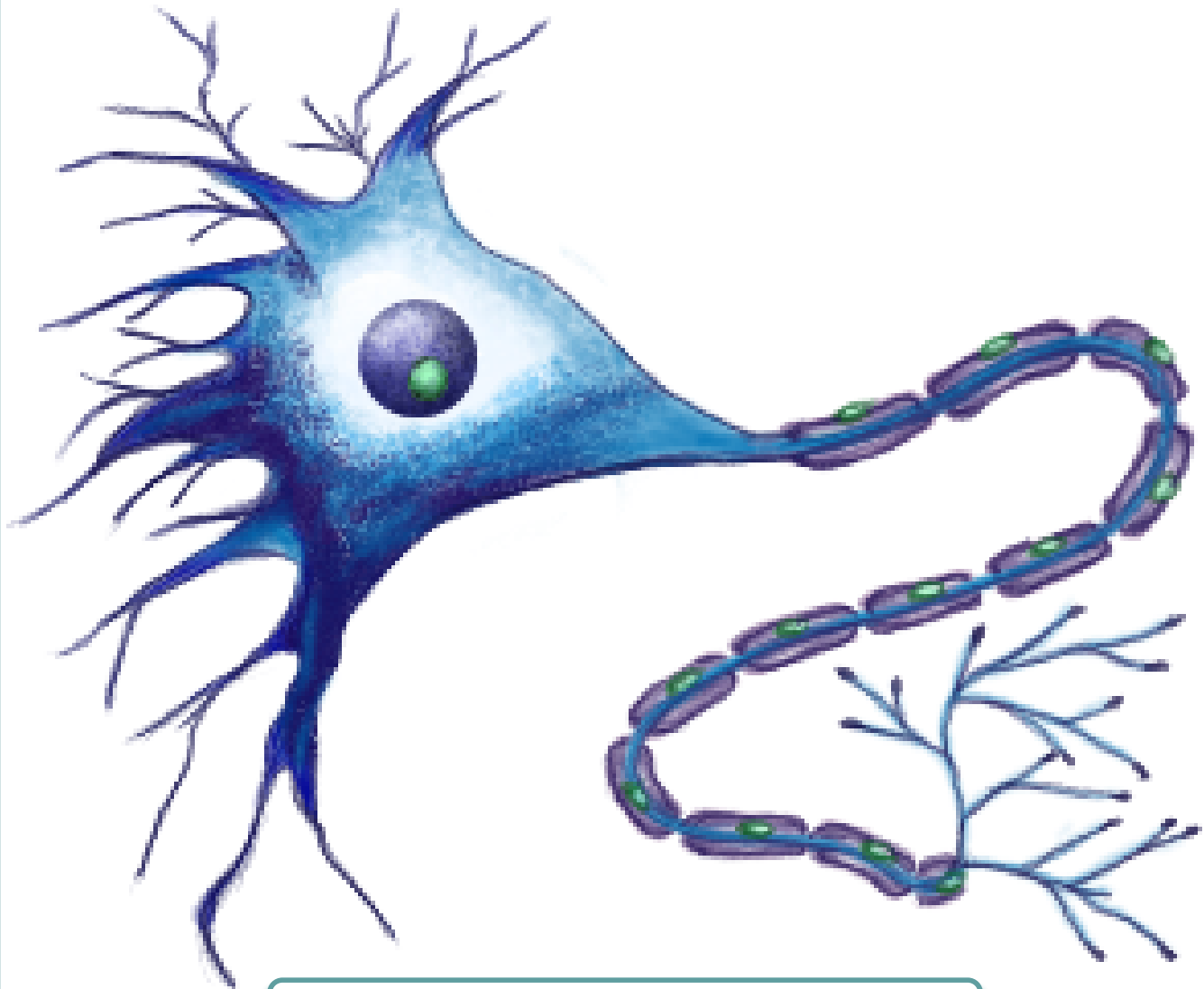


Redes Neuronales

Notas de clase

Karla Fernanda Jiménez Gutiérrez
Verónica Esther Arriola Ríos



FACULTAD DE CIENCIAS, UNAM

Índice general

Índice general	I
I Antecedentes	2
1 Neurona biológica	3
1.1 Neurociencias computacionales	3
1.2 Sistema Nervioso	4
1.3 Neurona biológica	5
1.3.1 La neurona	5
2 Modelo de Hodgkin-Huxley	6
2.1 Introducción	6
3 Aprendizaje de máquina	7
3.1 Introducción	7
3.2 Espacio de Hipotesis	9
3.3 Clasificación de los conjuntos de datos	10
3.3.1 Tipos de aprendizaje	11
II Redes dirigidas acíclicas	13
4 Perceptrón simple	14
4.1 Perceptrón	14
4.2 Compuertas lógicas con neuronas	17
4.3 Funciones de activación	19
4.4 Funciones de error	19
4.5 Medidas de rendimiento	22
5 Perceptrón multicapa	28
5.1 Intro	28
5.2 XOR	28
5.3 Propagación hacia adelante manual	30
5.4 Propagación hacia adelante vectorizada (con matrices)	31

5.5	Interpretación matemática del mapeo no lineal	34
5.6	Propagación hacia adelante para el perceptrón multicapa	35
6	Entrenamiento por retropropagación	37
6.1	Esquema general entrenamiento	37
6.2	Función de error: Entropía cruzada	40
6.3	Derivada de la función logística	41
6.4	Entrenamiento en la última capa	43
6.5	Parcial con respecto a los pesos en la última capa	46
6.6	Vectorización	49
7	Optimización del entrenamiento	50
7.1	Problemas en redes profundas	50
7.2	Gradiente desvaneciente (o que explota)	51
7.3	Estilos de entrenamiento	53
7.4	Normalización y normalización por lotes	55
7.5	Regularización	60
7.5.1	Descarte	64
8	Caso de análisis e interpretación	65
8.1	Red Hinton árbol familiar con numpy (entrenamiento)	65
8.1.1	Codificación de los datos de entrada	68
8.1.2	Alimentación hacia adelante	69
8.1.3	Entrenamiento	73
8.1.4	Momentum y decaimiento de los pesos	73
8.1.5	Ejecución del entrenamiento	73
8.2	Red Hinton árbol familiar con pytorch	74
9	Entrenamiento con genéticos	75
9.1	Algoritmos genéticos	75
9.2	Neuroevolución	75
9.2.1	Antecedentes: Aprendizaje por refuerzo en videojuegos	75
9.2.2	Arquitectura para estimar la función de recompensa	75
9.2.3	Entrenamiento	76
10	Mapeos autoorganizados	77
10.1	Introducción	77
10.2	Aprendizaje no supervisado	77
10.3	Mapeos autoorganizados	77
10.4	Kohonen	77
11	Redes Neuronales Convolucionales	78
11.1	Convolución	78
11.2	Redes Convolucionales	78
11.3	Softmax	78

11.4 MNIST	78
III Redes con ciclos	79
12 Redes Neuronales Recurrentes	80
12.1 Derivadas ordenadas	80
12.2 Retropropagación en el tiempo	80
12.3 Sistemas dinámicos y despliegue del grafo	80
12.4 Arquitectura recurrente universal	80
12.5 Función de error	80
12.6 Forzamiento del profesor	80
13 Atención	81
14 LSTM	82
15 GRU	83
16 Casos de análisis: etiquetado de palabras y conjugación de verbos	84
IV Redes no dirigidas	85
17 Redes de hopfield	86
17.1 Entrenamiento	86
18 Máquinas de Boltzman	87
18.1 Entrenamiento	87
18.1.1 Partículas y partículas de fantasía	87
18.1.2 Máquinas de Boltzman Restringidas	87
19 Redes adversarias	88
19.1 GANs	88
A Ecuaciones diferenciales	89

Etc

A lo largo del texto se utilizará la siguiente notación para diversos elementos:

Conjuntos	C
Vectores	x
Matrices	M
Unidades	cm

PARTE I

Antecedentes

1 | Neuronas biológicas

Neurociencias computacionales

El campo conocido como **neurociencias computacionales** es el que se dedica explícitamente al estudio/modelado de los sistemas biológicos con ayuda de varios campos de estudio. Se interesa notablemente en descripciones y modelos funcionales biológicamente realistas de neuronas y sistemas neuronales.

Los campos de estudio de los cuales nos ayudamos son:

- **Ciencias cognitivas** dedicadas a tratar directamente con los humanos, de estas tomamos los conceptos de aprendizaje.
- **Biofísica** por el estudio de las propiedades físicas en de los sistemas biológicos.
- **Neurociencias tradicionales** con modelos matemáticos.
- **Ciencias de la computación** modelar e implementar los modelos dados, por las áreas aquí listadas. Simulaciones computacionales.
- **Ingeniería eléctrica** diseño de hardware específico y eficiente para tomar los pulsos y medidas exactas.

Las neurociencias computacionales, estudian modelos del sistema nervioso y clasifican estos modelos en tres tipos, recordemos el método científico:

1. **Modelos descriptivos**, Describen. Por ejemplo describe el comportamiento de los ratones a ciertas sustancias. La **comparación**, que estaba pasando antes de la sustancia, con la sustancia y después.
2. **Modelos mecanistas**, Mecánicamente. El **cómo** paso el evento o la acción. Siguiendo con el ejemplo. El ratón se cayó y se levanto, ¿Cómo? Doblo sus articulaciones, tomo impulso con sus músculos, roto su cuerpo, hasta reincorporarse totalmente.

1. Neurona biológica

3. **Modelos interpretativos**, Se interpreta. ¿**Porque** hizo los movimientos mecánicos?. Siguiendo con el ejemplo. ¿Porque el ratón se reincorporo? Porque el ratón quiere regresar a su estado anterior antes de la perturbación por la sustancia. ¿Será verdad? Es lo que se interpreta, no la verdad absoluta. Se tiene que buscar intencionalidad, razonamiento de más alto nivel.

Los **objetivos del modelado** en neurociencias de nuestro interes en el curso son; Las corrientes, las proteínas, las oscilaciones de las redes completa, la arquitectura topografica y de columnas, el aprendizaje y la memoria. Por lo menos para el curso de Redes Neuronales en la Universidad Nacional Autonoma de México con la profesora Veronica Arriola Rios.

Para más información en Neurociencias **princNS5**, siempre está el interes propio.

Las redes neuronales artificiales (RNA), estan inspiradas en las redes neuronales biológicas, tales como las de los animales **incipiencias**, en un primer momento. Hasta llegar al ser humano con el sistema nervioso y nuestras neuronas. **ideaNeurona**.

Los modelos de redes neuronales que tomaremos, son simples. Distan mucho de los sistemas naturales y aun así dan solución. Lo que nos interesa es que resuelvan los problemas inmediatos y de corto plazo. Si el cerebro humano funciona, la imitación debe darnos algunos resultados, idea que ha funcionado para procesar información y dar solución con modelos diseñados.

Los problemas más notorios a resolver son:

- Problemas de visión por computadora.
- Procesamiento del lenguaje natural.

Sistema Nervioso

-Constestar de forma simple que es el sistema nervioso. -Porque es importante para el curso. -Que solución ha dado notar el sistema nervioso (humano) en el modelado de CC.

¿Qué es un nervio? Un nervio es una gran colección de axones empacados todos juntos en una especie de cable (fibra), pasan vasos sanguíneos por en medio de los nervios. Esto es de lo que está formando el sistema nervioso, se originan desde la médula espinal (31 pares de nervios raquídeos) o el encéfalo (12 pares de nervios craneales).

Neurona biológica

La neurona

¿Qué son las neuronas? La neurona es un tipo de célula perteneciente al sistema nervioso central, que se comunica tanto por señales eléctricas como por señales químicas. Son células con núcleo, sus cuerpos tienen dendritas con la habilidad de transferir electricidad y un axón. Éste es una terminal eferente, que permite transmitir un pulso eléctrico desde el cuerpo a través del axón hasta otras neuronas a distancias variadas.

2 | Modelo de Hodgkin-Huxley

Introducción

El modelo de HH.

3 | Aprendizaje de máquina

Introducción

En este capítulo se desarrolla el procesos que pasa una maquina para que *aprenda*, para esto notemos el concepto de aprender. En lo seres humanos se denota como el hecho de adquirir el conocimiento de algo mediante el estudio o la experiencia a partir de ejemplos específicos en nuestro medio, entonces aquellos problemas que inicialmente no pueden resolverse, puedan ser resueltos después de obtener más información acerca del problema. Desde pequeños empezamos por aprender por palabras (conceptos) que asociamos con algo específico, para después relacionar que varios objetos pertenecen a un tipo de conjunto y otros no. Como que un muñeco, una pelota y unos bloques de construcción de plastico, pertecen a un conjunto denotado como "juguetes", y que plato, taza, y tazón no pertecen a este conjunto sino al conjunto "vajilla". Entonces para organizar los conceptos que vamos aprediendo hacemos uso de una función booleana, donde la entrada es el concepto, la pregunta es, ¿Este objeto pertence a un cierto conjunto de objetos con características similares? y la salida es falso o verdadero. A este proceso, se le conoce como *aprendizaje de conceptos* y en este curso lo simplificamos como *función booleana de aproximación mediante ejemplos*.

Ahora lo que denotamos como el hecho que una maquina aprenda, lo vemos como cualquier programa que mejore su desempeño en alguna tarea mediante la experiencia. Con más formalidad se denota como:

Definición 3.1

Apredizaje maquina: Se dice que un programa de computadora aprende, si su desempeño en T , medido por P , mejora con la experiencia E . Tal que:

- T es un tipo de tarea.
- P es una medida de desempeño.

- E la experiencia con ejemplos (entrenamiento).

, ^a.

^aMachine Learning, Mitchell 1997, pag. 14.

Algunos ejemplos para definición anterior pueden ser los siguientes:

- T , como:
 - ★ Jugar un juego de mesa.
 - ★ Clasificar varios tipo de hojas.
 - ★ Reconocer una voz en particular.
- P , como:
 - ★ Porcentaje de juegos ganados en las partidas.
 - ★ Porcentaje de hojas correctamente clasificadas.
 - ★ Porcentaje de reconocimiento de timbre de la voz.
- E , como:
 - ★ Jugar juegos de práctica.
 - ★ Una secuencia de imagenes etiquetadas.
 - ★ Una secuencia de audios etiquetados.

A partir de ahora nos dedicaremos a definir correctamente tareas que nos interese que un programa aprenda, para entender la forma más abstracta del problema y así proponer los algoritmos que nos ayuden a resolverlo.

Consideremos también que los sistemas de redes neuronales artificiales son un tipo de algoritmo para la representación del proceso de aprendizaje. Un problema de aprendizaje bien definido requiere una tarea bien especificada, medidas de desempeño y datos para obtener experiencia.

El aprendizaje maquina se apoya de disciplinas, como la inteligencia artificial, probabilidad, estadística, complejidad computacional, psicología, neurobiología y filosofía.

Para proponer un algoritmos de aprendizaje automático necesitamos, elegir el tipo de experiencia de entrenamiento, definir la función objetivo a aprender y un algoritmo para aprender la función objeto a partir de ejemplos de entrenamiento.

