda a maximizar la descorrelación de las activaciones de las neuronas de una capa. Esta no es la única forma de introducir este tipo de regulación, y en lo que sigue definiremos otro método con estas propiedades.

2.2. Un Modelo Bioinspirado

Krotov and Hopfield [8] desarrollaron una generalización del modelo de Oja, que también cumple con el supuesto de localidad y que tiene un desempeño comparable al

de una red BP en los conjuntos de prueba MNIST (compuesto de imagenes de 28×28 pixeles de dígitos escritos a mano) y CIFAR-10 (compuesto de imagenes de 32×32 pixeles a color de imagenes de diferentes categorías). Su aproximación se basa en que la actividad coordinada de una neurona presináptica y una postsináptica lleva a cambios en el peso sináptico entre ellas, de manera similar a la teoría BCM. La principal diferencia es que en vez de que la dinámica que estabiliza los pesos sinápticos (la metaplasticidad en la teoría BCM) se halle en un umbral dinámico, más bien consideran conexiones recurrentes inhibitorias entre neuronas de la misma capa, y una dinámica rápida (en relación a la dinámica de los pesos sinápticos) que determina qué neuronas son más afines al patrón que representa la entrada, y únicamente estas cambian sus pesos sinápticos para capturar el patrón. Además las neuronas que sean afines al patrón pero que no superen un umbral determinado tendrán sus pesos empujados en la dirección opuesta. De esta forma se asegura que las neuronas aprendan diversas representaciones tempranas y no converjan todas sobre el mismo patrón. La idea es que en este modelo las neuronas compiten (espacialmente) por los patrones mientras que en un modelo BCM los patrones compiten (temporalmente) por las neuronas [8].

En este modelo el vector de pesos sinápticos de cada neurona converge eventualmente a un punto sobre la esfera unitaria. Para evitar que la forma particular de la esfera unitaria en espacios euclidianos restrinja el desempeño de la red, el modelo se define de tal manera que los vectores de pesos de cada neurona converjan a la esfera unitaria de un espacio l_p, con $p \geq 2$ definido de antemano como un hiperparámetro del modelo. Para esto se usa el siguiente producto interno que depende de la fila de pesos correspondiente a la neurona de la capa escondida:

$$< v, w >_i = \sum_k |W_{ik}|^{p-2} v_k w_k$$

Si $p = 2$ entonces el producto interno es el canónico.

Para un vector de entrada $x \in \mathbb{R}^n$ se define la *corriente* de cada neurona como se calcularía normalmente la activación:

$$I_i(x) = < W_{i\_}, x >_i$$

donde $W_{i\_}$ representa la i-ésima hilera de W. Estas corrientes se usan luego para calcular la dinámica de las *afinidades* h_i de las neuronas por un patrón dado:

$$\tau \frac{dh_i}{dt} = I_i(x) - w_{inh} \sum_{j \neq i} \text{máx}\{0, h_j\} - h_i$$

donde τ es un ritmo de aprendizaje pequeño (esta dinámica debe ser rápida), w_{inh} es un parámetro de inhibición que funciona a escala global (y es el mismo entre todas las neuronas de una misma capa). Las h_i se usan para calcular el valor de la siguiente función de actualización de pesos sinápticos:

$$g(h) = \begin{cases} 1 & \text{si } h_* \leq h \\ -\Delta & \text{si } 0 \leq h < h_* \\ 0 & \text{en otro caso} \end{cases}$$

donde Δ es un hiperparámetro del modelo y h_* un hiperparámetro que funciona como divisoria entre los valores de h que empujan una neurona hacia un patrón, o lo alejan de este. Es decir, cuando $g(h) = 1$ el régimen de aprendizaje es positivo (Hebbiano) y cuando $g(h) = -\Delta$ es negativo (anti-Hebbiano). Esta función sirve un propósito análogo a la función φ_{CLO} del capítulo 1, y el valor h^* corresponde a la θ_M 1.3, aunque aquí la función es discontinua.

Lo siguiente es actualizar los pesos entre la capa de entrada y la capa de neuronas ocultas usando los valores de arriba:

$$\tau_L \frac{dW_{ik}}{dt} = g(h_i)(R^p x_k - < W_{i\_}, x >_i W_{ik}).$$

La idea de esta forma de aprendizaje es que cada neurona de la capa oculta converge a un patrón distinto. Además, como este modelo generaliza a la regla de Oja, esperaríamos que los patrones a los que las neuronas convergen tengan alguna relación con los componentes principales de la distribución, y por lo tanto que las activaciones preserven buena parte de la información de la distribución original.

2.3. Implementación del Modelo

Para implementar el modelo se intentaron diferentes aproximaciones. La primera fue implementar el modelo directamente como se describe arriba, encontrando puntos estables de sistemas de ecuaciones. Sin embargo esto resultó ser inviable dado que los sistemas de ecuaciones diferenciales que se deben resolver para una buena cantidad de neuronas en la capa oculta resulta demasiado costoso computacionalmente. Finalmente se optó por usar la implementación simplificada que se utiliza en el artículo original [8]. Esta consiste en tomar el valor de las corrientes como indicador de la afinidad por un patrón, ordenar las neuronas de acuerdo a su afinidad, y aplicar una versión modificada de la función de intensidad de modificación de pesos sinápticos dada por:

$$g(i) = \begin{cases} 1 & \text{si } i = N \\ -\Delta & \text{si } i = N - k \\ 0 & \text{en otro caso} \end{cases}$$

donde i es la posición que ocupa la neurona en el orden anterior. Así, para cada imagen de entrada se calcula el valor de la función g para cada neurona, y con base en esto se modifican los pesos sinápticos una pequeña cantidad en la dirección dada por la fórmula más general de arriba. Este algoritmo se puede usar para generar pesos sinápticos que, luego de agregar una capa de etiquetamiento entrenada con *backpropagation*, se desempeñan a un nivel comparable con una red con una arquitectura igual entrenada *end-to-end* por *backpropagation*.

Se lograron reproducir los resultados descritos por Krotov & Hopfield [8] para una arquitectura casi idéntica a la que se describe cómo óptima en el artículo para el conjunto de datos de entrenamiento MNIST. Esta arquitectura está dada por una red de tres capas: la capa de entrada con 784 nodos que recibe los valores de las imágenes;

la segunda de 2000 nodos, y la capa de categorización de 10 nodos correspondientes a los dígitos. La función de activación está dada de la siguiente forma:

$$a^2(x) = \mathrm{RePU}_{4.5}(W^1 \cdot x)$$

$$a^3(a^2) := 0.01(W^2 \cdot a^2)$$

donde las constantes 4.5 en el exponente de la primera y el factor multiplicativo 0.01 se seleccionaron numéricamente en el artículo original, y la función $\mathrm{RePU}_\alpha(x) := (\mathrm{ReLU}(x))^\alpha$ es una "unidad polinomial rectificada". Esta es la misma que discuten Krotov y Hopfield y muestran que tiene ciertas propiedades interesantes en relación a la forma de los detectores de rasgos y robustez [36, 37]. Como función de costo se tomó la función de *entropía cruzada*, la cual puede definirse como:

$$\mathrm{CE}(\bar{y}, i) := -\log\left(\frac{\exp(\bar{y}_i)}{\sum_j \exp(\bar{y}_j)}\right)$$

donde $\bar{y}$ es el vector de salida de la red, e i es el índice de salida correcto. En el artículo se agrega en la última capa la aplicación entrada a entrada de la función tanh y luego como función de costo la m-norma (dada por $\|x\|_m := \sqrt[m]{\sum_i x_i^m}$) elevada a la m potencia. Es decir, $L(\bar{y}, i) := \|y(i) - \bar{y}\|_m^m$, donde m es un hiperparámetro del modelo y donde