Los algoritmos de aprendizaje maquina han sido utilizados apliamente por la industria bancaria, por gobiernos y por su puesto por el area de la salud. En la industria bancaria por

ejemplo, donde es necesario aprender las reglas generales para determinar la solvencia crediticia, a partir de las bases de datos. Por los gobiernos para el reconocimiento de rostros humanos a partir de imágenes. En el area de salud para a partir de bases de datos de pacientes descubrir automaticamente regularidades implicitas en los resultados de tratamientos.

Espacio de Hipotesis

El aprendizaje automático, es utilizar datos disponibles para, aprender una tarea mediante una función que mejor mapee entradas a ciertas salidas. A esto se le llama aproximación de función, en el que aproximamos una función de destino desconocida (que suponemos que existe) que puede asignar mejor las entradas a las salidas en todas las observaciones posibles del dominio del problema.

Una función de un modelo que se aproxima a la función objetivo y realiza asignaciones de entradas a salidas se denomina hipótesis.

Ahora estas funciones pueden tener formas muy generales en el aprendizaje de máquina pueden tener forma, por ejemplo, de estructuras de datos, como los árboles de decisión, donde cada nodo pregunta si o no, pertenece una clasificación. pueden ser también funciones matemáticas como el caso de las redes neuronales, entonces la forma que tomen estas hipótesis en general puede abarcar muchos métodos y estructuras.

Entonces el aprendizaje consiste en, explorar un espacio de posibles hipotesis para encontrar la hipotesis (una función) que mejor encaje, de acuerdo a lo se obtuvo en los ejemplos de entrenamiento, y predecir alguna característica de salida deseada. Usualmente se denotan como sigue:

- h (hipótesis): una sola hipótesis, por ejemplo una instancia o modelo candidato específico que asigna entradas a salidas, se puede evaluar y se usa para hacer predicciones.
- H (conjunto de hipótesis): Un espacio de posibles hipótesis para mapear entradas.

Una breve ejemplo para denotar un espacio de hipotesis sería el problema es saber los días que nos conviene ir al cine, donde nuestra tarea T es aprender a predecir el conjunto de días que nos conviene ir al cine, basado en los atributos de los días, donde cada hipotesis la representaremos a partir de un conjunto de atributos de las instancias (días), entonces cada hipotesis es un vector con tres atributos, *tiene2x1*, *esEstrenoDePelícula*, *actoresConocidos*. Para cada atributo de la hipotesis tendría uno de los siguientes valores; Si, No, ?. Donde ? indica que cualquier valor es valido para ese atributo.

Cuando alguna instancia x cumpla con todos los atributos de una h , entonces $h(x) = 1$ y x es un ejemplo positivo. Entonces para representar la hipotesis, que nos conviene ir

solo los dias con 2x1, y que hay peliculas donde los actores son conocidos, la escribimos como $h(<Si, ?, Si>) = 1$, la hipotesis que cualquier día nos conviene ir al cine la denotamos como $h(<?, ?, ?>) = 1$, nuestra función objetivo la denotamos como una función booleana $c : X \rightarrow \{0, 1\}$, el conjunto de los 365 días del año, entonces $c(x) = 1$ cuando en los datos nos dicen que con la instancia x conviene ir al cine, $c(x) = 0$ en caso que no. Por tanto para aprender la tarea T , necesitamos *una hipotesis h en H tal que $h(x) = c(x)$ para todas las x en X* . La tarea de aprendizaje del concepto c requiere aprender el conjunto de instancias que lo satisface, describiendo este conjunto mediante una conjunción de restricciones sobre los atributos de la instancia.

Estas hipotesis (funciones) pueden llegar a ser sumamente complejas y tener que mapear datos de entrada con muchas formas ej. imagenes, trayectorias, etc. En el caso de las redes neuronales, el espacio de hipótesis está determinado por la arquitectura de la red. Vamos a definir el espacio de hipótesis, cuando decidamos qué neuronas vamos a poner en nuestro sistema, como las conectamos entre sí y cómo van a transferirse información de una a la otra y cuántas neuronas van a ser. Lo que veremos a lo largo del curso son diferentes arquitecturas y el impacto que tiene hacer diferentes modificaciones así como las matemáticas que existen detrás de estas.

Clasificación de los conjuntos de datos

La experiencia E para aprender la vamos a obtener mediante un conjunto de datos, llamados datos de entrenamiento, estos se separan en tres bloques:

- **Entrenamiento:** Datos con los cuales se ajustan los parámetros de la hipótesis (del 50 % al 80 % de los datos). En este bloque se elige que función del espacio fue mejor para el aprendizaje.
- **Validación:** Datos utilizados para ajustar los parámetros (hiperparámetros) del algoritmo de entrenamiento, que puedan afectar qué hipótesis es seleccionada (del 25 % al 10 % de los datos y no deben ser usados durante el entrenamiento). Un ejemplo de un hiperparámetro para redes neuronales son el número de nodos ocultos en cada capa.
- **Prueba:** Datos utilizados para evaluar la posibilidad de que la hipótesis aprendida generalice ¹ a datos no vistos anteriormente. Esta porción que se mantiene aparte. Con estos se evalúa el modelo, se reporta la eficacia del modelo según los resultados en este conjunto (del 25 % al 10 % de los datos).

¹Se desea que nuestro modelo de aprendizaje, una vez entrenado con datos que ya hemos visto, se pueda usar con datos nuevos. Para ello debemos asegurarnos que el modelo no ha simplemente memorizado las muestras de entrenamiento, sino que ha aprendido propiedades del conjunto.

El conjunto de datos de entrenamiento se usa para aprender una hipótesis y el conjunto de datos de prueba para evaluarla.

Tipos de aprendizaje

Aprendizaje Supervisado , el modelo usa datos etiquetados a una respuesta específica (labeled data), durante el entrenamiento se intenta encontrar una función que aprenda a asignar los datos de entrada (input data) con los datos en el etiquetado. Para después predecir una relación, dado un dato totalmente nuevo para el modelo. Los modelos pueden ser:

- **Regresión:** Un modelo de regresión busca predecir valores de salida continuos. Por ejemplo, en predicciones meteorológicas, de expectativa de vida, de crecimiento de población.
- **Clasificación:** En un problema de clasificación se desea predecir una salida discreta. Por ejemplo, identificación de dígitos, diagnósticos.

Aprendizaje no supervisado , es usado cuando no se tienen datos “etiquetados” para el entrenamiento. Solo sabemos los datos de entrada. Por tanto, únicamente podemos describir la estructura de los datos, para intentar encontrar algún tipo de organización que simplifique un análisis. Por ello, no se tienen valores correctos o incorrectos (es utilizado para aprender de una manera autoorganizada).

Aprendizaje por refuerzo , inspirado en la psicología conductista; donde el modelo aprende por sí solo el comportamiento a seguir basándonos en *recompensas y penalizaciones*. Este tipo de aprendizaje se basa en mejorar la respuesta del modelo usando un proceso de retroalimentación (*feedback*). Su información de entrada es el feedback que obtiene del mundo exterior como respuesta a sus acciones. Aprende a base de ensayo-error.

Mientras que el aprendizaje supervisado y el no supervisado aprenden a partir de datos obtenidos en el pasado, el aprendizaje por refuerzo aprende desde cero, es decir, con un estado inicial y son su ambiente, va aprendiendo a futuro, mediante posibles penalizaciones o recompensas. El *aprendizaje por refuerzo* es usado en videojuegos porque cada vez que se realizan las acciones correctas se ganan puntos y entonces se entrena a la gente para que pueda conseguir la mayor cantidad de puntos. En este siempre hay: un agente, un ambiente definido por estados, acciones que el agente lleva a cabo (que le llevan de un estado a otro), y recompensas o penalizaciones que el agente obtiene.

En cada acción, el agente solo conoce el estado en el cual se encuentra y las acciones posibles que puede elegir a partir de ese estado. No sabe si llegando al siguiente estado, obtendrá mejores o peores recompensas, irá aprendiendo en cada estado qué acciones lo llevará a obtener una mayor recompensa a largo plazo, y que el valor de las acciones en ese estado puedan subir. *Se enfoca en que el agente aprenda una política óptima*

para alcanzar el objetivo. El agente siempre está en fases de *exploración* y *explotación*, en la fase de exploración el agente toma acciones de manera aleatoria, y en la de explotación va a tomar acciones basándose en cuán valiosa es realizar una acción a partir de un estado dado.

En plataforma de ventas en línea es donde podemos encontrar este tipo de modelo que están entrenados con este tipo de aprendizaje, donde al iniciar la sesión no conoce nada del usuario, solamente tiene un ambiente dado por los productos de la plataforma y su estado inicial es cero, para hacer individual la experiencia del usuario y que compre más. El algoritmo realiza la acción de mostrar ciertos productos (algún estado) si el usuario da clic a estos productos, el agente recibirá un punto de recompensa, por lo cual pasará a otro estado donde ofrecerá productos del mismo estilo donde pueda maximizar una venta, así se ira adaptando a cada usuario.

PARTE II

Redes dirigidas acíclicas

4 | Perceptrón simple

Perceptrón

El perceptrón fue la primera red neuronal artificial (o ANS, Artificial Neural Systems) descrita algorítmicamente. En las décadas de los 60's y 70's, los popularizó el psicólogo Franck Rosenblatt, en su libro llamado Principios de neurodinámica, donde presentó varios modelos de perceptrones, en el Laboratorio Aeronáutico de Cornell en Estados Unidos, originalmente estaba diseñado para ser una máquina, en vez de un algoritmo. Estaba diseñado específicamente para el reconocimiento de imágenes donde, cada peso era un cable físico por pixel de entrada, este era una matriz de 200×200 , conectados aleatoriamente a las "neuronas", las actualizaciones de los pesos se realizaron mediante motores eléctricos.

El perceptrón es en sí, es la representación de una sola neurona, este se ocupa para la clasificación de patrones en un conjunto de datos multivariados, con ese se obtienen fronteras lineales en el plano, mediante un algoritmo de aprendizaje que veremos más adelante.

Recordando, una neurona es una célula elemental que a partir de un vector de entrada procedente del exterior o de otras neuronas (estímulo), proporciona una única respuesta (si activo el potencial de acción o no), ver figura 4.1. Los elementos que actúan en una neurona los podemos listar como:

- **Entradas:** $x_j(t)$. Las variables de entrada y salida pueden ser binarias (digitales) o continuas (analógicas) dependiendo del modelo de aplicación.
- **Pesos sinápticos:** w_{ij} . Representan la intensidad de interacción entre cada neurona presináptica j y la neurona postsináptica i .
- **Regla de propagación:** $h_i(t) = \sigma(w_{ij}, x_j(t))$. Proporciona el valor del potencial postsináptico, de la neurona i en función de sus pesos y entradas.

★ $h_i(t) = \sum_{i=0}^n w_{ij}x_i$, Es una suma ponderada de las entradas con los pesos sinápticos. Así, si la entrada es positiva, dependiendo de los pesos podemos

saber si fue una sinapsis excitadora (pesos positivos) o inhibidora (pesos negativos).

- **Función de activación o de transferencia:** $a_i(t)$ Proporciona el estado de activación actual, de la neurona i en función de su estado anterior, $a_i(t-1)$ y de su potencial postsináptico actual.

★ $a_i(t) = f_i(a_i(t-1), h_i(t))$, es la que usualmente se usa.

★ $a_i(t) = f_i(h_i(t))$, en algunos modelos solo se considera que el estado actual no depende del tiempo anterior.

- **Función de salida:** $F_i(a_i(t))$ Da la salida actual, $y_i(t)$, de la neurona i en función de su estado de activación actual. El estado de activación de la neurona se considera como la propia salida.

★ $y_i(t) = F_i(a_i(t))$

★ $y_i(t) = F_i(f_i(a_i(t-1), \sigma(w_{ij}, x_j(t))))$

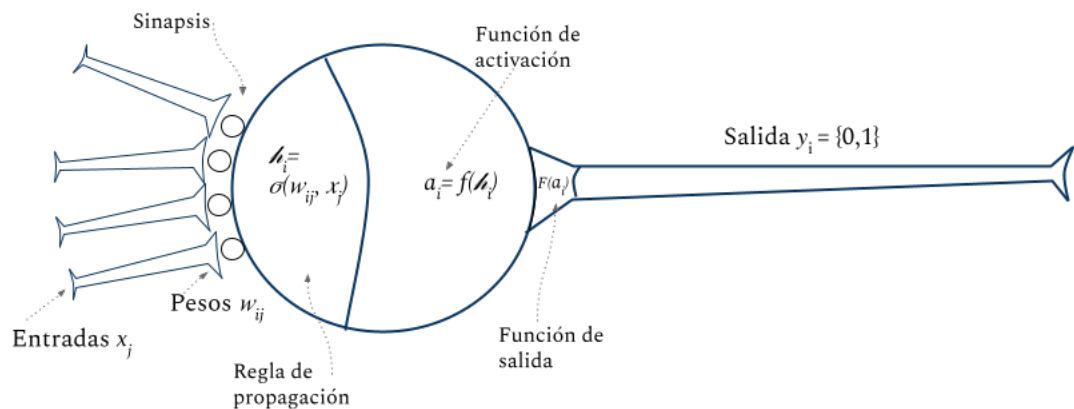


Figura 4.1 Neurona vista como un modelo artificial (perceptrón).

Un perceptrón toma un vector de entradas de números reales, calcula una combinación lineal de estas entradas, luego emite un 1 si el resultado es mayor que algún umbral y -1 de lo contrario. Es decir, dadas las entradas $x_1 \dots x_n$, la salida $o(x_1, \dots, x_n)$ calculada por el perceptrón es 1 si $w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0$ y -1 de lo contrario, donde cada w es una constante \mathbb{R} , un peso, que determina la contribución de la entrada x a la salida del perceptrón. La constante w_0 es un **umbral** (bias) que la suma de las entradas con los pesos debe superar para que el perceptrón emita un 1. En otras palabras es un peso que va a actuar junto con una entrada de valor 1, que vamos a poder ajustar para que nuestra función de activación, se mueva de derecha a izquierda en el plano para ayudarnos a ajustar nuestros resultados, provocando un gran impacto en el aprendizaje.

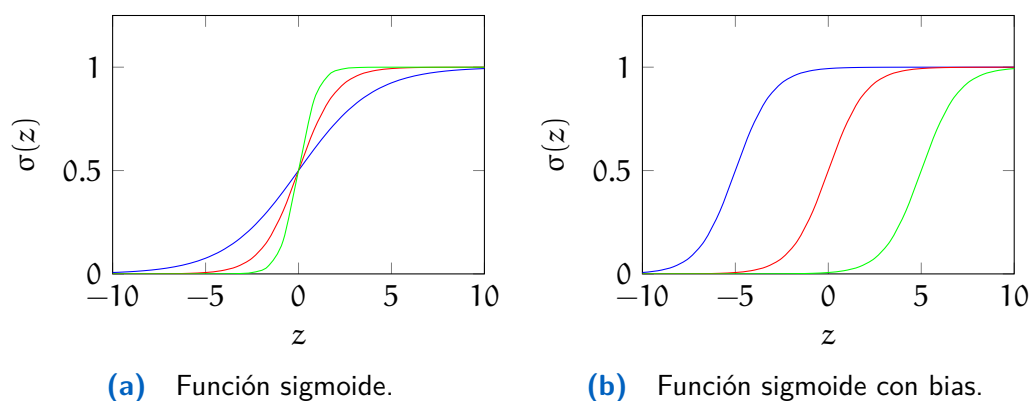


Figura 4.2 Comportamiento de la función de activación (sigmoide) de un perceptrón con una sola entrada, (b) el perceptrón sin el uso de bias, (b) con el uso del bias, donde apesar de estar representandos con la misma entrada , el uso del bias afecta en los resultados de salida. Así si quisieramos que este perceptrón nos diera $y = 0$ con una entrada $x = 2$ sin el uso, ni ajuste del bias sería imposible, pues en (a) apesar que la gráfica azul está la entrada está ajustada con el peso $w = 0.5$, la roja con el $w = 1$, y la verde con el $w = 2$ solo lo gramos alargarla un poco, haciendo que entradas que antes eran correctas ahora caigan 0 también. Entonces lo que necesitamos es más bien "mover"la gráfica, esto lo logramos con la gráfica (b) donde la entrada (única) está sumada con un bias (umbral) $x_0 = 1$, ajustado en azul con peso $w_0 = 5$, en rojo con $w_0 = 0$, en verde con $w_0 = -5$ y el peso $w_1 = 1$. Donde con $w_0 = -5$ logramos nuestro objetivo de tener una salida $y = 0$ con $x = 2$. El bias nos permite mover la función fuera del origen.

Esto se muestra en la siguientes graficas 4.2b. Más adelante hablaremos de su regla de entrenamiento (Training rule).

El hecho de que un perceptrón aprenda implica elegir valores para los pesos denotados también por θ . Ahora, el espacio H de las hipótesis candidatas consideradas en el aprendizaje del perceptrón, es el conjunto de todos los posibles vectores de pesos.

$$H = \{w | w \in \mathbb{R}^{n+1}\}$$

Si bien en el momento que se publicó los logros con el modelo del perceptrón las expectativas eran bastante altas, a medida de los años los científicos Marvin Minsky and Seymour Papert desestiman en gran medida los alcances que realmente se puede tener con el perceptrón, al mostrar ¹ que no puede predecir operaciones lógicas que no sean linealmente separables, tal es el caso de la función XOR, que no es separable linealmente, siendo imposible que pueda aprender esta función. Esto y el gran costo que representaba procesar todos los elementos que implicaba el entrenamiento, causa que por un buen

¹En 1969 publican Marvin Minsky y Seymour Papert que perceptrones de una sola capa (simples) solo son capaces de aprender a distinguir patrones linealmente separables, en el libro "Perceptrons".

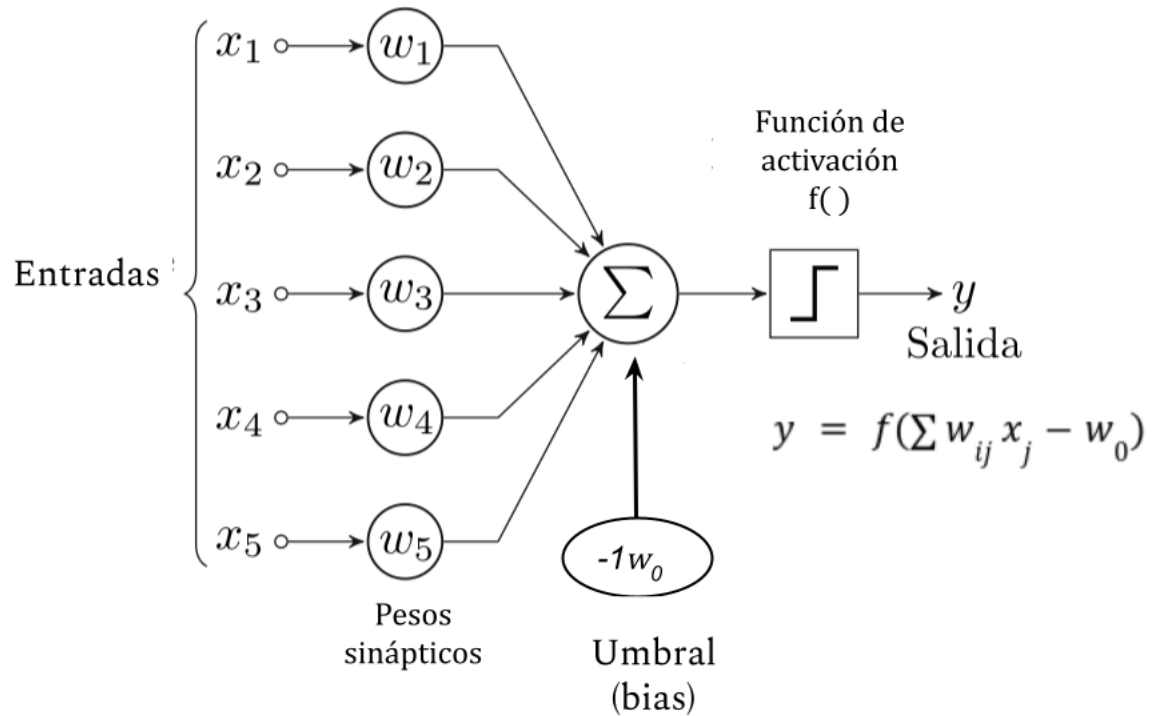


Figura 4.3 Modelo estándar de un perceptrón.

tiempo se desestime el uso del perceptrón. Es hasta después de unas décadas que vuelve a tener relevancia con la propuesta de un perceptrón multicapa usando retropropagación (Feedforward), siendo estos capaces de resolver la función XOR

Compuertas lógicas con neuronas

Aquí se muestra como se puede utilizar un perceptrón para simular compuertas lógicas tales como el or, not, and.

Para simular la compuerta *not*, como está es una función boolean de $B \rightarrow B$, talque $\text{not}(x) = -x$ entonces en un plano de dos dimensiones, la podemos representar con dos puntos, el $p_1 = (0,0)$ y $p_2 = (1,0)$ donde p_1 representa cuando $\text{not}(0) = 1$, p_1 representa cuando $\text{not}(0) = 1$. Teniendo el espacio de la función definido lo que nos toca es, separar el plano para clasificar las entradas, este claramente se puede separar con un linea vertical, o con lineas con pendiente 1 o -1 . Para esta función solo necesitamos de una entrada y un sesgo (bias), donde la entrada la combinaremos con un peso, este peso lo asignaremos a tanteo (por la sencillez de la operación). Así el peso $w_1 = -1$ y el peso asignado al bias sera $w_0 = 0.5$, ahora con esto datos podemos:

- Hacer la función de propagación donde $h(x) = (x * -1) + (0.5) * 1 = 0.5 - x$

- Hacer la función de activación escalón $\alpha(x) = \text{sgn}(h) = \text{sgn}(0.5 - x)$
- Dar la salida donde $s(1) = \text{sgn}(0.5 - 1) = 0$ y $s(0) = \text{sgn}(0.5 - 0) = 1$, en este caso la salida es la identidad de la activación.

x	h	s
0	0.5	1
1	-1.5	0

Algo similar va a pasar con la compuerta **and** y **or** donde al necesitar de dos entradas para la compuerta, asignaremos dos entradas para el perceptrón igualmente y las representaremos en el plano con cuatro puntos, donde cada punto representa una instancia y se le asigna valor positivo o negativo en el plano, así pues para el and tenemos los puntos $p_1 = (0, 0)$, $p_2 = (0, 1)$, $p_3 = (1, 0)$, negativos y $p_4 = (1, 1)$ el único positivo. Así nos damos cuenta que necesitamos una recta con pendiente negativa y fuera del origen, que nos separe estas clases de puntos. Por tanto para el bias le asignamos un peso de $w_0 = -1.5$, $w_1 = 1$ y $w_2 = 1$, con estos datos podemos:

- Hacer la función de propagación donde $h((x_1, x_2)) = (x_1 * 1) + (x_2 * 1) + (-1.5) * 1 = x_1 + x_2 - 1.5$
- Hacer la función de activación escalón $\alpha((x_1, x_2)) = \text{sgn}(h) = \text{sgn}(x_1 + x_2 - 1.5)$
- Dar la salida donde $s = \alpha(x)$, es la identidad de la activación.

x_1	x_2	h	s
0	0	-1.5	0
0	1	-0.5	0
1	0	-0.5	0
1	1	0.5	1

Para la compuerta **or** es algo muy similar pues podemos igualmente representar la función con cuatro puntos en el espacio cada uno representando una instancia, solo que ahora tres de estos puntos serán positivos y solo uno negativo, los puntos positivos serían $p_2 = (0, 1)$, $p_3 = (1, 0)$ y $p_4 = (1, 1)$, mientras que $p_1 = (0, 0)$ negativo, el plano lo podemos dividir con una línea recta con pendiente negativa, así le asignamos $w_0 = -0.5$, $w_1 = 1$ y $w_2 = 1$, con esto hacemos los paso que ya sabemos:

- Hacer la función de propagación donde $h((x_1, x_2)) = (x_1 * 1) + (x_2 * 1) + (-0.5) * 1 = x_1 + x_2 + 0.5$
- Hacer la función de activación escalón $\alpha((x_1, x_2)) = \text{sgn}(h) = \text{sgn}(x_1 + x_2 + 0.5)$

- Dar la salida donde $s = \alpha(h)$, es la identidad de la activación.

x_1	x_2	h	s
0	0	-0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	-1.5	1

Tomando el hecho que en la naturaleza las neuronas van pasando información en una estructura que forma niveles de abstracción, esto lo modelamos como capas de neuronas conectadas entre sí, cada capa haciendo su trabajo de abstracción. El perceptrón simple es un modelo neuronal unidireccional, una capa de entrada y otra de salida, que por si solo no puede separar todas la funciones lógicas pues tenemos el XOR, para resolver esto usaron perceptrones multicapa que se explicará más adelante en el curso.

Funciones de activación

Recordando que la forma de las funciones de activación es $y = f(x)$, donde x representa el potencial postsináptico e y el estado de activación de la neurona, es decir si va a lanzar un disparo o no. Las funciones de activación más empleadas son:

Funciones de error

Entonces tomando como base el perceptrón, una vez obtenidas las salidas con una primera iteración nos daremos cuenta de que tan lejos o que tan cerca estuvimos de la respuesta correcta, con esto darnos la oportunidad de que pesos ajustar respecto a sus entradas, en la segunda iteración. Ahora para facilitarnos esto y tomando en cuenta que el entrenamiento consiste en varias iteraciones hasta llegar a aprender la tarea T , hacemos uso de una función de error que nos ayude a minimizar la diferencia de error en las salidas.

Primero veamos el entrenamiento para una sola neurona, para esto haremos uso de la regla de aprendizaje del perceptrón (learning rule perceptron), donde para cada entrada, en la capa de salida se le calcula la desviación a la función objetivo. El cual utilizamos para ajustar los pesos del perceptrón (ver fig 4.5).

Usualmente al principio del entrenamiento se asignan pesos aleatorios, a medida que avance el entrenamiento, se van modificando con cada iteración, así $w_i \leftarrow w_i + \Delta w_i$. Esto con base a [la regla de aprendizaje](#) donde:

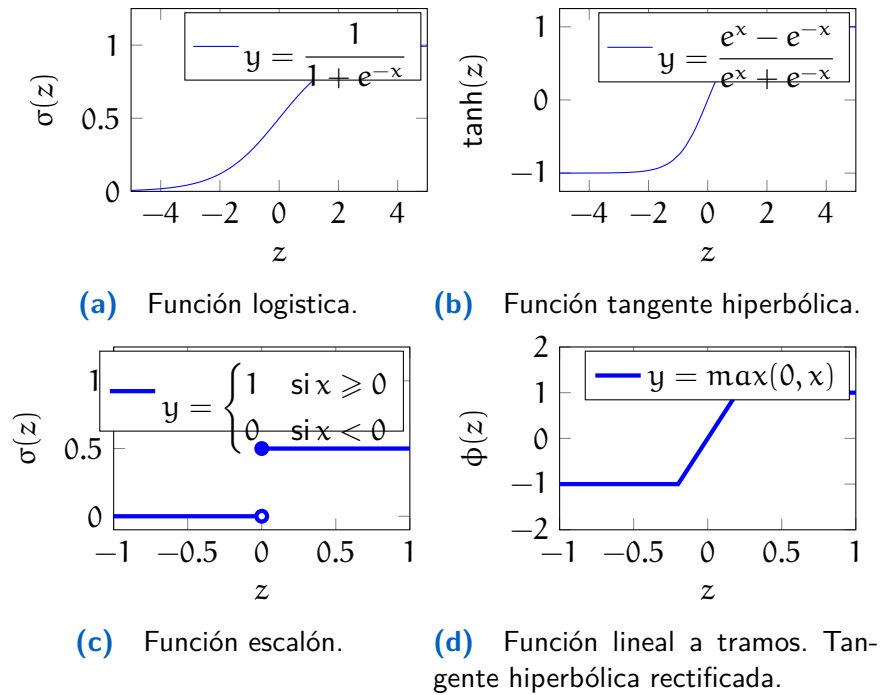


Figura 4.4 Las funciones de activación más usadas son la función sigmoide $\sigma(z)$ y la tangente hiperbólica $\tanh(z)$.

$$\Delta w_i = \alpha(y - y_{out})x_i \quad (4.1)$$

Con y es la salida deseada, y_{out} la salida generada, α la tasa de aprendizaje (learning rate) y x_i la entrada i . Lo que hace la tasa de aprendizaje es moderar el grado en que los pesos son cambiados con cada iteración, se le asigna un valor muy pequeño (0.1 o 0.2) y conforme se logran ajustar los pesos se minimiza aún más.

Para entrenar un perceptrón, utilizamos cualquier método de optimización de funciones para encontrar los parámetros w que minimizan el error con alguna de las siguientes funciones de error:

Diferencias al cuadrado, también conocida como regla aprendizaje delta, es la suma de cuadrados de errores que se tuvieron con cada ejemplar del conjunto de entrenamiento. Los podemos describir como:

$$\frac{1}{2m} \sum_{m=0}^{M-1} (y_m - a_m)^2 \quad (4.2)$$

con y_m y a_m , la salida obtenida dado un ejemplar m y la salida correcta del ejemplar m respectivamente.