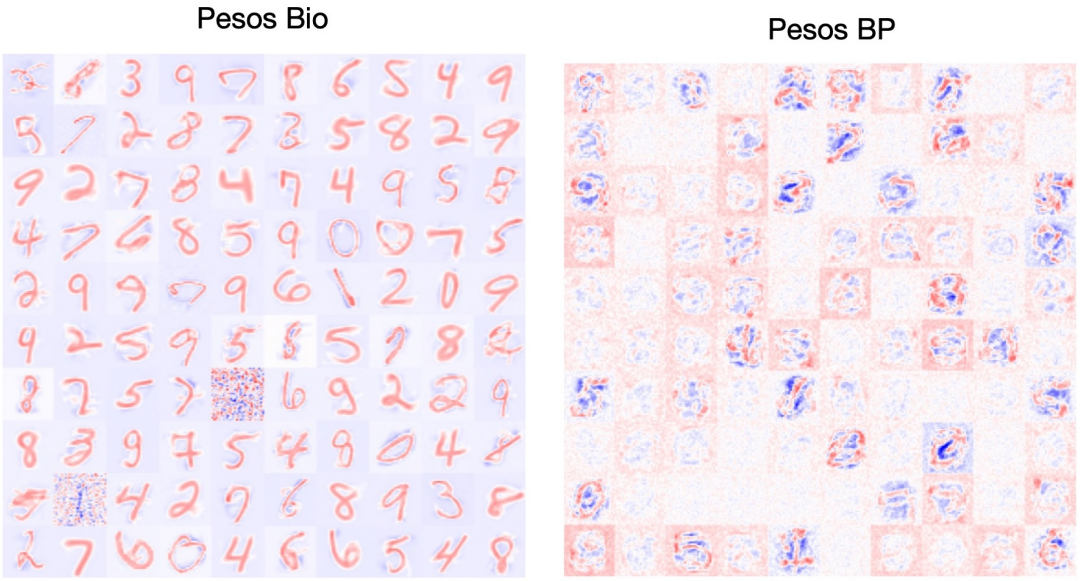
$$y(i)_j := \begin{cases} 1 & \text{si } j = i \\ -1 & \text{si } j \neq i \end{cases}.$$

Sin embargo la entropía cruzada tiene propiedades afines a la tarea de clasificación. Esta función se puede ver como la composición de la función SOFTMAX (dada por $x_i \mapsto \frac{\exp(x_i)}{\sum_j \exp(x_j)}$) y la función de costo logaritmo negativo de la verosimilitud (nll). La primera de estas es una función que toma el vector de salida y lo manda a un vector de probabilidad, donde todas sus entradas se encuentran en el intervalo $[0, 1]$ y la suma de todas es igual a 1. Esto significa que podemos interpretar el vector de salida pasado por SOFTMAX como un vector de la certidumbre con la que la red asigna la entrada a cada categoría de salida. Luego la función de costo nll manda un vector de probabilidad al logaritmo negativo de la probabilidad de la categoría correcta. Por estas propiedades se prefirió usar esta función de costo, además de que se observó que el uso de esta no percude el desempeño de la red.

Para entrenar la red se utilizó una versión normalizada del conjunto de datos MNIST. Esta normalización es una transformación lineal de los datos de manera que el promedio de los valores de todo el conjunto esté en 0 y la desviación estándar sea de 1. El promedio de los valores del MNIST es $\mu_{MNIST} \approx 0.1307$ y su desviación estándar $\sigma^2_{MNIST} \approx 0.3081$. Para esto se aplica la siguiente transformación entrada a entrada $x' = \frac{(x - \mu_{MNIST})}{\sigma^2_{MNIST}}$. La razón de esto es que ayuda a acelerar la convergencia del proceso de descenso del gradiente [25]. Así, mientras que en el conjunto de datos original las imágenes se encuentran en el conjunto $[0, 1]^{784}$, la versión normalizada no está restringida a este intervalo, y puede contener valores negativos. Se hicieron algunas pruebas y se corroboró este efecto.

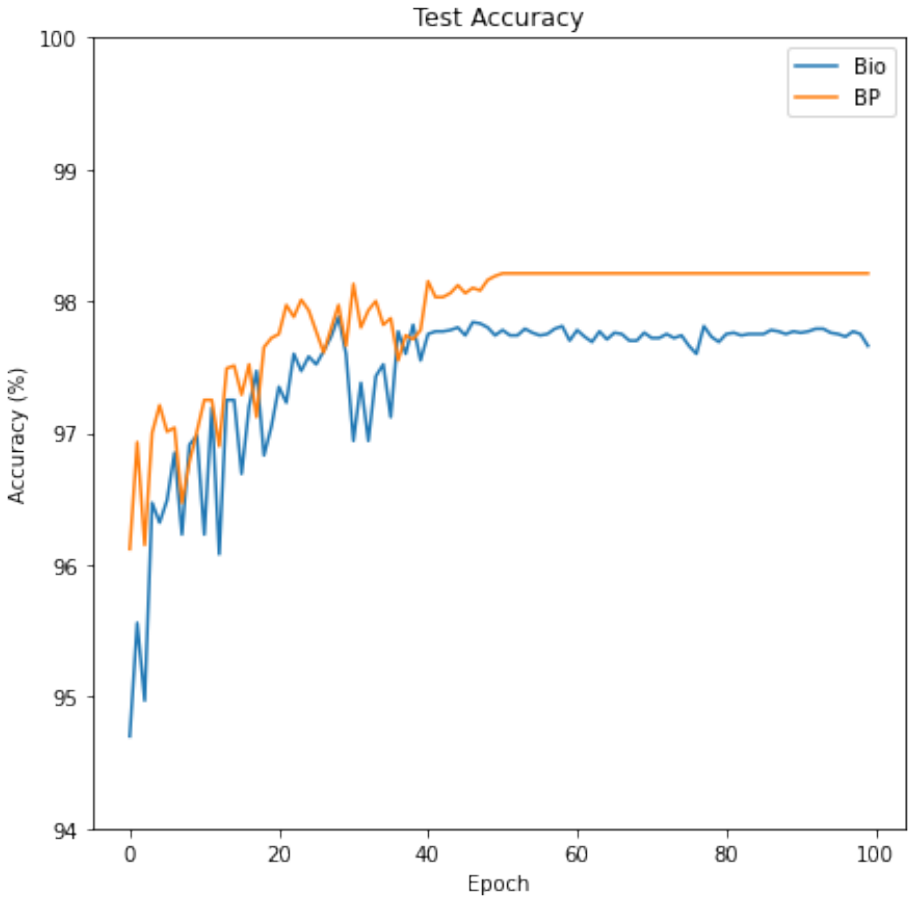
Esta arquitectura se implementó en el biblioteca PyTorch para *Deep Learning* en Python [38]. Esto para las dos versiones con el conjunto MNIST normalizado: una donde todos los pesos se inicializaron aleatoriamente y fue entrenada *end-to-end* con *backpropagation* (red BP); y la otra donde los pesos de la primera capa se entrenaron de acuerdo al método descrito arriba y luego se fijaron (excluyéndolos del proceso de diferenciación automática de PyTorch), mientras que los pesos de la segunda capa se inicializaron aleatoriamente y se entrenaron por backpropagation (red Bio). Ambos modelos se entrenaron usando GPUs en Google Colab, una herramienta digital de libre acceso para correr libretas de Jupyter en la nube. Los resultados fueron consistentes con lo descrito en el artículo original a pesar de que se modificó ligeramente la arquitectura [8]. Por ejemplo, se observó que la mayoría de los pesos aprendidos con el método bioinspirado adquirieron la forma de prototipos de dígitos del MNIST (fig. 2.1).

Figura 2.1: En la derecha se muestran 100 detectores de rasgos tomados al azar del modelo bioinspirado. En la izquierda 100 detectores del modelo entrenado con *backpropagation*. Claramente los detectores del modelo bioinspirado se asemejan más a dígitos.



También se observó que ambos modelos alcanzan precisiones comparables ($\sim$ 98 %) en el conjunto de prueba, con una diferencia de 0.5 % a favor del modelo BP (97.7 % en la red Bio vs. 98.2 % en la red BP; véase 2.2). En el conjunto de entrenamiento la red BP alcanza el 100 % de aciertos, mientras que el porcentaje de acierto de la red Bio llega a 99.7 %, ambas prácticamente se ajustan perfectamente al conjunto de entrenamiento.