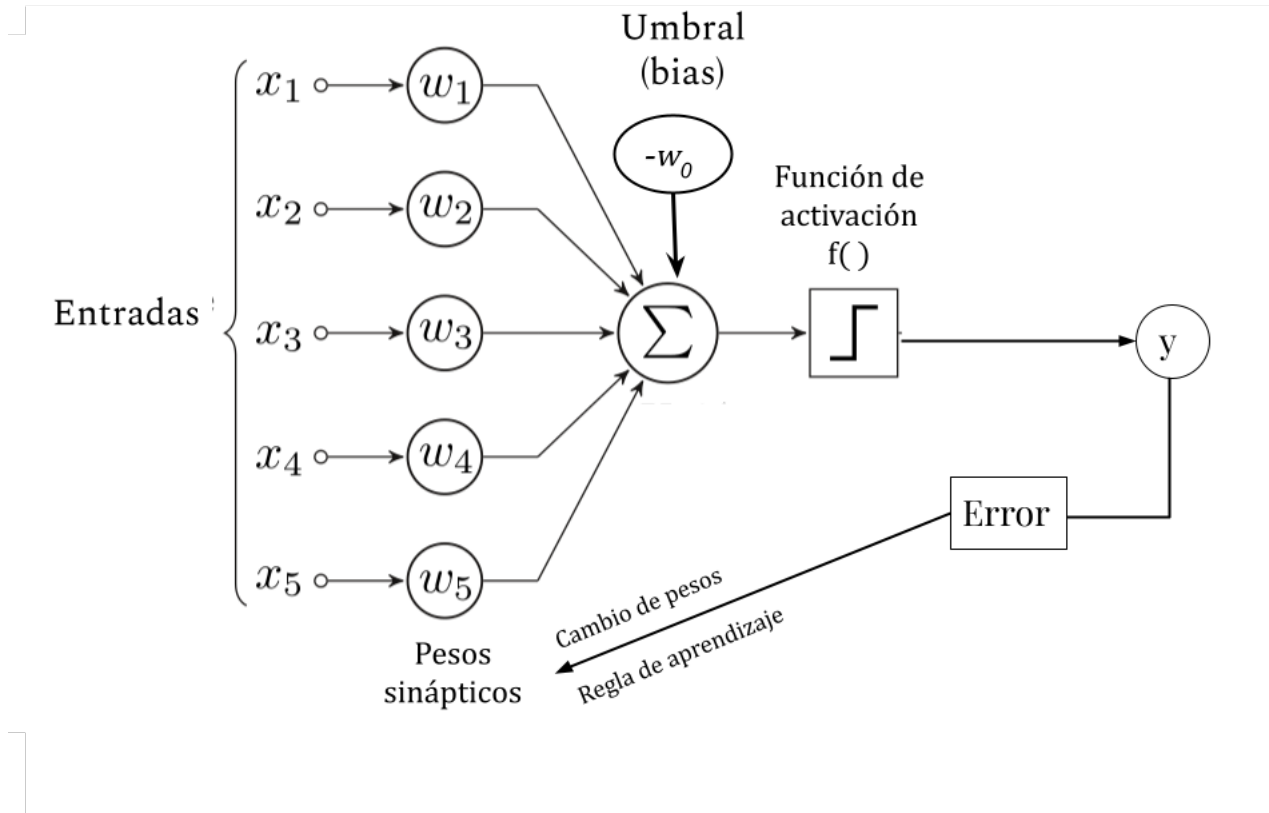


Figura 4.5 Modelo estándar de un perceptrón.

Entropía cruzada. Se usa para problemas de clasificación ya que su comportamiento es más suave (soft) y permite hacer clasificaciones más certeras. Se define como:

$$L(\Theta) = -\frac{1}{m} \sum_{m=0}^{M-1} (y^{(m)} \log(a_m) + (1 - y_m) \log(1 - a_m)) \quad (4.3)$$

Entonces juntando los conceptos que ya sabemos podemos describir el algoritmo de entrenamiento para el perceptrón de la siguiente forma:

1. Iteramos sobre todos los ejemplares.
2. Para cada ejemplar se calcula la función de propagación, es decir la suma ponderada.
3. Se calcula la función de activación con esta suma.

4. Se calcula la función de salida.
5. Actualización de pesos de acuerdo a la regla de aprendizaje.
6. Repetir de 1-6 hasta que los pesos nos satisfagan, un número de iteraciones establecidas.

Medidas de rendimiento

Las medidas de rendimiento de una red nos sirven para ver de manera concreta como se comportó nuestra red, durante el entrenamiento. Que fue lo que pudo aprender (clasificar). Si tuvo un sobreajuste, es decir, memorizo y por tanto, será incapaz de predecir la clase a la que pertenece realmente el ejemplar. Para esto se usan las siguientes herramientas:

Matriz de confusión : (Confusion matrix) Es una matriz, donde las celdas representan las predicciones que hizo nuestro modelo de clasificación de clases, respecto a las salidas esperadas. Así siendo las columnas las salidas y 's del modelo entrenado y las filas las salidas esperadas y_{true} . Nos facilita a ver cuando un clasificador está confundiendo clases, contabilizando a que clase etiqueto a los diferentes ejemplares. Ahora veamos, que puede estar representando cada celda en la matriz de confusión, estos pueden ser:

1. **VP** Verdaderos positivos (*TP, True Positive*): La clasificación de los ejemplares predichos, conciden con las etiquetas esperadas de los ejemplares.
2. **VN** Verdaderos negativos (*TN, True Negative*): El ejemplar que no es parte de clase, no son asignados a esa clase, son predichos correctamente.
3. **FP** Falso positivo (*FP, False Positivo*): El ejemplar que **no** es parte de una clase i fue clasificado como tal.
4. **FN** False negativo (*FN, False Negative*): El ejemplar que es parte de una clase i no fue clasificado como tal.

Ejemplo 4.1. *Notemos un ejemplo sencillo, supongamos que tenemos una tarea binaria, donde queremos indicar que una persona está embarazada (de acuerdo a unos estudios). Ahora tenemos que:*

- *VP, sería predecir que una mujer está embarazada y que en efecto esté embarazada. (Correcto)*
- *VN, sería con un hombre que no está embarazado y pues en efecto no está embarazado. (Correcto)*
- *FP, sería predecir que un hombre está embarazado y **no** este embarazado. (Error, tipo 1)*

- **FN**, sería predecir que una mujer **no** está embarazada y esta embarazada. (**Error, tipo 2**)

Entonces dados los resultados binarios que nos entregó el modelo, los médicos nos dan las respuestas correctas a 10 estudios, representadas en la siguiente tabla.

Ejemplar	1	2	3	4	5	6	7	8	9	10
Sujeto	M	F	M	F	M	M	F	F	F	M
Etiqueta	No	Si	No	Si	No	No	No	Si	No	No
Clase	0	1	0	1	0	0	0	1	0	0
Predicho	0	0	0	1	0	1	0	1	0	0
Valores	VN	FN	VN	VP	VN	FP	VN	VP	VN	VN

Con estos datos, podemos construir nuestra matriz de confusión MC contabilizando las salidas obtenidas respecto a las deseadas. Quedando así de la siguiente manera:

		Salidas y	
		Si	No
Etiquetas	Si	VP = 2	FN = 1
	No	FP = 1	VN = 6

Tabla 4.1 Matriz de confusión binaria.

Ahora para una modelo que sea multiclase, en el que tengamos que clasificar varias clases de ejemplares. Tendremos una matriz M de $n \times n$ donde n es el número de clases, así los valores **VP**, **VN**, **FP**, **FN**, son calculados para cada clase e identificados en la matriz M de la siguiente forma, también puedes ver la figura 4.6:

- **VP**, para la clase i será la celda $M[i][j]$ con la $i = j$.
- **VN**, para la clase i será la suma de los valores de toda la matriz menos los valores de la fila i ni los valores de la columna i .
- **FN**, para la clase i será la suma de los valores en la fila i , excepto la celda $M[i][j]$, con $i = j$ que es el **VP**.
- **FP**, para la clase i será la suma de los valores en la columna i , excepto la celda $M[i][i]$ que es el **VP**.

Valores para la clase i		Salidas y del clasificador			
		clase 1	clase 2	clase i	clase n
Etiquetas ytrue	clase 1	VN	VN	FP	VN
	clase 2	VN	VN	FP	VN
	clase i	FN	FN	VP	FN
	clase n	VN	VN	FP	VN

Figura 4.6 Matriz de confusión para clasificador multiclases.

Ejemplo 4.2. Tenemos un clasificador encargado de identificar las cinco vocales del español, escritas a mano. Entonces tenemos un total de 5 clases representadas por a, e, i, o, u, cada letra representada por un número del 0 al 4. Así la clase 0 = a, la clase 1 = e y así respectivamente. Este fue entrenado con 15 ejemplares etiquetados y se obtuvieron los siguientes resultados:

Ejemplar e	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Etiqueta	a	e	i	o	u	a	a	e	i	i	o	u	o	u	a
Clase y_{true}	0	1	2	3	4	0	0	1	2	2	3	4	3	4	0
Predicho y	0	1	3	3	2	1	1	1	2	2	4	4	3	2	0

Ahora para hacer la matriz de confusión MC notamos que tenemos una matriz de 5x5, donde $MC[i] = \text{Predicho } y$ y $MC[j] = \text{Etiquetas Reales}$. Entonces necesitamos contabilizar las predicciones y asignarlas a sus respectivas celdas, donde con la tabla anterior notamos que en el ejemplar e3 con $y_{true}(e3) = 2$, se predijo que $y(e3) = 3$, entonces $MC[2][3] += 1$, pues la etiqueta nos posiciona en la fila 2 y lo predicho en la columna 3. Ahora en el ejemplar e4 con $y_{true}(e4) = 3$ se predijo que $y(e4) = 3$, entonces $MC[2][2] += 1$. Así con cada ejemplar vamos a ir sumando su clasificación. Para construirla tendríamos un pseudocódigo siguiente:

```
def getMC (Ejemplares, y):
    """
    :param y: salidas obtenidas (int)
    """
    MC = [len(y)][len(Ejemplares)] # y == Ejemplares
    full_zeros(MC)
    for e in range (0, len(Ejemplares)):
        predicho = y[e]
        y_true = Ejemplares[e].etiqueta
```

```

        MC[predicho][y_true] += 1
    return MC

```

Dados los datos anteriores tenemos, la siguiente matriz de confusión:

		Salidas y				
		1	2	3	4	5
Etiquetas	1	2	2	0	0	0
	2	0	2	0	0	0
	3	0	0	2	1	0
	4	0	0	0	2	1
	5	0	0	2	0	1

Tabla 4.2 MC = Matriz de confusión multiclase.

Para calcular los valores *VP*, *VN*, *FP*, *FN*, por clase se propone el siguiente pseudo-código:

```

MC = getMC(y_salidas, y_true)

VP = VN = FP = FN = [0,0,0,0,0]

for i in range(len(MC)):
    for j in range(len(MC[0])):
        FN[i] = FN[i] + MC[i][j] if (i != j) else FN[i]
        FP[i] = FP[i] + MC[j][i] if (i != j) else FP[i]
        VP[i] = MC[i][j] if (i == j) else VP[i]
    VN[i] = sum(MC) - FN[i] - FP[i] - VP[i]

```

Si quisiéramos saber los valores *VP*, *VN*, *FP*, *FN*, para todo el desempeño total, simplemente sumamos lo obtenido en cada clase así:

```

VP_total = sum(VP)
VN_total = sum(VN)
FP_total = sum(FP)
FN_total = sum(FN)

```

Ahora los valores de cada celda nos representan lo siguiente:

- VP_{total} , toda la diagonal de la matriz. La salida del modelo coincide con lo etiquetado con el ejemplar.
- VN_{total} , todas las veces que no era la clase i y dijo que no era de la clase i .
- FN_{total} , todas las veces que era clase i y dijo que era clase x . (deseado $FN = 0$)
- FP_{total} , todas las veces que predijo que era clase i y era clase x . (deseado $FP = 0$)

Así que, si los valores que no están en la diagonal de la matriz son cero o todas nuestras clasificaciones están en la diagonal, podemos decir que nuestro modelo aprendió a clasificar correctamente todas las clases.

Precisión y recuperación : La precisión (*precision*) nos dice la proporción de ejemplares que se logró clasificar correctamente. Mientras que recuperación (*recall*) nos dice cuantas asignaciones de lo que nos interesa pudo clasificar correctamente en otras palabras donde del total de las respuestas correctas que se pueden tener, cuantas respuestas positivas acertadas se tuvo.

$$P = \frac{VP}{VP + FP} = \frac{\text{VerdaderosPositivos}}{\text{ValoresEsperados}} \quad (4.4)$$

$$R = \frac{VP}{VP + FN} = \frac{\text{VerdaderosPositivos}}{\text{ValoresPredichos}} \quad (4.5)$$

Valores para la clase i		Salidas y del clasificador			
		clase 1	clase 2	clase i	clase n
Etiquetas y true	clase 1	VN	VN	FP	VN
	clase 2	VN	VN	FP	VN
	clase i	FN	FN	VP	FN
	clase n	VN	VN	FP	VN

Recall = $VP / (VP + FN)$
Cuantos valores esperados fueron asignados realmente.

Precisión = $VP / (VP + FP)$
Cuanto de lo que clasifico fue correcto

- Cuando el modelo detecta los ejemplares, pero los incluye en otras clases también: P es bajo y R es alto.
- Cuando el modelo no detectó bien los ejemplares, pero tampoco los incluyo en otras clases: P es alto y R es bajo.

- Cuando el modelo detecta los ejemplares y no los incluye en otras clases: P y R es alto.
- Cuando el modelo no detecta los ejemplares: P y R es bajo.

En ocasiones le daremos más importancia al recall y en otras a la precisión. Retomando el ejemplo previo 4.1, no nos importa tanto los casos de error tipo 1, donde el modelo se equivoca con los ejemplares negativos, nos importa los errores de tipo 2, los Falsos Negativos, en este caso le tomaremos especial atención al Recall y que este sea alto. Pero si bien nuestra tarea es detectar cuando un correo es spam, no impacta tanto que aunque en ocasiones un correo sea spam, no lo clasifique como tal (error tipo 2 FN), nos es crucial que un correo que no sea spam nos lo clasifique como tal (error tipo 1 FP). Así deseando que los falsos positivos se acerquen a cero, dándonos como resultado una precisión alta aunque el recall sea bajo.

Exactitud y medida f : La exactitud (*accuracy*) es una medida de cuántas predicciones correctas en total hizo el modelo para el conjunto de datos completo (no se recomienda usar si tienes clases desbalanceadas, es decir, muchos elementos de una clase y poco de otra pues nos puede fallar totalmente con las clases pequeñas y aun así su valor sería alto). La medida f (*f score*) se utiliza para combinar las medidas de precisión y recall en un solo valor. Es práctico porque hace más fácil el poder comparar el rendimiento combinado de la precisión y la recall. Se dan las ecuaciones a continuación:

$$\text{Accuracy} = A = \frac{VP + VN}{VP + VN + FP + FN} = \frac{VP + VN}{\text{TodosLosValoresClasificados}} \quad (4.6)$$

$$F = \frac{2}{\frac{1}{P} + \frac{1}{R}} = 2 \frac{P * R}{P + R} \quad (4.7)$$

La medida f, también la podemos escribir (con un poco de aritmética) como:

$$F = \frac{2VP}{2VP + FP + FN} \quad (4.8)$$

5 | Perceptrón multicapa

Intro

Perdí el archivo :CCCC

XOR

Anteriormente, habíamos logrado separar clases de ejemplares siempre y cuando estos se pudieran modelar en un plano linealmente separable. Hasta ahora había sido un reto importante (no logrado) hacer clasificaciones de ejemplares distribuidos en un plano no separable linealmente, como lo es *la función XOR* donde, tenemos que las respuestas positivas se encuentran en una diagonal opuesta a las respuestas negativas y no hay manera de separar a los blancos de los negros con una sola frontera lineal, lo que hace imposible separar con un solo perceptrón. Entonces notamos que necesitamos de dos líneas que nos permitan separar el plano. Así llegamos a la idea que necesitamos más de una capa, que nos permita hacer la siguiente separación del plano.