Figura 2.2: Desarrollo de precisión respecto al conjunto de prueba a lo largo de 100 épocas de entrenamiento para ambos modelos.



3 Análisis de Robustez

Resumen

Una propiedad de gran interés en las redes neuronales artificiales es la de robustez, particularmente robustez a ataques adversarios. En este capítulo se discuten diferentes formas en las que se pueden definir formalmente los conceptos anteriores, y se discuten algunos métodos para generar ejemplos adversarios y poner a prueba la robustez de una red. También se presentan los resultados de algunos experimentos, donde se observó que las redes Bio presentadas en el capítulo anterior se desempeñaron de forma más robusta que una red con la misma arquitectura entrenada *end to end* con backpropagation. Además se observó que los pesos aprendidos para ambas redes tenían la propiedad de ocultar gradientes.

3.1. Ejemplos Adversarios

3.1.1. Definición de Robustez de una ANN

Como hemos mencionado antes, una propiedad de gran interés en las redes neuronales es la robustez, por lo cual entendemos que modificaciones pequeñas a una entrada (imperceptibles para nosotros) no son suficientes para modificar las categorías predichas por la red neuronal.

Para dar una definición más formal de esto primero definimos algunos otros conceptos. Sean Λ una ANN dada por $\eta : \mathbb{R}^{n(1)} \to \mathbb{R}^{n(N)}$ y $\|\cdot\|_p$ la p-norma en el espacio correspondiente con $p \in [1, \infty)$. Sea también $x \in \mathbb{R}^{n(1)}$ una entrada a la red tal que $\eta(x) = y \in \mathbb{R}^{n(N)}$ (usando las convenciones del capítulo 1). Supongamos también que la función de la red es categorizar entradas. Entonces existe un conjunto C finito (podemos suponer que $C = \{0, 1, 2, ..., k-1\}$) y una función $\kappa : \mathbb{R}^{n(N)} \to C$ que manda las salidas de la red a su categoría predicha. Tomamos $\hat{\eta} = \kappa \circ \eta$ y $\hat{y} = \hat{\eta}(x)$. Dado esto definimos una *perturbación adversaria* para x como un $r \in \mathbb{R}^{n(1)}$ tal que $\hat{\eta}(x+r) \neq \hat{y}$. Cabe observar que esta definición por si sola no es de mucho interés, pues siempre que $|C| > 1$ y la preimagen de al menos dos elementos de C sea no vacía toda entrada tendrá una perturbación adversaria. Solo se vuelve interesante la definición cuando la restringimos un poco: dado $\varepsilon > 0$, decimos que $r \in \mathbb{R}^{n(1)}$ es una *ε-perturbación adversaria* para x cuando es una perturbación adversaria para x y además $\|r\|_p < \varepsilon$. Así, decimos que Λ es *ε-robusta* en x si no existe una ε-perturbación adversaria para x; y decimos que Λ es *robusta* en x si existe una $\varepsilon > 0$ tal que Λ es ε-robusta en x. Sin embargo esta última definición es equivalente a preguntarnos si x se encuentra en la

frontera de la preimagen de la categoría correcta. También es claro que siguiendo en esta línea sería de poco interés definir robustez para una red de forma general, pues si definimos que Λ es robusta cuando es robusta en toda $x \in \mathbb{R}^{n(1)}$, esto es equivalente a decir que para cada $\hat{y} \in C$, el conjunto $\hat{\eta}^{-1}[\{\hat{y}\}]$ es abierto en $\mathbb{R}^{n(1)}$, y esto es imposible ya que las únicas funciones continuas de $\mathbb{R}^{n(1)}$ en un conjunto discreto son las constantes. Más bien podríamos decir que Λ es *robusta* si es robusta en todas las x que sean clasificables por algún humano como alguna de las categorías. De forma similar definimos que Λ sea *ε-rubusta*. En práctica estas x clasificables se toman como las que pertenecen a la distribución con la que se entrenó a la red (nosotros lo tomaremos como el conjunto de x que la red clasifica correctamente). Además, en vez de determinar dicotómicamente si una red dada es robusta o no, es de mayor utilidad dar una medida de la robustez de la red. Algunas formas comunes de hacer esto son las siguientes [39]:

$$\mathop{\mathbb{E}}_{(x,y)\in\mathcal{X}} [\, \sup\{C(\eta(x+r), y) \mid r \in \mathcal{B}_\varepsilon(x)\}\,]$$

es decir que también se puede tomar la robustez como la esperanza del valor máximo del costo en una bola de radio $\varepsilon > 0$ alrededor de cada entrada x con etiqueta correcta y en una distribución $\mathcal{X}$. Otra posible definición es:

$$\mathop{\mathbb{E}}_{(x,y)\in\mathcal{X}} [\, \text{ínf}\{\|r\|_p \mid r \in A_{x,y}\}\,]$$

donde $A_{x,y} := \{r \in \mathbb{R}^{n(1)} \mid \hat{\eta}(x) \neq \hat{\eta}(x+r)\}$. Es decir, la esperanza de la distancia entre cada entrada x y su ejemplo adversario más cercano. Ambas de estas funcionan a un nivel formal, pero en la práctica es imposible evaluar estas cantidades numéricamente en la mayoría de los casos, pues requieren encontrar supremos e ínfimos en bolas. Lo más que se puede hacer normalmente es aproximar estas cantidades. Para no complicar demasiado los cómputos de este trabajo se optó por una definición más simple:

$$\Xi_\varepsilon(\Lambda) := \mathop{\mathbb{E}}_{(x,y)\in\mathcal{X}} [\, A_{x,y} \cap \mathcal{B}_\varepsilon(x) = \emptyset\,]$$

es decir la esperanza de que para cada x, y en la distribución Λ sea ε-robusta en x.

3.1.2. Algunas Formas de Evaluar la Robustez

Aún con la última definición más simple, no siempre es posible computar la robustez de una red de esta forma. Evaluar si una red es ε-robusta para una entrada x particular requeriría ver si la bola de radio ε contiene algún ejemplo adversario, pero esto no se puede hacer para toda una bola de forma computacional. Para esto se han desarrollado diversos métodos para generar perturbaciones adversarias que funcionan como criterios para decidir sobre la ε-robustez de la red en x. Que todos estos métodos fallen en encontrar una perturbación es condición necesaria para afirmar que la red es

ε-robusta en x, más no es condición suficiente. Dicho esto, hasta la fecha no existe ninguna arquitectura ni defensa para redes neuronales que consistentemente evite todos los métodos conocidos, o que no implique una disyuntiva entre la robustez de la red y su desempeño [39]. Es una pregunta actual de gran interés si siquiera existe alguna defensa contra las perturbaciones adversarias o si son una característica intrínseca de las redes neuronales y/o los datos de entrenamiento [7]. En general estos métodos se pueden pensar como funciones $\Upsilon : \mathbb{R}^{n(1)} \to \mathbb{R}^{n(1)}$ tales que para cada entrada x, $\Upsilon(x)$ es el ejemplo adversario que propone el método (que puede ser o no ser de hecho un ejemplo adversario). Con esto podemos definir la ε-robustez de una red frente a un método Υ, como una modificación de la última definición, donde en vez de tomar la esperanza de que la red sea ε-robusta en las entradas, lo definimos como la esperanza de que $\Upsilon(x)$ no sea un ε-ejemplo adversario. En notación:

$$\Xi_\varepsilon(\Lambda \mid \Upsilon) := \mathop{\mathbb{E}}_{(x,y)\in\mathcal{X}} [\, \Upsilon(x) \notin A_{x,y} \,]$$