Entrada x_1	Entrada x_2	Salida y
0	0	0
0	1	1
1	0	1
1	1	0

Figura 5.1 Tabla de verdad para la función XOR.

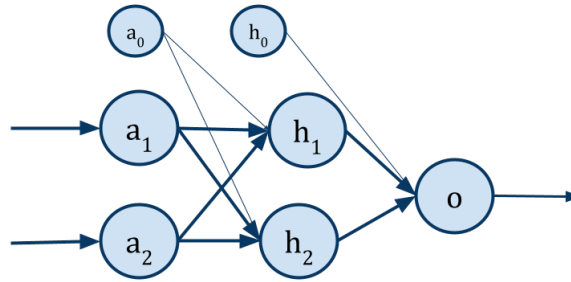


Figura 5.2 Perceptrón para la función XOR, con una capa oculta.

En la tabla de verdad notamos que para la función XOR, cuando la suma de las entradas es:

- par (tomando en cuenta al cero como par) la salida es 0.
- impar la salida es 1.

Para poder aprender la función únicamente tenemos que añadir una capa intermedia, la llamada capa oculta. Esta capa será la responsable de tomar las características de los vectores de entrada.

Ahora para la función tenemos dos entradas, una capa oculta (con dos neuronas) y una salida. Las neuronas en la capa oculta nos permitirán dividir el plano. Así, una solución sería que las neuronas ocultas denotadas por h (*hidden* de oculto en inglés) compartan pesos, la primera neurona h_1 haría la función OR, haciendo la distinción entre las entradas 00, la segunda neurona h_2 haría la función NAND haciendo posible distinguir el vector de entrada 11, una vez obtenido estos resultados de la capa oculta la neurona de salida o de output se encargara de distinguir la intersección entre estas, ejecutando la función AND. En resumen, la función **XOR** la podemos describir como:

$$\begin{aligned} \text{XOR} &= (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2) \\ \text{XOR} &= \text{AND}(\text{OR}(x_1, x_2), \text{NAND}(x_1, x_2)) \end{aligned} \quad (5.1)$$

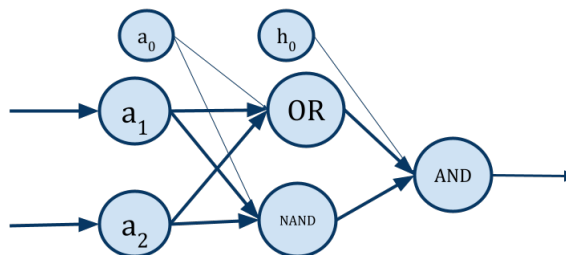


Figura 5.3 Solución para perceptrón de la función XOR.

Propagación hacia adelante manual

En esta parte vamos a ver como podemos evaluar la red para la solución que se dio en la sección anterior. (Recordemos que también se pueden dar otras soluciones para el XOR).

Entonces el procedimiento matemático general para poder evaluar una red cuando tenemos más de un perceptrón. Veamos primero que pasa con el **xor**, y el **nand**, por el momento no tomaremos en cuenta las neuronas de entrada a puesto que solo son usadas para almacenar las entradas. La capa oculta está formada realmente por dos perceptrones, que dan su salida a una neurona más que es la capa de salida de nuestra red. Para calcular sus salidas debemos aplicar la suma ponderada de las entradas a una función activación, así podemos ver que la salida de una neurona oculta es la siguiente:

$$z_j = g\left(\sum_i w_{ij} a_i\right) \quad (5.2)$$

A estos perceptrones a su vez se le están aplicando la función de activación sigmoide:

$$h_j = \frac{1}{1 + e^{-z}} \quad (5.3)$$

Así la neurona de salida recibe a los perceptrones ya evaluados y listos para aplicar pesos a estos y una función de activación, que en este caso son dados de la siguiente forma:

$$z_o = g\left(\sum_j w_{jo} h_j\right) \quad (5.4)$$

$$o = \frac{1}{1 + e^{z_o}} \quad (5.5)$$

Así tomando de ejemplo la función XOR, los valores de la capa oculta son evaluados de la siguiente forma, donde x_1 y x_2 son evaluados por 0 o 1 según sea necesario:

$$\begin{aligned} h_0 &= 1 \\ h_1 &= g(1w_{01} + x_1w_{11} + x_2w_{21}) \\ h_2 &= g(1w_{02} + x_1w_{12} + x_2w_{22}) \end{aligned} \quad (5.6)$$

Entonces hasta aquí ya tenemos los valores de la capa oculta, una vez que ya tenemos este conjunto de valores podemos empezar a trabajar con el tercer perceptrón, sus valores de entrada van a estar dados por los valores de activación de h_1 y h_2 y por un sesgo,

pues recordemos que es la función Nand, por tanto necesitamos movernos ligeramente del origen. Lo que vamos a tener aquí la fórmula se ve similar, lo único es que ahora los valores de entrada fueron los valores que obtuvimos en el cálculo de la capa anterior:

$$o = g(h_0w_{0o} + h_1w_{1o} + h_2w_{2o}) \quad (5.7)$$

Como estamos trabajando capa por capa, primero entran los valores con los que van a trabajar todos los perceptrones, después calculamos todos los de la capa oculta que son independientes entre sí, aunque tengan en común las mismas entradas y finalmente hacemos el cálculo de la siguiente capa, que en ese caso es la de salida, pero bien podría ser otra oculta.

Por la forma de recorrer la red (de izquierda a derecha) este algoritmo obtiene su nombre de *propagación hacia adelante*, pues lo calculado en la neurona de la capa anterior se propaga directamente a la siguiente capa, en inglés se conoce como *feedforward*. Cuando tenemos la evaluación de las capas ocultas, podemos obtener finalmente la salida final, con una evaluación final.

Si bien esta forma de evaluación nos lleva al resultado correcto, este método se puede simplificar sobretodo en el caso que estemos manejando más entradas y más perceptrones en las capas ocultas. Así nos daremos cuenta que es posible usar matrices, esta forma de evaluación la veremos en la siguiente sección.

Propagación hacia adelante vectorizada (con matrices)

La sección anterior si bien nos da la idea somera de como van a ser las operaciones para el aprendizaje ahora veamos lo de forma matricial. Esto nos ayudará a en un momento dado escribirlo en el lenguaje de nuestra convencia.

Retomando la red neuronal de la sección anterior ahora con pesos.

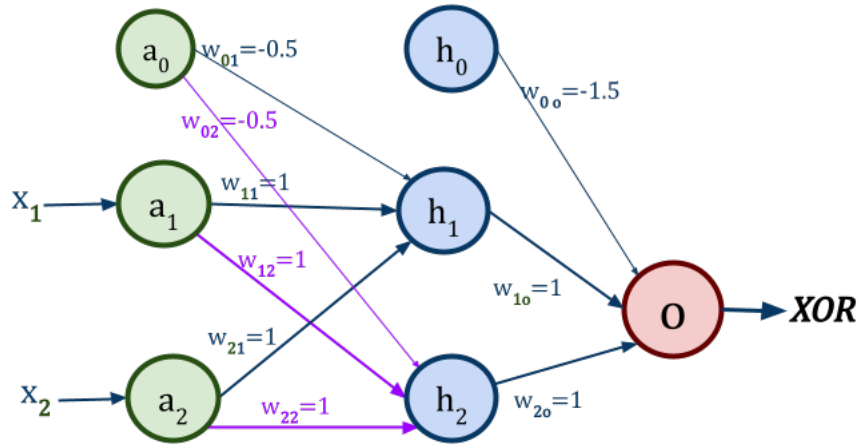


Figura 5.4 Función XOR, con capa oculta

Ahora veamos a las entradas como un vector de la siguiente forma:

$$A = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Ahora para poder hacer los siguientes cálculos acomodamos los pesos correspondientes a cada perceptrón. Para el primer perceptrón tomamos los pesos que están conectados desde la neurona de origen a_0 , a_1 y a_2 hacia la neurona oculta h_1 (los índices de los pesos indican, el origen y el destino, en ese orden), los colocamos en un renglón de la matriz de pesos, así representamos nuestro primer perceptrón y hacemos lo mismo para h_2 , así obtenemos la siguiente matriz de pesos:

$$W = \begin{bmatrix} w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \end{bmatrix} = \begin{bmatrix} -0.5 & 1 & 1 \\ 1.5 & -1 & -1 \end{bmatrix}$$

Ya que tenemos la representación de las entradas y pesos en vectores podemos hacer la representación de la activación de estas neuronas de la capa oculta, que es de la forma $H = g(W * A)$, que de forma matricial lo podemos escribir de la siguiente forma:

$$H = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = g \left(\begin{bmatrix} -0.5 & 1 & 1 \\ 1.5 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) = g \left(\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

La matriz resultante representa los valores de activación resultantes en cada perceptrón, el primer valor de la matriz representa el resultado de la activación de h_1 y el segundo a h_2 . Así ahora tenemos al vector H que será el vector de entrada para la neurona de

salida o , procedemos de la misma forma tomando en cuenta al sesgo h_0 y a los pesos en forma matricial, nos queda de la siguiente forma $o = g(W_{ho} * H)$:

$$o = g \left(\begin{bmatrix} -1.5 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = g([0.5]) = [1]$$

Hasta este momento ya logramos resolver la función XOR para una sola entrada, esta forma de resolver una red la llamaremos convención 1. Si queremos que nos de la respuesta a varias entradas al mismo tiempo tendríamos que modificar un poco la representación de nuestros valores, transponiendo las matrices anteriores que teníamos. Que se verían de la siguiente forma:

$$A^T = [a_0 \ a_1 \ a_2] = [1 \ 0 \ 1]$$

$$W^T = \begin{bmatrix} w_{01} & w_{02} \\ w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$H = g(A^T * W^T)$$

$$H = [h_1 \ h_2] = g \left(\begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} \right) = g([0.5 \ 0.5]) = [1 \ 1]$$

$$o = g \left(\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \right) = g([0.5]) = [1]$$

Esta forma se llama la convención 2, lo que estamos haciendo en la convención 2 es más bien poner nuestro ejemplar de manera horizontal. Así cada renglón de A va a representar una entrada el primer valor de cada renglón siempre será el sesgo. Así al tener varias entradas de la siguiente forma:

$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

A la matriz A es X transpuesta y le agregamos los sesgos, quedando así:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

La capa oculta quedaría de la siguiente forma $H = g(W^T A)$

$$H = g \left(\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} \right) = g \left(\begin{bmatrix} -0.5 & 1.5 \\ 0.5 & 0.5 \\ 0.5 & 0.5 \\ 1.5 & -0.5 \end{bmatrix} \right) = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

El resultado de esta matriz H es que, la primera columna está representando la neurona que evalúa la función OR y la segunda columna representa el NAND. Para obtener la salida de la neurona o que es un AND, haríamos las mismas operaciones anteriores solo que con sus respectivos pesos y valores de H^T , así obteniendo lo siguiente $o = g(H^T W^T) = \text{XOR}$:

$$o = g \left(\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \right) = g \left(\begin{bmatrix} -0.5 \\ 0.5 \\ 0.5 \\ -0.5 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Más adelante esta representación, nos ayudará cuando estemos trabajando con redes neuronales que necesiten *diferentes tipos de características*, como; edades, número de hijos, tamaño de una casa, etc. Esta anotación es mucho más visual para la forma en que vamos a recibir la información (tabla de datos), por eso es más usada esta versión.

Interpretación matemática del mapeo no lineal

En esta sección vamos a ver por qué agregar una capa más nos permite calcular funciones que no era posible calcular antes es decir no basta con tener más percepciones en realidad necesitamos tener percepciones que reciban como entrada la salida de percepciones anteriores y esto lo vamos a notar aquí por eso el selector es una función tan útil para poder explicar estos conceptos de entrada que teníamos al inicio de la función sort eran los valores 0 0 0 1 1 0 y 1 1 estos solamente los podría bueno puede utilizar perceptron únicamente para evaluar funciones que sean linealmente separables entonces cuando hemos aplicado la primera capa donde pudimos poner tantas operaciones como que hicimos lo que hicimos fue tomar estas entradas y mapear las a un conjunto distinto y esté ocurriendo en este nuevo conjunto

Observen que las entradas ahora son 01 11 11 10 es decir estas dos están duplicadas ya no tengo el 0 0 por eso es que se dice que trabajamos con un mapeo no lineal nuestra función sigmoide o nuestra función escalón transformaron nuestro espacio de manera que no tenemos ahora una relación 1 a 1 qué efecto va a tener. Aquí tenemos una vez más cuáles son las entradas originales las entradas que se obtuvieron es bueno salidas de las capas ocultas que van a hacer ahora las entradas para el último perceptron entonces veamos acá qué pasó con este cuadro aquí teníamos cuatro datos que no eran linealmente

separables después de la primera capa fue como haberlo plegado a otro espacio de dos dimensiones, pero observen ahora que como que esto es bueno vamos a ver quién se mató a quién el 0 0 se convirtió en 0,1 es decir este para acá el 0,1 se convirtió en 1,1 este fue para acá el 1010 también se convirtió en 1,1 es otro de acá y el 11 se convirtió en el 10 está aquí entonces nuestros puntos negros como que giraron un poquito y quedaron aquí y nuestros dos puntos blancos quedaron mapeados uno encima del otro en el mismo punto de este lado.

Entonces ahora sí lo puedo separar con un plano qué pasa en medio de los dos cualquiera que lo logre es bueno entonces estos tres valores quedaron mapeados ahora a un nuevo espacio que por cierto tiene una dimensión solo hay un valor. Entonces el paso fundamental radicó en que pudimos mapear este espacio hacia un espacio nuevo donde si es posible separar a nuestros datos linealmente y sobre esto pues ya simplemente aplicamos lo que podía hacer cualquier perceptual matemáticamente ese es el poder que nos está añadiendo una capa de en medio y básicamente lo podemos generalizar ahora si a cualquier función. Es posible quitar los sesgos si utilizamos la función escalón como función de activación y definimos precisamente el menor o igual para la parte del escalón en esta ocasión vamos a obtener valores que están exactamente en este punto de salto así es que vamos a tener que decir quién queda a la izquierda quien quiere a la derecha y entonces podemos hacer esto igual se necesitan dos capas, pero podemos quitarnos la parte de los sesgos porque nuestras fronteras si están pasando por ser un coma se hace solo por curiosidad bien aquí tenemos entonces ya esa interpretación vemos que los ceros negativos se van a tomar como ceros y solamente los positivos van a quedar como unos y automáticamente se puede hacer esta versión resumida.

Propagación hacia adelante para el perceptrón multicapa

Ahora para el modelo de una red en general tenemos nuestra arquitectura base que va a ser una red en capas también se le conoce como el perceptron multicapa (tipo feed-forward) en esta primera versión tenemos la capa de entrada que realmente solo recibe las entradas y el sesgo. Las salidas de cada capa sirven de entradas a la capa inmediatamente posterior en la red multicapa. Por lo general todas las salidas de una capa se distribuyen a todas las neuronas de la siguiente capa, formando capas completamente conectadas (fully-connected layers). Cuando la red multicapa incluye más de una capa oculta, se dice que la red neuronal es profunda [deep neural network]. Los niveles de actividad de las neuronas de cada capa vienen dados por una función de activación no lineal de los niveles de actividad de las neuronas de la capa inferior.