Uno de los métodos más simples y computacionalmente eficientes para generar perturbaciones adversarias es el Fast Gradient Sign Method (FGSM) [40]. Este método se basa en el supuesto de que η será aproximadamente lineal en una vecindad cercana a x, y define la perturbación como:

$$\Upsilon_{FGSM\varepsilon}(x) := \varepsilon \cdot \overline{sign}(\nabla_x C(\eta(x), y))$$

donde $sign : \mathbb{R} \to \mathbb{R}$ es la función

$$sign(a) := \begin{cases} 1 & \text{si a}>0 \\ -1 & \text{si a}<0 \\ 0 & \text{si a}=0 \end{cases}$$

y $\overline{sign}$ su versión aplicada entrada a entrada. Definida de esta forma estas perturbaciones tienen la propiedad de siempre estar sobre la bola de radio ε alrededor de x con la norma infinito. Sin embargo, este método no es muy riguroso, pues simplemente da un salto en una dirección basada en el gradiente. Existen métodos mucho más sofisticados que también se basan en el gradiente. Algunos ejemplos son: el método de signo del gradiente iterado se basa en el FGSM pero en vez de dar un único paso da varios pasos más pequeños [41]; el método L-BFGS de Szegedy *et al.* [42] encuentra ejemplos adversarios como solución a un problema de optimización acotada, es el primer método diseñado para este propósito; el método JSMA, a diferencia de los anteriores, está diseñado para encontrar ejemplos adversarios con la norma L_0, es decir encuentra un conjunto mínimo de pixeles que se pueden modificar para generar un ejemplo adversario [43]; el método *Deepfool* es otro método que encuentra ejemplos adversarios como solución a un problema de optimización y se desempeña mejor que L-BFGS, aunque también es más complejo de implementar [44]; Carlini & Wagner describen un método (CW) para las normas L_0, L_2 y L_∞ que se desempeña mejor que todas las anteriores, su aproximación también ve el problema como uno de optimización, pero usa una función objetivo distinta obtenida por prueba y error que

tiene varias propiedades favorables para este problema [45]; Madry *et al.* propusieron un método para generar ejemplos adversarios con la norma infinito basado en projected gradient descent (PGD) que en algunas condiciones se desempeña mejor que el método CW [46]. A parte de los mencionados arriba, se han desarrollado varios otros métodos.

3.1.3. Dos Formas de Explicar las Perturbaciones Adversarias

Existen varias explicaciones tentativas para la falta de robustez en redes neuronales BP. Dos ideas interesantes, que no son mutuamente excluyentes pero que reflejan dos aspectos del problema son: Por un lado que la falta de robustez de las redes se debe a su naturaleza lineal. En el mismo artículo donde proponen el método FGSM, Goodfellow, Shlens y Szegedy [40] discuten la posibilidad de que, a diferencia de como se creía al principio, los ejemplos adversarios no se deben a una falta de linealidad en los modelos de redes neuronales, sino precisamente lo opuesto, pues muestran cómo pequeñas perturbaciones, de norma infinito pequeña, pueden tener efectos muy grandes en modelos lineales cuando la dimensionalidad del problema es muy alta. Además, describen una posible defensa contra los ataques adversarios que llaman entrenamiento adversario ("adversarial training"), donde los ejemplos adversarios hallados para cada entrada del conjunto de entrenamiento son agregados al conjunto y usados para entrenar la red. La idea es que entonces la red estará siendo entrenada para reconocer los ejemplos más difíciles de las vecindades cercanas de las entradas. En efecto encontraron un desempeño considerablemente mejor en redes entrenadas de esta forma [40]. Esta sigue considerándose una de las mejores defensas actualmente, y una de las pocas que se desempeñan bien frente a ataques fuertes [47, 48]. Por otro lado está la idea comúnmente citada como "adversarial examples are not bugs, they are features". Esta idea, propuesta por Ilyas *et al.* [7], dice que los ejemplos adversarios son rasgos de los datos que no son reconocibles por humanos, pero que si tienen un poder predictivo que los modelos aprovechan. Según esto hay una tensión entre el desempeño de las redes y su robustez, pues pedirle a una red que sea robusta sería restringir los rasgos que puede utilizar para resolver un problema a aquellos que son reconocibles por un humano como rasgos de una categoría o grupo. De hecho describen un método para robustificar un conjunto de datos, haciendo que contenga (en la medida posible) únicamente rasgos robustos, y por lo tanto que una red clásica entrenada sobre este conjunto sea más robusta simplemente por el hecho de que los rasgos no robustos fueron removidos de los datos, haciendo que la red ya no pueda aprender a reconocerlos. Más aún, demostraron que es posible entrenar una red sobre imágenes cuya única señal predictiva son rasgos no robustos, y aún así obtener un buen desempeño.

Así, por un lado tenemos una forma de explicar los ejemplos adversarios como una característica intrínseca de los modelos, mientras que por otro como una característica de los datos usados para entrenarlo. A final de cuentas si el problema de los ataques adversarios reside en cierta medida en el hecho de que las redes aprenden a hacer uso de rasgos no robustos de los datos, esto ya indica una brecha entre el aprendizaje biológico y el aprendizaje BP, pues las redes biológicas parecen extraer los rasgos

robustos de la experiencia durante el aprendizaje. Con esto en mente las soluciones propuestas para robustificar las redes, como el entrenamiento adversario y la elaboración de conjuntos de datos robustificados, a pesar de ser efectivos y útiles, resultan insatisfactorios desde un punto de vista teórico, pues se esperaría que exista un método de aprendizaje tal que naturalmente aprenda un modelo robusto. La justificación de que tal método debe existir es que es lo que parece suceder en las redes biológicas. Desde un punto de vista matemático se puede demostrar que estas soluciones robustas y precisas si existen, pero los métodos actuales no las encuentran [49].

3.1.4. Rasgos no Robustos y Red Bioinspirada

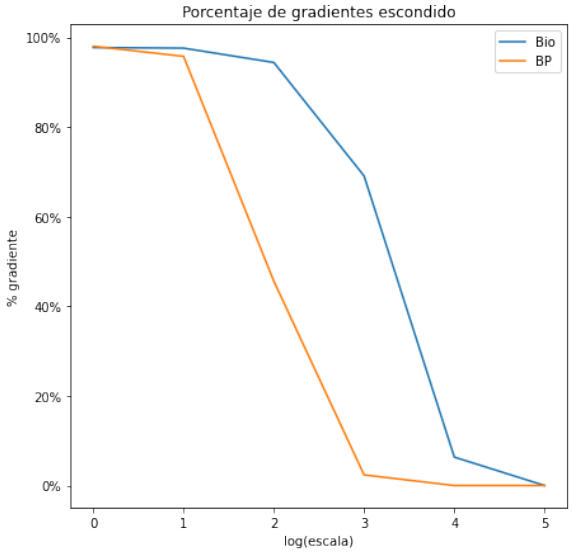
Siguiendo la definición dada en [7], decimos que un *rasgo* es cualquier función $f : \mathbb{R}^{n(1)} \to \mathbb{R}$. Los rasgos útiles para una tarea de clasificación son aquellos donde existe una correlación positiva entre el valor del rasgo y que la categoría correcta sea alguna particular (en tal caso decimos que ese rasgo es predictivo de la categoría), pues son estos los que pueden ser utilizados exitosamente como entradas de un clasificador lineal. Para ANN clasificadoras notamos que las activaciones de las neuronas de la penúltima capa definen un conjunto de rasgos, que luego son pasados por un clasificador lineal para dar la clasificación final de la red. Es decir que podemos ver el proceso de aprendizaje de la red en dos partes: la convergencia de las activaciones de la penúltima capa (por medio de los pesos) a rasgos que son útiles para la tarea de clasificación; y la convergencia de los pesos de la última capa a unos que aprovechen los rasgos de la penúltima capa para hacer la clasificación. Así, el modelo bio-inspirado de Krotov se puede ver como una modificación del régimen de entrenamiento estándar donde los dos aspectos del aprendizaje no se dan simultáneamente, sino más bien primero se construyen los rasgos útiles a partir de los datos, y solo después se integran a un clasificador lineal que aprende la mejor manera de usar esos rasgos para hacer la clasificación. Como la arquitectura usada únicamente tiene una capa escondida, los rasgos aprendidos se pueden visualizar por medio del vector de pesos entrantes a cada neurona de la capa escondida, dándonos una idea de cómo se ven los rasgos que están siendo utilizados para la tarea de clasificación (fig. 2.1). En el caso Bio lo que se observa es que los rasgos que detecta la red muestran una gran similaridad con entradas de la distribución que se busca aprender. Estos rasgos intuitivamente parecen ser del tipo robusto, pues pequeñas perturbaciones en las entradas de la distribución difícilmente invertirán la correlación entre un rasgo y la etiqueta correcta de una entrada perturbada; y de hacerlo esperaríamos que la perturbación tuviera una forma reconocible. Basado en lo anterior y en la observación de Ilyas *et al.* [7] de que parte de la falta de robustez de las redes se debe a la detección de rasgos no robustos, hipotetizamos que las redes Bio serán más robustas a perturbaciones adversarias que las redes BP. Además esperaríamos que las perturbaciones adversarias encontradas para el modelo Bio tengan una forma geométrica reconocible (a diferencia de lo que pasa con las perturbaciones adversarias en modelos BP).