(Insertar imagen de red).

Anteriormente habíamos estado asignando pesos a ojo/intuición, esto porque eran pocos los valores de entrada, esto normalmente no es así y vamos a desconocer en la totalidad los pesos necesarios que se aproximen a nuestra función objetivo.

La red neuronal es en sí, es una función que nos va permitir pasar un vector de n dimensiones a uno de m dimensiones, con n la cantidad de características en nuestros datos y m la cantidad de características que necesitamos obtener.

Entonces vamos a utilizar un tipo de codificación que se utiliza para la clasificación, llamado One-hot encoding, donde nos vamos a enforar en el valor más elevado en nuestros resultados. Ahora nuestros problemas involuran más de dos clases, donde cada renglón de nuestra matriz de salida nos va indicar si pertenece o no a una clase, algún ejemplar dado. Así tenemos cada dimensión en la matriz va a representar una clasificación, por ejemplo si queremos distinguir en una imagen entre un coche, casa, animal lo podríamos codificar de la siguiente forma:

$$\text{Salida1} = \text{coche} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\text{Salida2} = \text{casa} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\text{Salida3} = \text{vaca} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Toda la parte del aprendizaje, de reconocer distintos patrones, obtención de características varias, va a quedar entre *la capa de entrada y las capas ocultas*, para que al final nos quedemos únicamente con el problema de separar las clases unas de otras tantas sean necesarias, *en la capa de salida* y tengamos nuestra información clasificada. Así cada neurona de salida es una clase, específica, entonces si se activo esa neurona nos indica que nuestro ejemplar pertenece a dicha clase. Es importante no escatimar en el número de perceptrones necesarios para la clasificación a la salida, pues esto podría resultar en asignación conjunta de características, que se podrían interpretar como similitudes entre clases y en caso de no existir, dificultarían enormemente la distinción entre características y generar fallos a la red. Siempre asignar tantas dimensiones de salida (perceptrones) como clases.

Los calculos para cada capa intermedia realmente son las mismas que hemos visto anteriormente representandose de la siguiente forma:

$$a_j^{(l+1)} = g \left(\sum_i w_{ij}^{(l)} a_i^{(l)} \right) \quad (5.8)$$

Con $l + 1$ representando el número de capa a la que vamos, g la función activación (normalmente podemos usar la función sigmoide) y a_i los valores de las entradas.

6 | Entrenamiento por retropropagación

Esquema general entrenamiento

En este capítulo vamos a trabajar con el entrenamiento (específicamente por retropropagación), en esta sección abordaremos primero el esquema general del entrenamiento. A lo largo del capítulo veremos la derivación del primer algoritmo de entrenamiento para perceptrones multicapa (este es de los más comunes en esta área y el primero descrito propiamente).

En el tercer capítulo habíamos visto el concepto de aprendizaje para una máquina, ahora vamos a definir en que consiste aprender algo en el campo de las redes neuronales. Retomando lo descrito en ese capítulo, recordemos que aprender para un modelo es más que nada encontrar una función que se asemeje a ciertas salidas específicas, esta función está dentro de un espacio de funciones posibles propuestas. La función que deseamos encontrar modela algún tipo de tarea que nos interesa que se aprenda.

Definición 6.1

El problema de aprendizaje para un perceptrón multicapa: Dada una arquitectura de red neuronal, encontrar los pesos tales que la función $h_{\Theta}(x)$ (definida por la red neuronal) aproxime, hasta cierta tolerancia, la función de interés $f(x)$. Con x los datos de entrada.

En la definición anterior, el vector x , consisten en distintas características de los ejemplares provistos, según sea el problema a resolver, por ejemplo:

- Segmentos de alguna canción.
- Píxeles de imágenes.
- Valores equivalentes a precios del mercado.

En este momento nos estamos concentrando en problemas de clasificación así que estas redes están clasificando los ejemplares según ciertas características. Estas clasificaciones

pueden ser muy interesantes, algunas por ejemplo se pueden dedicar a clasificar canciones, identificando ciertas emociones que transmiten como tristeza o alegría.

Nuestra red neuronal va a empezar sus cálculos a partir de los datos de entrada x , entonces al finalizar cada uno de los ejemplares va a ser mapeado a alguna clase (en el espacio de clases). Por medio de una función donde $f(x)$ es nuestra función ideal que hace el mapeo exacto de los ejemplares a su clase. Así si logramos que nuestra función $h\Theta(x)$ se aproxime a $f(x)$, la red neuronal nos daría las respuestas esperadas dado ciertos ejemplares, como por ejemplo:

- Dado unas imágenes de distintas plantas, identificar correctamente el nombre de cada una.
- Dado un fragmento de canción indicarnos su género.
- Dada una oración identificar si contiene un mensaje de odio.

Entonces el reto de la red neuronal es aproximar una función $f(x)$ (que se asume existe en el universo), primero debemos intentar calcular $h\Theta(x)$ (h de hipótesis, de un espacio de hipótesis). Esta hipótesis puede que aproxime muy bien la función pero en la mayoría de los casos no es así, entonces iremos haciendo ajustes a los pesos (que definen la función h) para que pueda aproximarse lo más posible a nuestra función objetivo.

Como en muchos casos vamos a intentar aproximar funciones que no corresponden a un concepto matemático, sino más a percepciones humanas tales como sentimientos o clasificar abstracciones humanas, entonces se puede dar el caso que realmente no se pueda aproximar con total exactitud la función y le demos un cierto *grado de tolerancia*, donde se asume que se pueda dar ciertos errores en ciertos ejemplares. Dependiendo del contexto vamos a asignar el rango de tolerancia de errores.

Es importante notar que si los datos de entrenamiento resultan clasificados con mucha precisión es probable que la red esté aprendiendo datos no útiles para la tarea dada, es decir está aprendiendo *ruido*. Y al momento de pasar los ejemplares de prueba no entregue buenos resultados.

Entrenar una red para que se aproxime a la función objetivo cada vez más, debemos hacer lo siguiente:

- Definir una *función de error* o pérdida J (o L de lost, pérdida en inglés) que mida la distancia entre los valores deseados y los valores obtenidos con un conjunto de pesos dados Θ .
- Utilizar alguna *técnica de optimización* de funciones que minimice este error.
- Definir la *arquitectura* de la red, la *función de error* y la técnica de optimización, considerando el tipo de función objetivo.

Una vez que ya sabemos en que consiste el entrenando veamos un esquema general conocido como *entrenamiento por retropropagación* o propagación hacia atras, en inglés *Backpropagation*. Anteriormente habíamos visto la evaluación de una red con propagación hacia delante (de izquierda a derecha), con la retropropagación vamos a recorrer la red hacia atras (de derecha a izquierda), de la siguiente forma:

- Utilizar *la regla de la cadena* para obtener el gradiente de la función de error con respecto al conjunto de pesos, dado un conjunto de datos de entrenamiento X con sus etiquetas Y .
- Utilizar *descenso por el gradiente* para encontrar un mínimo, lo suficientemente bueno, de la función de error. Esta técnica es la más sencilla, por eso usualmente se utilizaba en un principio
 - ★ La idea es que al inicio no sabemos cuánto deben de valer los pesos para modelar correctamente la función, entonces al comienzo se hace una asignación aleatoria de pesos. Existe un conjunto de pesos ideales a los cuales queremos aproximar nuestros pesos aleatorios. Con estos calculamos el error que tiene nuestros pesos, haciendo las comparaciones con los datos que se obtuvieron al aplicar el feedforward, sobre los datos de entrada con los datos esperados. El descenso por el gradiente va modificar poco a poco los pesos, para tengamos una nueva (y mejor) aproximación y nos acerquemos más a una región donde el error sea menor. Para calcular el descenso por el gradiente, se calculan las derivadas de la función de error con respecto a los parámetros.
 - ★ Entonces tenemos una actualización de la siguiente forma:

$$\Theta' = \Theta - \alpha \nabla_{\Theta} J \quad (6.1)$$

Donde Θ es la propuesta para nuestros parametros iniciales, ∇_{Θ} es el gradiente que nos da el máximo ascenso de la función de error, el cual lo invertimos con el menos, para que siempre nos dirijamos al minimo local más proximo.

- Se puede usar otra técnica de optimización más avanzada, pero esta también involucraría al gradiente.
- La derivación original utilizaba diferencias al cuadrado como función de error. Esta la pueden consultar en Russell, Stuart y Peter Norving (2010). Artificial Intelligence, A Modern Approach.
- Para problemas de clasificación es más efectivo usar entropía cruzada como método de optimización. Para problemas de regresión es adecuado usar diferencias al cuadrado.

Función de error: Entropía cruzada

En esta sección veremos una de las funciones de error más utilizadas en el campo de redes neurales, usada sobre todo en problemas de clasificación, la entropía cruzada (en inglés *cross entropy*) la cual tiene su origen en el área de la teoría de la información, la cual mide: **la cantidad de bits necesarios para identificar una clase** dada la hipótesis de la red representada como $h_{\Theta}(x)$, que trata de asemejar la función objetivo $y(x)$, la cual es la que tiene los valores reales asociados a las etiquetas.

Uno de los labores de la red neuronal es codificar con *bits* la información que recibe para hacer asignaciones de las características encontradas y así clasificarlos. Todo el tiempo se está intentando predecir si la función propuesta se asemeja lo suficiente a la función objetivo, en caso de ser así se podrá codificar adecuadamente nuestros ejemplares.

La fórmula de entropía cruzada usada en este texto será la siguiente:

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^{s_L} y_k^i \log(h_{\Theta}(x^i))_k + (1 - y_k^i) \log(1 - h_{\Theta}(x^i))_k \right] \quad (6.2)$$

donde:

- m es el número de ejemplares del entrenamiento.
- s_L es el número de neuronas en la capa L .
- y_k es el valor para la k -ésima neurona de salida y toma valores en 0,1.

En la primera parte notamos que tenemos una fracción, esta es en esencia la que nos da el promedio de error de la red sobre el número de ejemplares de entrenamiento evaluados en ese momento (m). Seguido de una multiplicación de la suma de los ejemplares que nos da todas las contribuciones de error, que está a su vez contiene la suma de las neuronas en la capa de salida de las neuronas de cada clase. Así en esta suma podemos distinguir dos partes, la primera es una multiplicación que compone de la etiqueta real asociada al ejemplar multiplicado por el logaritmo de la etiqueta asignada por la red, y la segunda parte de una multiplicación de uno menos la etiqueta real por el log de uno menos lo dado por la red. Así en esta parte se está tomando en cuenta las dos posibilidades que puede tomar cada neurona de salida respecto al ejemplar dado, 0 no pertenece a la clase y 1 pertenece a la clase.

En el primer caso y_k toma el valor de cero, entonces la primera parte de esta sección se va también a cero, es ahí donde la segunda parte nos permite evaluar que tan lejos estuvimos de la respuesta correcta usando el logaritmo de esta. Ahora supongamos que lo deseado era que la neurona predijo 1 pero lo correcto era 0, la primera parte se va a cero y en la segunda parte tendremos un logaritmo de un número cercano a cero dándonos

que nos vamos a acercar a menos infinito, es por eso importante el signo negativo en la parte inicial de la función, pues nos va ayudar a que la función nos indique a infinito cuando estamos errados en el resultado.

Ahora en caso de obtener la respuesta deseada, con una salida de 0 cuando en efecto era este, entonces tendremos el $\log(1)$ que es cero, así nuestra función de error valdrá 0, este caso es poco común pues recordemos que para función de activación h_{Θ} usando la función logística solo se acerca al 1. El caso que lo deseado era que nos diera 1 la neurona esta contemplado en la primera parte donde de nuevo con la ayuda del logaritmo en caso de dar la respuesta incorrecta 0 este apuntara hacía menos infinito y con ayuda del menos de afuera va hacía infinito, y en caso de estar en lo correcto nos vamos a acercar al cero, dandonos como resultado un error cercano a cero.

En resumen con esta función si estamos en lo correcto vamos a acercarnos a cero, de lo contrario va a tener hacia infinito.

Derivada de la función logística

Para poder calcular el gradiente, primero debemos ver la derivada de la función logística. Recordemos en la siguiente figura como se ve la función logística.

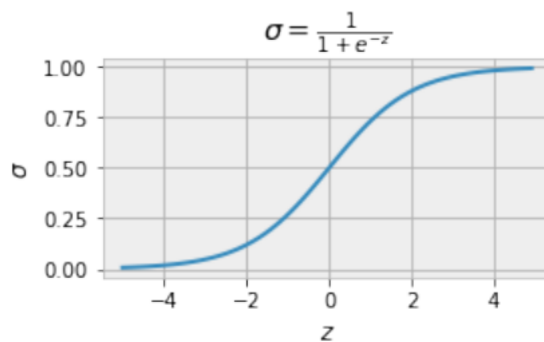
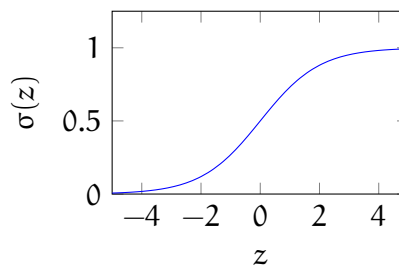


Figura 6.1 Gráfica función logística.



(a) Función logística.

Figura 6.2

La función logística estandar centrada en el origen como en la figura 6.1 se escribe de la siguiente forma:

$$\sigma(z) = \frac{1}{1 + e^{-(z)}} \quad (6.3)$$

Los siguientes calculos son realizados para un solo perceptrón, en concreto uno de salida, donde este está recibiendo los datos de salida de neuronas en la capa oculta, así el factor z que está recibiendo la función, es la combinación lineal de los valores de entrada multiplicados por los pesos asignados para este perceptrón y sumados todos. Entonces $z = X\Theta = [X_0\theta_0 + X_1\theta_1 + \dots + X_n\theta_n]$. Así nuestra función queda de la siguiente forma:

$$\sigma_{\Theta}(X) = \frac{1}{1 + e^{-X\Theta}} \quad (6.4)$$

Entonces vamos a calcular la derivada de esta función con respecto a cualquiera de estos parámetros i de z . Por la regla de la cadena comenzamos calculando la derivada queda de la siguiente forma:

$$\frac{\partial \sigma}{\partial \theta_i} = -\frac{1}{(1 - e^{-X\Theta})^2} e^{-X\Theta} (-x_i) \quad (6.5)$$

Reescribiendo la misma derivada con un poco de algebra nos queda de la siguiente forma:

$$\frac{\partial \sigma}{\partial \theta_i} = \frac{1}{1 + e^{-X\Theta}} \frac{e^{-X\Theta}}{1 + e^{X\Theta}} x_i \quad (6.6)$$

Con la reescritura anterior nos damos cuenta que tenemos la función sigma en uno de los productos así que procedemos a sustituirlo y a sumar y restar unos unos. Así obteniendo lo siguiente:

$$\frac{\partial \sigma}{\partial \theta_i} = \sigma_{\Theta}(X) \frac{e^{-X\Theta} - 1 + 1}{1 + e^{X\Theta}} x_i \quad (6.7)$$

Vamos a separar a conveniencia los terminos de la siguiente forma:

$$\frac{\partial \sigma}{\partial \theta_i} = \sigma_{\Theta}(X) \left(\frac{1 + e^{-X\Theta}}{1 + e^{-X\Theta}} - \frac{1}{1 + e^{-X\Theta}} \right) x_i \quad (6.8)$$

Así sustituyendo con la función sigmoide y llevando a uno, llegamos a lo siguiente:

$$\frac{\partial \sigma}{\partial \theta_i} = \sigma_{\Theta}(X)(1 - \sigma_{\Theta}(X))x_i \quad (6.9)$$

Entonces lo que tenemos es una propiedad bastante interesante de la función logística. Se está calculando la derivada de sigma con respecto al exponente osea que sería: Entonces lo que vemos es que la derivada de la sigmoide es la sigmoide por uno menos la sigmoide:

$$\frac{\partial \sigma}{\partial \theta_z} = \sigma(1 - \sigma) \quad (6.10)$$

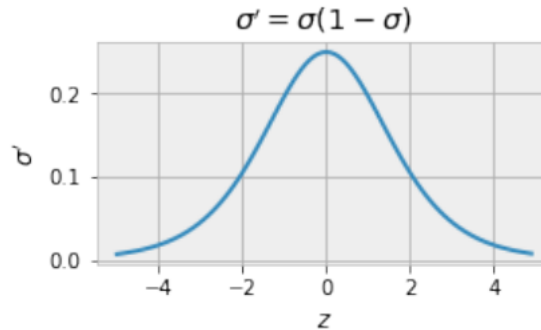


Figura 6.3 Gráfica de la derivada de la función logística.