3.2. Resultados y Discusión

Se pusieron a prueba las robusteces de las redes Bio y BP con el entrenamiento descrito en el capítulo anterior para ver si existía alguna diferencia. El primer resultado interesante del proceso de entrenamiento por backpropagation con esta arquitectura es que tanto la red Bio como la BP aprendieron pesos sinápticos tales que esconden el gradiente de la función de costo respecto a los valores de entrada en casi todos los ejemplos del conjunto de prueba. Es decir, cuando se intentó evaluar $\nabla_x C$ en los ejemplos de prueba usando el diferenciador automático de PyTorch, en casi todos los casos el resultado fue el vector cero. Esto tuvo como consecuencia que al aplicar el método FGSM a las redes, este parecía ser casi totalmente inefectivo. Esta observación curiosamente no la encontramos mencionada en ninguno de los artículos que discuten esta arquitectura [8, 36, 37]. Este efecto parece deberse a la aplicación de la función $RePU_{4.5}$, la cual hace que las activaciones de la capa escondida crezcan enormemente, resultando en activaciones en la capa de salida de varios órdenes de magnitud de diferencia. Luego al pasar estas activaciones de salida por la función SOFTMAX se produce una distribución que, de acuerdo a cómo está implementada la función en PyTorch y a las limitaciones de la aritmética de punto flotante, es simplemente un vector de ceros con un único valor unitario en el índice predicho por la red, y correspondientemente el valor de la función de costo CE se evalúa a cero. Además, hay una vecindad del punto donde aplica el mismo redondeo de la capa SOFTMAX. Es decir que esta arquitectura lleva a la red a decidirse con total seguridad por una única categoría, y esto tiene el efecto de que el gradiente en todos los ejemplos que la red identifica con seguridad alta es tan pequeño que se toma como un vector constante cero. Una consecuencia de lo anterior es que cualquier método para generar ejemplos adversarios basado en el gradiente fallará en encontrarlos para estos puntos. Así, esta arquitectura por si misma funciona como una defensa natural contra métodos para generar ejemplos adversarios basados en el gradiente directamente. Esta defensa tiene mucho en común con la defensa conocida como *defensive distillation* [50], cuya idea es entrenar a una red de forma normal pero haciendo un reescalamiento en la última capa para forzar a la red a generar activaciones grandes (sin el escalamiento); luego las activaciones de salida de la red se usan para entrenar a otra red del mismo tamaño y con el mismo reescalamiento en la capa final. Finalmente para evaluar ejemplos de prueba se quita el reescalamiento. Este método se propuso originalmente como una defensa general contra los ejemplos adversarios [50, 51]. Sin embargo más adelante se mostró que la destilación defensiva únicamente tenía el efecto de ocultar los gradientes de una forma similar a la que mencionamos arriba, y que con unos simples reajustes en los métodos para generar ejemplos adversarios era posible generar perturbaciones adversarias de tamaños comparables con los de una red normal [45]. Una forma simple para romper esta defensa es hacer un reescalamiento adecuado en la capa de salida, antes de aplicar la función SOFTMAX, de tal manera que los gradientes ya no se desvanezcan. En el caso de las redes estudiadas en este trabajo, se requiere un reescalamiento de $T = 10^5$ antes de que tanto la red BP como la Bio muestren gradientes distintos de cero en todos los puntos correspondientes a las

imágenes de prueba. Además es notable que el porcentaje de imágenes para las cuales la red esconde el gradiente cae bastante más rápido en la red BP que en la Bio (véase 3.1), esto probablemente se debe a que la red Bio aprende pesos que llevan a activaciones considerablemente más grandes en la capa de salida.

Figura 3.1: Se muestra en la abscisa el factor de reescalamiento en escala logarítmica, y en la ordenada la proporción del total de entradas en el conjunto de prueba para las cuales el cálculo de gradientes devuelve el vector cero.



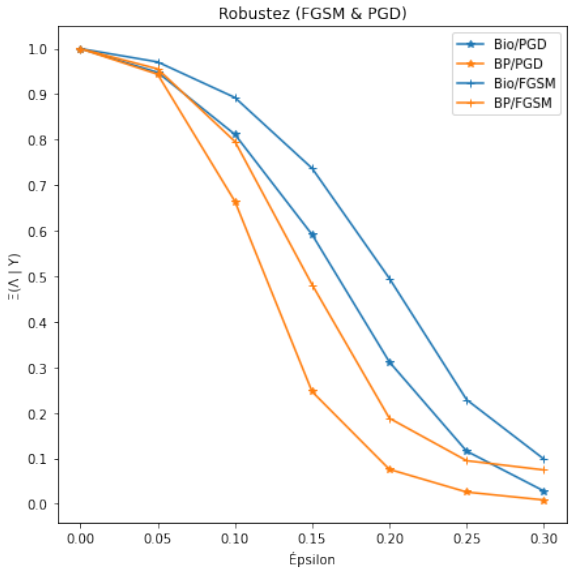
Para probar la robustez de las redes frente a los diferentes métodos se usó el paquete *torchattacks* de PyTorch. En este están implementados la mayoría de los métodos actuales para generar ejemplos adversarios. Sin embargo las funciones de este paquete esperan recibir redes que toman entradas con valores en el intervalo $[0, 1]$, y devuelven ejemplos adversarios con valores que permanecen en este intervalo. Como hemos mencionado antes, para el entrenamiento de los modelos se usó una versión normalizada del MNIST. Para que los modelos cumplan el supuesto de arriba se compuso cada modelo con una capa de normalización $\mathcal{N}$ que manda las imágenes del MNIST original a su versión normalizada con la ecuación que se dió en el capítulo 2. Esta operación se toma en cuenta cuando el diferenciador automático calcula los gradientes, y por lo tanto es compatible con los métodos implementados en *torchattacks*.

Usando el valor de 10^5 para dividir el vector de salida de las redes, de manera que en ambos sea posible calcular los gradientes en cada uno de los ejemplos, se obtuvo la curva que se muestra en la figura 3.2 para el método FGSM descrito arriba (en la implementación del paquete *torchattacks* para PyTorch). En ella se puede observar que el desempeño de la red Bio cae más lento que en la BP, lo cual indica que la red Bio es más robusta al método FGSM en estas condiciones. Ambas redes eventualmente caen debajo del 10 % de acierto, quedando más bajo que si seleccionara categorías al azar. Lo anterior debe interpretarse con cuidado, pues está sujeto a la elección del factor de escalamiento, el cual no necesariamente es el óptimo para cada red. Una hipótesis que surgió de lo anterior es que tal vez debido a que la red Bio requiere de

un factor de escalamiento mayor para poder calcular todos los gradientes, también requiere un factor mayor para que el método FGSM funcione adecuadamente. Sin embargo algunas pruebas con factores mayores a 10^5 resultaban más bien en un mejor desempeño de la red Bio.

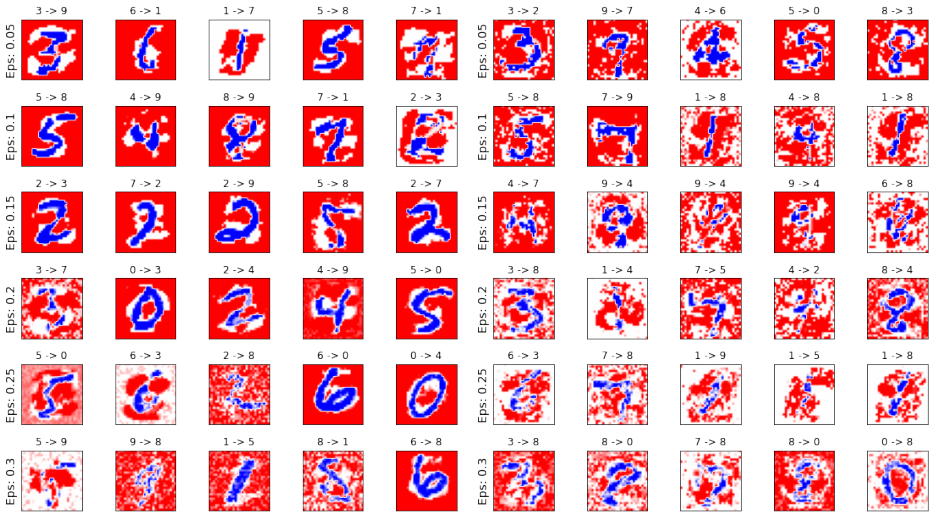
De igual manera se revisó el desempeño de ambas redes frente al método PGD con norma infinito para generar ejemplos adversarios de norma ε [46]. Este método es mucho más fuerte que el FGSM pues no usa como supuesto la linealidad del modelo [46]. Para estudiar la robustez de los modelos frente a este método se generaron dos curvas con los valores de éxito de los modelos para predecir las etiquetas correctas del conjunto de prueba al variar el parámetro ε (los otros parámetros propios del método se mantuvieron fijos: $alpha = 2/255$, 40 pasos, e inicialización aleatoria), los resultados se muestran en la figura 3.2. Frente a este segundo método la diferencia en robustez se hace incluso más notable. Para $\varepsilon = 0.15$ la diferencia en el porcentaje de acierto de las redes es de 33.6 % (57.8 % vs 24.2 %). Es notable que la robustez de la red Bio contra el método PGD sigue siendo mayor que la de la red BP contra el método FGSM que es más débil.

Figura 3.2: En la abscisa se representa el valor de la variable ε, que determina el tamaño de la perturbación. En la ordenada se muestra la robustez ($\Xi_\varepsilon(\Lambda \mid \Upsilon)$) de las redes en la tarea de clasificar los ejemplos del conjunto de prueba contra los métodos FGSM y PGD.



También resulta ilustrativo ver la forma de las perturbaciones encontradas. Algo interesante que se puede notar es que en ambos casos para la mayoría de los ejemplos es posible predecir la etiqueta original de ver la perturbación adversaria. Esto parecería indicar que al menos en parte las perturbaciones funcionan borrando el trazo original de los dígitos. Sin embargo, las perturbaciones generadas para el modelo Bio son mucho más claras y parecen contener mucho menos ruido al compararlas con las generadas para el modelo BP (fig. 3.3).

Figura 3.3: Se muestran algunas perturbaciones exitosas obtenidas por el método PGD tomadas aleatoriamente. Arriba de cada perturbación se muestra la etiqueta original seguida de la etiqueta predicha después de la perturbación. Las primeras cinco columnas de la izquierda corresponden a las perturbaciones del modelo Bio y las de las derecha al modelo BP.



3.3. Conclusión

Los resultados presentados arriba sugieren que las redes entrenadas con el método de Krotov y Hopfield [8] definido en el capítulo 2 son más robustas que redes con la misma arquitectura entrenadas *end-to-end* con backpropagation. Esto parecería deberse a la forma de los rasgos que la red aprende a reconocer durante la etapa no supervisada del entrenamiento. Aún así, el incremento en la robustez no es suficiente para decir que la red es robusta. Más investigación se podría hacer sobre la robustez de los rasgos aprendidos por la red Bio. Además, sería interesante investigar posibles modificaciones al modelo de Krotov que promuevan ciertas propiedades adicionales. Por ejemplo se podrían probar diferentes formas de interacción lateral, o incluso recurrente, de las neuronas. También podría ser interesante traducir este modelo al dominio de las Spiking Neural Networks (SNN), una familia de modelos con una dinámica más cercana a la biológica [52]. Tal vez dentro del contexto de los modelos SNN sería posible investigar reglas de aprendizaje más cercanas a la regla originalmente propuesta por Krotov y Hopfield en términos de ecuaciones diferenciales. Más aún, en el espíritu de modelos biológicamente plausibles, sería interesante estudiar las propiedades de estas redes cuando las hacemos más profundas, es decir, cuando el número de capas ocultas se vuelve mayor a 1, pues en las redes biológicas es claro que la profundidad juega un papel importante en el procesamiento de la información. Por otro lado, investigar el desempeño de estas redes frente a otros problemas podría ayudar a entender mejor las limitaciones de los mismos, así como poner en contexto sus virtudes en relación a otros modelos.

De manera general, más investigación acerca de métodos de aprendizaje alterna-

tivos, y en particular sobre métodos bioinspirados, podría ser de gran importancia. Pues además de ser un punto de contacto entre el dominio de las redes neuronales artificiales y el de la neurociencia, permitiendo a esta última generar y poner a prueba modelos del funcionamiento cerebral, también podría ser importante en la búsqueda de la solución al problema de los ejemplos adversarios en ANN.

Bibliografía

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] A. H. Marblestone, G. Wayne, and K. P. Kording, "Toward an integration of deep learning and neuroscience," *Frontiers in computational neuroscience*, vol. 10, p. 94, 2016.

[5] D. Heaven, "Deep trouble for deep learning," *Nature*, vol. 574, no. 7777, pp. 163–166, 2019.

[6] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, pp. 828–841, 2019.

[7] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," *arXiv preprint arXiv:1905.02175*, 2019.

[8] D. Krotov and J. J. Hopfield, "Unsupervised learning by competing hidden units," *Proceedings of the National Academy of Sciences*, vol. 116, no. 16, pp. 7723–7731, 2019.

[9] J. Jozefowiez, *Associative Learning*, pp. 330–334. Boston, MA: Springer US, 2012.

[10] E. G. Jones, "Santiago ramón y cajal and the croonian lecture, march 1894," *Trends in neurosciences*, vol. 17, no. 5, pp. 190–192, 1994.

[11] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[12] G. Mongillo, *Hebbian Learning*, pp. 1417–1419. Boston, MA: Springer US, 2012.

[13] T. V. Bliss and T. Lømo, "Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path," *The Journal of physiology*, vol. 232, no. 2, pp. 331–356, 1973.

[14] A. Citri and R. C. Malenka, "Synaptic plasticity: multiple forms, functions, and mechanisms," *Neuropsychopharmacology*, vol. 33, no. 1, pp. 18–41, 2008.

[15] E. L. Bienenstock, L. N. Cooper, and P. W. Munro, "Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex," *Journal of Neuroscience*, vol. 2, no. 1, pp. 32–48, 1982.

[16] L. N. Cooper and M. F. Bear, "The bcm theory of synapse modification at 30: interaction of theory with experiment," *Nature Reviews Neuroscience*, vol. 13, no. 11, pp. 798–810, 2012.

[17] L. N. Cooper, F. Liberman, and E. Oja, "A theory for the acquisition and loss of neuron specificity in visual cortex," *Biological cybernetics*, vol. 33, no. 1, pp. 9–28, 1979.

[18] T. N. Karaminis and M. S. C. Thomas, *Connectionist Theories of Learning*, pp. 771–774. Boston, MA: Springer US, 2012.

[19] N. Caporale and Y. Dan, "Spike timing–dependent plasticity: a hebbian learning rule," *Annu. Rev. Neurosci.*, vol. 31, pp. 25–46, 2008.

[20] E. M. Izhikevich and N. S. Desai, "Relating stdp to bcm," *Neural computation*, vol. 15, no. 7, pp. 1511–1523, 2003.

[21] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[22] H. A. Mallot, *Computational Neuroscience*, vol. 486. Springer, 2013.

[23] A. Hernandez-Garcia, "Rethinking supervised learning: insights from biological learning and from calling it by its name," *arXiv preprint arXiv:2012.02526*, 2020.