Las funciones logísticas se utilizan a menudo en redes neuronales para introducir no linealidad en el modelo o para sujetar señales dentro de un intervalo específico. Un elemento de red neuronal popular calcula una combinación lineal de sus señales de entrada y aplica una función logística limitada como función de activación al resultado; este modelo puede verse como una variante "suavizada" de la neurona umbral clásica.

Estas relaciones dan como resultado implementaciones simplificadas de redes neuronales artificiales con neuronas artificiales. Las funciones sigmoideas que son antisimétricas con respecto al origen (por ejemplo, la tangente hiperbólica) conducen a una convergencia más rápida cuando se entrenan redes con retropropagación.

Una opción común para la activación o "aplastamiento" funciones, usadas para grandes magnitudes para mantener la respuesta de la red neuronal limitada.

Las funciones logísticas se utilizan en varios roles en estadística. Por ejemplo, son la función de distribución acumulativa de la familia logística de distribuciones y, un poco simplificadas, se utilizan para modelar la posibilidad que tiene un jugador de ajedrez de vencer a su oponente en el sistema de clasificación.

En la siguiente sección veremos la derivada de la función de error.

Entrenamiento en la última capa

En esta sección nos vamos a enfocar en la última capa de red, consideremos la siguiente red de la figura 6.4.

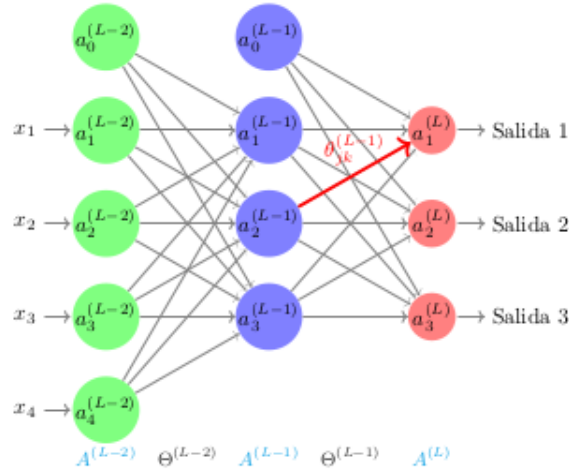


Figura 6.4 Red neuronal(entrenamiento última capa.)

Vamos a calcular el gradiente considerando que estamos evaluando a la red para un ejemplar. Una vez que la red tomó los valores de entrada (neuronas en verde) y los evaluó pasando por la capa oculta (neuronas en azul) hasta asignar el ejemplar a una neurona de salida (neuronas en rojo). Lo que la función de error está midiendo es la *distancia* entre lo que obtuvimos en la capa de salida y una colección de etiquetas objetivo, es decir, la clasificación real del ejemplar.

Entonces por ahora nuestra función de error está actuando solo sobre la capa de salida, pero recordemos que para llegar a la evaluación de cualquiera de las neuronas en la capa de salida, se calculó la activación de la capa oculta y de la capa de entrada. Con los respectivos pesos de cada capa. Entonces podemos decir que los valores obtenidos en la evaluación final dependen directamente de los pesos en cada capa.

Tomando en cuenta lo anterior, el gradiente lo vamos a calcular capa por capa. Desde la capa de salida hasta la capa de entrada. Lo primero que tenemos que calcular es como dependió la función de error de los pesos de la capa oculta (azul) que la antecede.

La notación que vamos a usar más adelante va a considerar las neuronas de entrada con el índice i , las neuronas de la capa oculta con el índice j , y las neuronas de salida con el índice k . Así, los pesos que conectan a la neurona j con la neurona k , se representan como θ_{jk} .

Para denotar la capa que estamos haciendo referencia vamos a emplear el superíndice (L) con referencia a last, último en inglés, dado que estamos haciendo los cálculos desde la última capa, hacia atrás, la penúltima capa se va a indicar con $(L-1)$, la anterior de esta como $(L-2)$ y así sucesivamente. Entonces, para referirnos a un peso para la última capa, que conecta a la penúltima capa con la de salida, se escribe como θ_{jk}^{L-1} que es el peso marcado en rojo en la figura 6.4.

Usemos la entropía cruzada como función de error, para un ejemplar 6.2:

$$J(\Theta) = - \left[\sum_{k=1}^{s_L} y_k^{(i)} \log(a_k^{(L)}) + (1 - y_k^{(i)}) \log(1 - a_k^{(L)}) \right] \quad (6.11)$$

La suma está tomando en cuenta que el ejemplar puede ser clasificado a una de las varias clases a poder asignar s_L , es decir, el número de neuronas en la capa L . Los valores obtenidos de la activación las neuronas están definidos por $a_k^{(L)}$, y los deseados por y_k .

Lo siguiente a hacer es calcular la derivada con respecto a los pesos que conectan la última capa con la penúltima capa, es decir, $\theta_{jk}^{(L-1)}$.

Entonces vamos a calcular la parcial de error con respecto a uno de los pesos de la capa anterior. Así la suma se "va" dado que el peso solo va a afectar a la k -ésima neurona, quedando solo con la operación que se efectúa en esta, usando la regla de cadena obtenemos lo siguiente:

$$\frac{\partial J}{\partial \theta_{jk}^{L-1}} = - \left[\frac{y_k}{a_k^{(L)}} \frac{\partial}{\partial \theta_{jk}^{(L-1)}} g(z_k) - \left(\frac{1 - y_k}{1 - a_k^{(L)}} \right) \frac{\partial}{\partial \theta_{jk}^{(L-1)}} g(z_k) \right] \quad (6.12)$$

Con:

$$z_k = \sum_{j'=0}^{s_{L-1}} \theta_{j'k} a_{j'}^{L-1} \quad (6.13)$$

Recordemos que $a_k^{(L)}$ está calculado por una función de activación $g(z_k)$. Donde z_k es la suma de las combinaciones lineales de las neuronas conectadas hacia la neurona k_n . Estos valores son calculados en el algoritmo de propagación hacia adelante. Los escribimos aquí para recordar como está siendo calculada la parcial.

De la derivada de la suma sólo queda $a_{j'}$ con $j' = j$. Siguiendo la regla de la cadena para los términos que nos faltaba nos queda lo siguiente:

$$\frac{\partial J}{\partial \theta_{jk}^{L-1}} = - \left[\frac{y_k}{a_k^{(L)}} g'(z_k) - \left(\frac{1 - y_k}{1 - a_k^{(L)}} \right) g'(z_k) \right] a_j^{(L)} \quad (6.14)$$

Simplificando, nos queda lo siguiente:

$$\frac{\partial J}{\partial \theta_{jk}^{L-1}} = - \left[\frac{y_k(1 - a_k^{(L)}) - a_k^{(L)}(1 - y_k)}{a_k^{(L)}(1 - a_k^{(L)})} \right] g'(z_k) a_j^{(L)} \quad (6.15)$$

Recordando que la derivada de la sigmoide queda como $g' = g(1 - g)$ tenemos que:

$$g'(z_k) = a_k^{(L)}(1 - a_k^{(L)}) \quad (6.16)$$

Así nos queda como:

$$\frac{\partial J}{\partial \theta_{jk}^{L-1}} = - \left[\frac{y_k - y_k a_k^{(L)} - a_k^{(L)} + a_k^{(L)} y_k}{a_k^{(L)} (1 - a_k^{(L)})} \right] a_k^{(L)} (1 - a_k^{(L)}) a_j^{(L)} \quad (6.17)$$

Y haciendo un poco de álgebra nos queda como:

$$\frac{\partial J}{\partial \theta_{jk}^{L-1}} = -a_j^{(L-1)} (y_k - a_k^{(L)}) \quad (6.18)$$

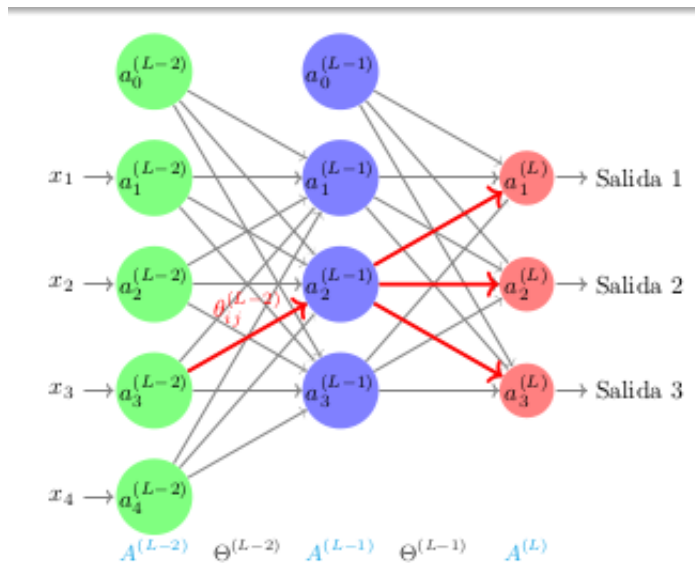
Representando a $(y_k - a_k^{(L)})$ como δ_k nos queda:

$$\frac{\partial J}{\partial \theta_{jk}^{L-1}} = -a_j^{(L-1)} \delta_k \quad (6.19)$$

Esta última notación con delta se usa pues con esto estamos representando el error que se cometió en la última capa, pues es la diferencia entre lo desado y lo que se obtuvo realmente.

Con esto ya sabemos calcular todas las parciales del error con respecto a los pesos hacia la última capa. En la siguiente sección seguimos con el cálculo para las siguientes capas.

Parcial con respecto a los pesos en la última capa



En esta sección vamos a trabajar con la última capa, vamos a tener un poco más de elementos pero va a ser sencillo agregarlos en primer lugar vamos a tener que observar quienes se ven afectados por el nuevo peso con respecto al cual queremos derivar bien aquí tenemos un poco más de transmisión de efecto este peso solamente afecta a esta neurona pero a través de esta neurona. ahora si estamos conectados con absolutamente todas las otras neuronas.

Retomando la entropía cruzada de la sección anterior 6.11

$$J(\Theta) = - \left[\sum_{k=1}^{s_L} y_k^{(i)} \log(a_k^{(L)}) + (1 - y_k^{(i)}) \log(1 - a_k^{(L)}) \right] \quad (6.11)$$

Capa anterior:

$$\frac{\partial J}{\partial \theta_{jk}^{L-2}} = - \sum_{k=1}^{s_L} \left[\frac{y_k}{a_k^L} \frac{\partial}{\partial \theta_{jk}^{(L-2)}} g(z_k) - \left(\frac{1 - y_k}{1 - a_k^L} \right) \frac{\partial}{\partial \theta_{jk}^{(L-2)}} g(z_k) \right] \quad (6.20)$$

Con:

$$z_k = \sum_{j'=0}^{s_{L-1}} \theta_{j'k} a_{j'}^{L-1} \quad (6.21)$$

Entonces tenemos que:

$$\frac{\partial J}{\partial \theta_{jk}^{L-2}} = - \sum_{k=1}^{s_L} (y_k - a_k^L) \theta_{jk} \frac{\delta}{\delta \theta_{Lj}^{(L-2)}} a_j^{(L-1)} \quad (6.22)$$

Sustituyendo a $(y_k - a_k^L) = \delta_k$ y a $a_j^{(L-1)} = g(z_j)$ nos queda:

$$\frac{\partial J}{\partial \theta_{jk}^{L-2}} = - \sum_{k=1}^{s_L} (\delta_k) \theta_{jk} \frac{\delta}{\delta \theta_{Lj}^{(L-2)}} g(z_j) \quad (6.23)$$

Con:

$$z_j = \sum_{i'=0}^{s_{(L-2)}} \theta_{i'j}^{L-2} a_{i'}^{(L-2)} \quad (6.24)$$

Así finalmente tenemos que:

$$\frac{\partial J}{\partial \theta_{jk}^{L-2}} = - \sum_{k=1}^{s_L} \delta_k \theta_{jk} g'(z_j) a_i^{(L-2)} = \delta_j^{(L-1)} a_i^{(L-2)} \quad (6.25)$$

Entonces aquí vamos a tener que hacer los toman en cuenta más términos clientes para poder hacer esta derivación voy a tener que volver a empezar desde un principio es decir

vamos a volver a empezar con la función de error original y vamos a tener que empezar a derivar desde aquí vamos a recordar ahora que cada vez que calculamos los valores de activación en la última capa bueno estos provienen en realidad también dependen de el valor de activación en esta neurona de acá atrás entonces eventualmente vamos a tener que regresar hasta acá y el valor de esta neurona pues dependió ahora sí directamente del peso que nos está interesando entonces vamos a ir desarrollando poco a poco esa composición de funciones entonces comencemos con el mismo paso de antes calculamos la derivada parcial de jota con respecto al peso correspondiente pero ahora si no puedo eliminar la suma que tengo al inicio porque todos los valores de salida dependen de el peso con el que estamos trabajando entonces lo que vamos a tener aquí ahorita es que nos quedamos con la suma lo único que estoy haciendo ahora es pasarla junto con el signo menos recordemos que la derivada de una suma es la suma de las derivadas entonces podemos meter inmediatamente nuestro problema de derivar a la parte de adentro y simplemente dejar la suma que fuera entonces la primera parte se ve idéntica que antes esto es una constante derivada del logaritmo eso no entre acá para poder calcular esta parcial entonces tenemos que recordar como estaba escrita y lo mismo va a ocurrir de este lado va a salir del signo menos por lo que teníamos acá adentro y tenemos entonces 1 - entre 1 - acá por la parcial de estar acá es lo que estamos sustituyendo acá ya hicimos todas las cuentas todo lo que se cancela entonces no es necesario volverlo a hacer simple y sencillamente llegamos a lo que ya teníamos antes era ye menos llega a menos acá todo esto a lo que se le había llamado delta acá y lo que vamos a empezar ahora es a sacar un poquito los detalles estábamos antes derivando con respecto antes de vestido con respecto a j_k y solamente nos había quedado un término aquí en este caso estamos derivando con respecto a y y $jota$ y quien depende de y y $jota$ pues son las a es básicamente estos pesos son constantes. Entonces de la primera capa que el error cometido por la última capa lo podíamos ver directamente como la diferencia entre lo que queríamos y lo que logramos calcular esto tiene una característica bastante interesante es exactamente por cada neurona tenemos uno de estos después venía la parcial de j con respecto a cada uno de los pesos observemos que por cada uno de estos errores vamos a tener de hecho tantos componentes de éstas como pesos estaban conectados con la carísima neurona entonces de estos tenemos un montón y lo único que teníamos que hacer era multiplicar este valor único por el valor de activación de la neurona con la cual estaba conectado y bueno aquí el signo menos que venimos cargando por la definición de la función de error y ya está ahora equipo se ve con estas tres neuronas bien en la definición de este delta j donde tenemos un producto de las delta casa que venían capa que está más hacia adelante multiplicados por los pesos correspondientes tenemos una suma sobre todos los elementos en última capa y una vez que hicimos esto viene multiplicar por función prima evaluada en la receta j observemos una vez más que está nada más depende de $jota$ no depende de las casas entonces por eso lo podríamos poner entonces realmente le suma nada más afecta este ya que tenemos entonces este producto este que está aquí.