[24] J. C. Whittington and R. Bogacz, "Theories of error back-propagation in the brain," *Trends in cognitive sciences*, vol. 23, no. 3, pp. 235–250, 2019.

[25] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[26] N. Kriegeskorte, "Deep neural networks: a new framework for modeling biological vision and brain information processing," *Annual review of vision science*, vol. 1, pp. 417–446, 2015.

[27] D. G. Barrett, A. S. Morcos, and J. H. Macke, "Analyzing biological and artificial neural networks: challenges with opportunities for synergy?," *Current opinion in neurobiology*, vol. 55, pp. 55–64, 2019.

[28] B. Illing, W. Gerstner, and J. Brea, "Biologically plausible deep learning—but how far can we go with shallow networks?," *Neural Networks*, vol. 118, pp. 90–101, 2019.

[29] C. Pehlevan and D. B. Chklovskii, "Optimization theory of hebbian/anti-hebbian networks for pca and whitening," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1458–1465, IEEE, 2015.

[30] A. Meulemans, M. T. Farinha, J. G. Ordóñez, P. V. Aceituno, J. Sacramento, and B. F. Grewe, "Credit assignment in neural networks through deep feedback control," *arXiv preprint arXiv:2106.07887*, 2021.

[31] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature communications*, vol. 7, no. 1, pp. 1–10, 2016.

[32] S. Bartunov, A. Santoro, B. A. Richards, L. Marris, G. E. Hinton, and T. Lillicrap, "Assessing the scalability of biologically-motivated deep learning algorithms and architectures," *arXiv preprint arXiv:1807.04587*, 2018.

[33] A. M. Zador, "A critique of pure learning and what artificial neural networks can learn from animal brains," *Nature communications*, vol. 10, no. 1, pp. 1–7, 2019.

[34] E. Oja, "Simplified neuron model as a principal component analyzer," *Journal of mathematical biology*, vol. 15, no. 3, pp. 267–273, 1982.

[35] C. Fyfe, *Hebbian learning and negative feedback networks*. Springer Science & Business Media, 2007.

[36] D. Krotov and J. J. Hopfield, "Dense associative memory for pattern recognition," *Advances in neural information processing systems*, vol. 29, pp. 1172–1180, 2016.

[37] D. Krotov and J. Hopfield, "Dense associative memory is robust to adversarial inputs," *Neural computation*, vol. 30, no. 12, pp. 3151–3167, 2018.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[39] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *arXiv preprint arXiv:1902.06705*, 2019.

[40] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[41] A. Kurakin, I. Goodfellow, S. Bengio, *et al.*, "Adversarial examples in the physical world," 2016.

[42] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[43] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European symposium on security and privacy (EuroS&P)*, pp. 372–387, IEEE, 2016.

[44] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582, 2016.

[45] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*, pp. 39–57, IEEE, 2017.

[46] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[47] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein, "Adversarial training for free!," *arXiv preprint arXiv:1904.12843*, 2019.

[48] E. Wong, L. Rice, and J. Z. Kolter, "Fast is better than free: Revisiting adversarial training," *arXiv preprint arXiv:2001.03994*, 2020.

[49] A. Bastounis, A. C. Hansen, and V. Vlačić, "The mathematics of adversarial attacks in AI – Why deep learning is unstable despite the existence of stable neural networks," *arXiv e-prints*, p. arXiv:2109.06098, Sept. 2021.

[50] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE symposium on security and privacy (SP)*, pp. 582–597, IEEE, 2016.

[51] N. Papernot and P. McDaniel, "On the effectiveness of defensive distillation," *arXiv preprint arXiv:1607.05113*, 2016.

[52] W. Gerstner and W. M. Kistler, *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

A Código del modelo

```
import torch
import torchvision

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import torchattacks

import numpy as np
np.random.seed(0)
from numpy import linalg as LN


import pickle

#####
## Código para entrenar de forma no supervisada los detectores de
    razgos para la red Bio
#####

# Definimos una clase para las redes porque resulta conveniente no
    repetir cachos de código. Para entrenar los pesos usamos el
    método hebbLearn.
class bioNN:
    def __init__(self,neuroNum,inSize,outSize,p=2,delta=.4,k=2,tau
        =.02,mu=0.0,sigma=1.0):
        #Add the output layer to neuroNum. The output layer must
            have the same size as the expected output vectors.
        self.neuroNum=neuroNum+[outSize]
        self.layerNum=len(neuroNum)+1
        self.inSize=inSize
        self.p=p
        self.delta=delta
        self.k=k
        self.tau=tau
```

```
        self.W=[np.random.normal(mu, sigma,(m,n)) for n,m in list(
            zip([inSize]+neuroNum, self.neuroNum))]

    # Aquí se implementa casi exactamente el algorítmo de Krotov
        (2019).
    def hebbLearn(self,epochs,train_loader,Nmini=100):
        Nnn=self.neuroNum[0]
        for ep in range(epochs):
            # La velocidad de aprendizaje va bajando a medida que
                avanza.
            speed = self.tau*(1-ep/epochs)
            for batch_idx, (input, target) in enumerate(
                train_loader):
                trainBatch = np.transpose(torch.flatten(input,1).
                    numpy())

                # Calculamos las activaciones linealmente
                aa=np.dot(self.W[0]*np.abs(self.W[0])**(self.p-2),
                    trainBatch)

                # nnSrt es una matriz con los índices de las
                    activaciones ordenados de menor a mayor (por
                    columnas)
                nnSrt=np.argsort(aa,axis=0)

                # gres es el resultado de aplicar la función g(i)
                    por columnas.
                gres=np.zeros((Nnn,Nmini))
                gres[nnSrt[Nnn-1],np.arange(Nmini)]=1.0
                gres[nnSrt[Nnn-self.k],np.arange(Nmini)]=-self.
                    delta

                avgaa=np.sum(gres*aa,axis=1).reshape(Nnn,1)

                dW=np.dot(gres,np.transpose(trainBatch))-avgaa*
                    self.W[0]
                maxval=np.amax(np.abs(dW))

                # Acotamos el valor de normalización
                if maxval<1e-10:
                    maxval=1e-10

                # Actualizamos los valores con una forma
                    normalizada de dW (haciendo que el valor
                    absoluto de sus entradas sea menor o igual a 1)
                self.W[0]=self.W[0]+speed*dW/maxval
```

```
            if ep%10==0:
                print(ep)

#####
## Código para entrenar modelos en PyTorch
#####

torch.backends.cudnn.enabled = False # Más adelante pasaré a GPU
torch.manual_seed(0)

# Cargamos conjunto de datos MNIST (normalizado). Devuelve dos "
    data loaders", uno para el conjunto de entrenamiento y otro
    para el conjunto de prueba.
def load_mnist(batch_size_train, batch_size_test, normalize = True
    ):
    mnist_data_dir = '/Users/ianxul/Google\ Drive/Neuro/Codigo\ Py
        /data/mnist' # Ubicación de los datos MNIST

    if normalize:
        mean = 0.1307 # MNIST mean
        std = 0.3081 # MNIST std
    else:
        mean = 0.
        std = 1.

    mnist_train_data = torchvision.datasets.MNIST(mnist_data_dir,
        transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean, std)
        ]),
        train = True, download = True)

    mnist_train_loader = torch.utils.data.DataLoader(
        mnist_train_data,
            batch_size=batch_size_train,
            shuffle=True,
            num_workers=1)

    mnist_test_data = torchvision.datasets.MNIST(mnist_data_dir,
        transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean, std)
        ]),
        train = False, download = True)

    mnist_test_loader = torch.utils.data.DataLoader(
        mnist_test_data,
```

```
            batch_size=batch_size_test,
            shuffle=True,
            num_workers=1)

    return mnist_train_loader, mnist_test_loader

# Definimos la clase de redes con una sola capa oculta, y con la
    arquitectura descrita por Krotov & Hopfield.
class L2Net(nn.Module):
    def __init__(self, hidden, alpha = 4.5, beta = 0.01, insize =
        784, outsize = 10):
        super(L2Net, self).__init__()
        # Primera capa "fully connected"
        self.fc1 = nn.Linear(insize, hidden, bias = False)
        # Parámetro para la función RePU
        self.alpha = alpha
        # Segunda capa "fully connected"
        self.fc2 = nn.Linear(hidden, outsize, bias = False)
        # Parámetro de escalamiento de la última capa
        self.beta = beta

    # Propagación hacia delante de las activaciones. Devuelve las
        activaciones de la última capa
    def forward(self, x):
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = x ** self.alpha
        x = self.beta * self.fc2(x)

        return x

# Esta clase es útil para probar los ataques, ya que la biblioteca
    "torchattacks" espera que estén en el intervalo "[0,1]". Para
    hacer los ataques hacemos la composición de la red L2Net con
    las dos de abajo.
# Funciona como un filtro para normalizar datos
class Normalize(nn.Module) :
    def __init__(self, mean, std) :
        super(Normalize, self).__init__()
        self.register_buffer('mean', torch.Tensor(mean))
        self.register_buffer('std', torch.Tensor(std))

    def forward(self, x):
        return (x - self.mean) / self.std

# Simplemente realiza un escalamiento uniforme del vector de
    entrada.
```

```
class Scale(nn.Module) :
    def __init__(self, scale) :
        super(Scale, self).__init__()
        self.scale = scale

    def forward(self, x):
        return (x/self.scale)



# Función para correr una época de entrenamiento. Esta se corre
    por cada época
def train(network,train_loader, optimizer, loss_fn, epoch, device,
    log_interval):
    network.train()
    correct = 0
    for batch_idx, (input, target) in enumerate(train_loader):
        input, target = input.to(device), target.to(device)
        optimizer.zero_grad() ## Vuelve a asignar el valor de cero
            al gradiente acumulado
        output = network(input) # Calculamos activaciones ff
        loss = loss_fn(output, target) # Función de costo
        loss.backward() # Se calculan gradientes con
           backpropagation
        optimizer.step() # Damos un paso en el algoritmo del
           optimizador

        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).sum()

        # Reportar progreso y guardar cambios
        if batch_idx %log_interval == 0:
            print('Época de entrenamiento: {} [{}/{} ({:.0f}%)]\t
               Costo: {:.6f}'.format(
                epoch, batch_idx * len(input), len(train_loader.
                   dataset),
                100. * batch_idx / len(train_loader), loss.item())
                   )

            torch.save(network.state_dict(), 'results/model.pth')
            torch.save(optimizer.state_dict(), 'results/optimizer.
               pth')

    # Devuelve la precisión y el costo actuales
    return [(100. * correct/len(train_loader.dataset)).item(),loss
       .item()]
```

```
# Función para probar la red contra el conjunto de datos de prueba
    (aquí no se entrena la red).
def test(network, loss_fn, test_loader, device):
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for input, target in test_loader:
            input, target = input.to(device), target.to(device)
            output = network(input)
            test_loss += loss_fn(output, target).item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
    test_loss /= len(test_loader.dataset)
    print('\nCosto prueba: {:.4f}, Acc. prueba: {}/{} ({:.0f}%)\n'
        .format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
    return [(100. * correct / len(test_loader.dataset)).item(),
        test_loss]


## "train_routine" define una rutina de entrenamiento y entrena la
    red. Los parámetros que toma son:
# N : número de neuronas en la capa oculta.
# n_epochs : Una lista con los números de épocas a entrenar la red
    .
# learning_rate : Una lista del mismo tamaño que "n_epochs" donde
    cada entrada es la velocidad a la que se entrenará durante esas
    épocas.
# alpha : Parámetro de la función RePU
# beta : Parámetro de escalamiento
# freeze : Booleano que indica si la primera matriz de pesos está
    fija o no.
# fixW : En caso de ser "freeze" verdadero, este parámetro
    contiene a la matriz que se mantendrá fija.
# device : Determina si se entrenará en el cpu o gpu (cpu por
    default).
# batch_size_train : Tamaño de los batches para el entrenamiento.
# log_interval : Cada cuántas iteraciones se reportan resultados.
# test_interval : Cada cuántas iteraciones de entrenamiento se
    reporta desempeño en el conjunto de prueba.

def train_routine(N = 50, n_epochs = [100,50,50,50,50],
    learning_rate = [0.001, 0.0005, 0.0001, 0.00005, 0.00001],
    alpha = 4.5, beta = 0.01, freeze = False, fixW = None, device =
```

```
    torch.device('cpu'), batch_size_train = 100, batch_size_test =
    100, log_interval = 1000, test_interval = 10):
    # Creamos una instancia de nuestra red
    network = L2Net(N, alpha, beta)
    loss_fn = F.cross_entropy

    # Fijamos los pesos si es el caso
    if freeze:
        network.fc1.weight = torch.nn.Parameter(fixW)

    # Mandamos a GPU si es el caso
    network = network.to(device)

    # Cargamos datos
    train_loader, test_loader = load_mnist(batch_size_train,
       batch_size_test)

    # Estas listas guardan datos a lo largo del entrenamiento para
        dar seguimiento
    train_track = []
    test_track = []
    train_track_file = open("results/train_track.pkl", "wb")
    test_track_file = open("results/test_track.pkl", "wb")


    test_track.append(test(network, loss_fn, test_loader, device))
    for ii in range(len(n_epochs)):
        for epoch in range(1, n_epochs[ii] + 1):
            optimizer = optim.Adam(network.parameters(), lr=
               learning_rate[ii])

            if freeze:
                network.fc1.weight.requires_grad = False

            train_track.append(train(network, train_loader,
               optimizer, loss_fn, epoch, device, log_interval))
            if epoch % test_interval == 0:
                test_track.append(test(network, loss_fn,
                   test_loader, device))


    pickle.dump(train_track,train_track_file)
    pickle.dump(test_track, test_track_file)

#####
##  Código para realizar ataques
#####
```

```
# Una función para checar la proporción de gradientes que se puede
    calcular (que no da 0)
def check_gradients(model, data_loader, loss_fn = nn.
    CrossEntropyLoss(), scale = 1):
    model.eval()
    count_hasgrad = 0
    count_tot = 0
    for images, labels in data_loader:
        model.zero_grad()
        images.requires_grad = True

        output = model(images)
        loss = loss_fn(output/scale, labels)
        loss.backward()
        gradC_x = images.grad
        count_hasgrad += (gradC_x.sum().item() != 0)
        count_tot += 1

    return count_hasgrad, count_hasgrad/count_tot

# Una función para simplificar el uso del paquete torchattack
def torch_attack(model, atk_type, scale, p1 = False, p2 = False,
    report = False):

    # Se define el modelo como composición de tres redes. La
        primera normaliza los datos, la segunda es el modelo en si,
        la tercera un factor de escalamiento
    m = nn.Sequential(Normalize([0.1307], [0.3081]), model, Scale(
        scale))

    _, test_loader = load_mnist(1,1, normalize = False)

    # Ataques usados
    if (atk_type == "FGSM") & (p1 != False):
        atk = torchattacks.FGSM(m, eps=p1)
    elif (atk_type == "PGD") & (p1 != False):
        print("Ataque PGD con eps = {}".format(p1))
        atk = torchattacks.PGD(m, eps = p1, alpha = 2/255, steps =
            40, random_start = True)
    else:
        print("Ataque vainilla")
        atk = torchattacks.VANILA(m)

    total = 0
    correct = 0
    pert_ejs = []
```

```
for images, labels in test_loader:
    if torch.max(m(images).data, 1)[1].item() == labels.item()
       :
        pert_images = atk(images, labels)

        outputs = m(pert_images)

        _, predicted = torch.max(outputs.data, 1)

        if labels.item()!=predicted.item():
            pert_ejs += [[images[0,0].detach().numpy(), ((
               pert_images-images)[0,0]).detach().numpy(),
               labels.item(), predicted.item()]]

        total += 1

        correct += (predicted.item() == labels.item())

        if report and total%100 == 0:
            print("Correct: {}; Total: {}".format(correct,
               total))


final_acc = (correct/total)

print("Par 1: {}, Par 2: {}\tPrecisión de prueba = {} / {} =
   {}".format(p1,p2, correct, total, final_acc))

return final_acc, pert_ejs
```