Resumiendo tenemos:

$$\delta_k^{(L)} = (y_k - a_k^{(L)}) \quad (6.26)$$

$$\delta_j^{(L-1)} = \left(\sum_{k=1}^{S_L} \delta_k^{(L)} \theta_{jk} \right) g'(z_j) \quad (6.27)$$

Y sus respectivas derivadas:

$$\frac{\partial J}{\partial \theta_{jk}^{(L-1)}} = -a_j^{(L-1)} \delta_k^{(L)} \quad (6.28)$$

$$\frac{\partial J}{\partial \theta_{ij}^{(L-2)}} = -a_j^{(L-2)} \delta_j^{(L-1)} \quad (6.29)$$

Otra vez tenemos uno por cada neurona en la capa de en medio entonces aquí tenemos las cosas aunque vamos a tener las jotas y tenemos uno por cada una de estas este se suele interpretar también como la contribución el error o el error que cometieron todas las neuronas en la capa de en medio y observen que el error de cada neurona pues realmente tiene una es la suma de como contribuyó al error de todas las neuronas que estaban en la capa siguiente pues tiene mucho sentido no se están participando en todos lados pues su error es la suma de todos los errores a los cuales contribuyó y entonces para calcular el gradiente otra vez nos queda una fórmula bastante sencilla es multiplicar este único error por el valor de activación de la neurona con la cual estaba conectada en la capa anterior y de esta manera podemos obtener todas las parciales con respecto a los pesos que estaban conectados con estas neuronas y aquí tenemos todos esos y este que quedaría la parte de hacer el cálculo directamente el siguiente problema que vamos a tener es bueno si podríamos implementar perfectamente ya con un algoritmo de descenso por el gradiente con estas fórmulas que tenemos aquí. Simplemente tendremos un montón de ciclos for para estar calculando todas estas sumas y multiplicaciones sin embargo la forma en la que se acostumbra a trabajar ahora con las redes neuronales no es directamente calculando esto componente por componente sino que lo vamos a utilizar con notación matricial esto va a tener varias ventajas por un lado la notación va a ser muchísimo más compacta y por otro lado va a permitir en la implementación de los algoritmos con procesadores con gpu de manera que estas operaciones se están realizando en paralelo y entonces trabajamos muchísimo más rápido con las redes neuronales y realmente el estar utilizando notación matricial va a tener un impacto directo sobre el tiempo que tardan en ejecutarse nuestros algoritmos.

Vectorización

La vectorización.

7 | Optimización del entrenamiento

Problemas en redes profundas

Hasta este momento ya se ha visto como modelar desde una neurona hasta una red neuronal propiamente, se ha visto como hacer la configuración de tal manera que nos clasifique ejemplares, así como detectar el error en los pesos asignados y ajustarlo para mejores resultados. Ahora al momento de modelar redes neuronales profundas (en inglés deep neuronal networks, *DNN*) tenemos que aceptar que el cálculo de estas asignaciones y ajustes requieren tiempo de cálculo, así llegando al *primer problema, el tiempo de computación*. Y además nos podemos encontrar con *el problema del sobreaprendizaje* este aparece cuando tenemos demasiadas hipótesis válidas pero no de suficientes datos para poder descartar todas menos la correcta.

Cuando ajustamos los parámetros de una red neuronal a los datos del conjunto de entrenamiento, no podemos diferenciar las características realmente útiles de las irrelevantes o de las debidas al muestreo del conjunto de entrenamiento, por lo que siempre estamos expuestos al riesgo de sobreajuste (overfitting en inglés).

Métodos de regularización o la disminución de pesos o la dispersión, se puede aplicar durante el entrenamiento para combatir el sobreajuste. Alternativamente, la regularización de dropout, omite aleatoriamente neuronas de las capas ocultas durante el entrenamiento. Finalmente, los datos se pueden aumentar a través de métodos como el recorte y la rotación, de modo que los conjuntos de entrenamiento más pequeños se pueden aumentar de tamaño para reducir las posibilidades de sobreajuste.

Las DNN deben considerar muchos parámetros de entrenamiento, como el tamaño (número de capas y número de unidades por capa), la tasa de aprendizaje y los pesos iniciales. El barrido a través del espacio de parámetros para obtener parámetros óptimos puede no ser factible debido al costo en tiempo y recursos computacionales. Varios trucos, como el procesamiento por lotes (calcular el gradiente en varios ejemplos de entrenamiento a la vez en lugar de ejemplos individuales) aceleran el cálculo, lo veremos más adelante. Las grandes capacidades de procesamiento de las arquitecturas de muchos núcleos (como GPU) han producido aceleraciones significativas en el entrenamiento.

Para mayor información del tema se sugiere leer el siguiente enlace:

CHAPTER 5 Why are deep neural networks hard to train? <http://neuralnetworksanddeeplearning.com/chap5.html>

El siguiente enlace les puede ayudar a ver como cada hiperparametro puede afectar a los resultados de la red neuronal. <https://quetzalcoatl.fciencias.unam.mx/moodle/mod/url/view.php?id=634&redirect=1>

Para ayudarnos a los calculos con el gradiente vea los siguientes enlaces:

- <https://youtu.be/nUUqwaxLnWs>
- <https://youtu.be/FDCfw-YqWTE>

Gradiente desvaneciente (o que explota)

El gradiente desvaneciente o que explota en ingles *Gradient descent* es un método general de minimización para cualquier función f . En redes neuronales el gradiente es un cálculo que nos permite saber cómo ajustar los parámetros de la red de tal forma que se minimice su desviación a la salida. Entonces retomando lo que hacemos en el entrenamiento una vez calculadas las salidas de las neuronas es calcular los deltas asociados a las diferentes neuronas de la red, para lo que recorremos la red hacia atrás(desde la capa de salida hasta la capa de entrada):

Con las ecuaciones que ya vimos anteriormente (insertar ecuaciones).

Una vez realizada la propagación hacia atrás de los errores, actualizamos los pesos de la red utilizando la expresión asociada al gradiente.

(insertar ecuaciones :P).

En el ajuste de un peso asociado a la entrada x_i de una neurona, intervienen tres factores:

- el valor de la entrada x_i
- la derivada de su función de activación dada su entrada neta
- una señal de error que depende de la capa en la que nos encontremos y que proviene de las siguientes capas de la red.

En el algoritmo de propagación de errores, partimos de una señal de error observada en la capa de salida de la red y vamos propagando esa señal de error hacia atrás por toda la red. Pudimos notar que el gradiente del error de una neurona oculta se podía calcular combinando los gradientes del error para las neuronas de la siguiente capa de la red.

Si el nivel de activación de una neurona es bajo, los pesos asociados a las sinapsis que reciben como entrada la salida de la neurona se ajustarán lentamente. La presencia de pesos elevados en los caminos desde la neurona oculta hasta la salida indicarán una contribución elevada al gradiente del error. En el momento en el que una neurona (sigmoideal) se satura, los gradientes del error serán muy bajos y la neurona oculta dejará de aprender, ya que los ajustes que realizamos sobre sus pesos son proporcionales a la derivada de su función de activación. En el mejor de los casos, aprenderá muy lentamente.

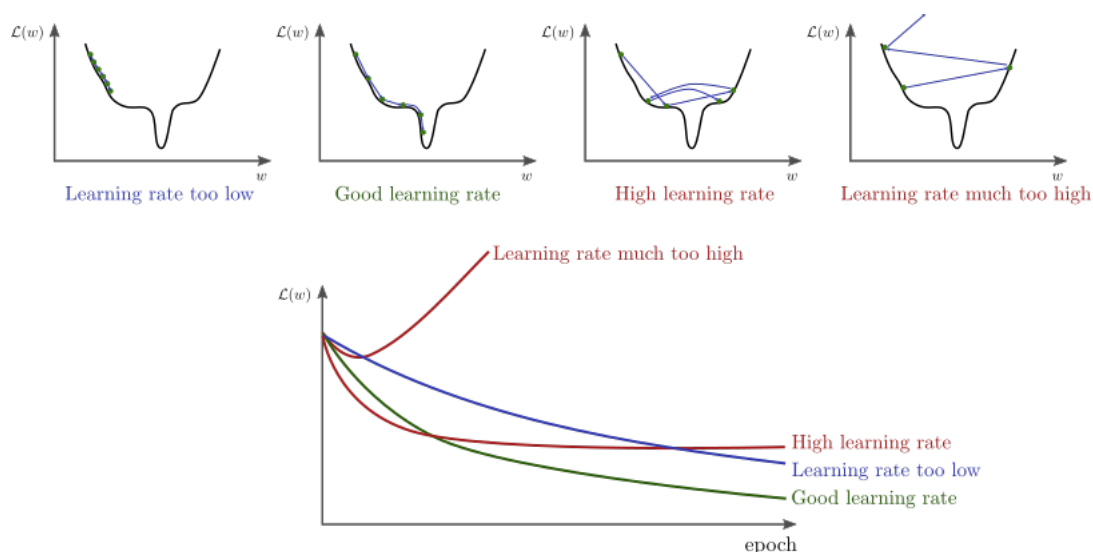


Figura 7.1 Rehacerla bien. por copyright..

Lo más habitual es que las capas más alejadas de la salida de la red aprendan más lentamente que las capas más cercanas a la salida. Dado que los niveles de activación de las neuronas están acotados y los resultados a menudo son menores que 1. En casos extremos, el gradiente será prácticamente nulo, por lo que la red será incapaz de aprender. Dándonos el problema conocido como gradiente desvaneciente en inglés *vanishing gradient*.

También puede darse el caso opuesto. Si los pesos de la red aumentan en exceso durante el entrenamiento de la red (p.ej. si utilizamos una tasa de aprendizaje demasiado elevada que hace que el gradiente descendente sea inestable y diverja), los factores involucrados en el cálculo del gradiente tomarán valores mucho mayores que 1. En este caso, el problema es el contrario al gradiente devanescente y tendríamos la explosión del gradiente *exploding gradient*.

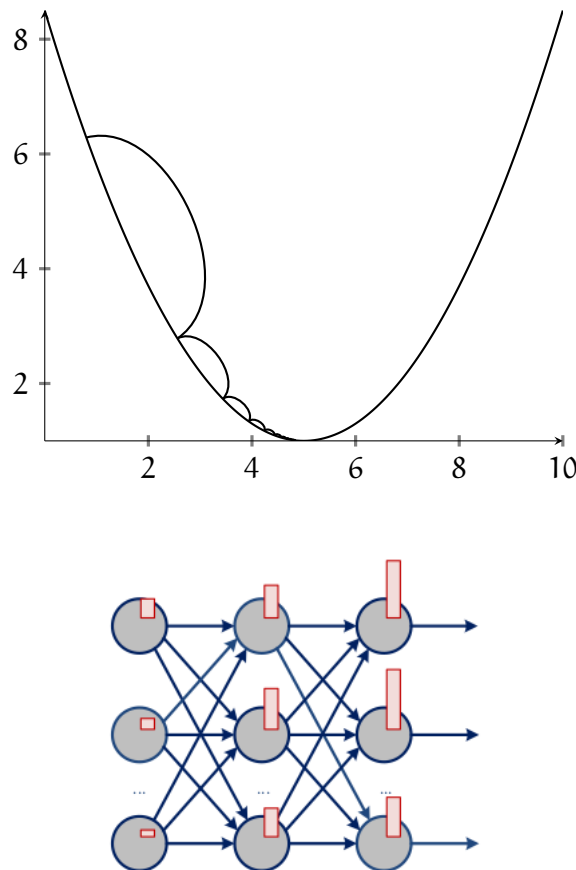


Figura 7.2 La magnitud del gradiente varía de una capa a otra durante el entrenamiento de una red neuronal multicapa

En resumen, el aprendizaje basado en gradiente descendente usando retropropagación puede no funcionar demasiado bien cuando el nivel de activación de una neurona j es bajo (lo que afecta al ajuste de los pesos de las neuronas que reciben como entrada la salida de la neurona j), las neuronas sigmoideas se estancan (por lo que las derivadas de sus funciones de activación serán bajas, lo cual afecta al entrenamiento de sus pesos y, potencialmente, al de las neuronas que las preceden en la red) o la profundidad de nuestra red hace que se vea afectada por problemas como desvanecimiento o explosión del gradiente.

Estilos de entrenamiento

Las varias formas de entrenar una red se dividen en la frecuencia de la actualización de pesos, y el ajuste de sus parámetros, para ello tenemos tres formas:

El entrenamiento en línea en inglés *batch learning* tiene esencialmente el siguiente algoritmo 7.3. Este se caracteriza en que la red es entrenada *ejemplar por ejemplar*,

7. Optimización del entrenamiento

los pesos son actualizados con cada ejemplar (más frecuencia), haciendo que durante el entrenamiento el gradiente se mueva a direcciones diferentes, por tanto es más propenso a que se llegé a dar un gradiente desvaneciente o una explosión de este. El algoritmo se da a continuación en la fig??.

Algoritmo 1 Entrenamiento en línea.

```

1: for each  $x_i$  in  $X$  do
2:    $h_i \leftarrow \text{PROPAGACIÓN HACIA ADELANTE}(x_i)$ 
3:    $\nabla_{\Theta} J \leftarrow \text{PROPAGACIÓN HACIA ATRÁS}(h_i, y_i)$ 
4:    $\Theta \leftarrow \text{OPTIMIZA}(\Theta, \nabla_{\Theta} J)$ 
5: end for

```

Figura 7.3 Algoritmo en línea.

El entremiento en lotes en inglés *online learning* es cuando se ajustan los pesos de la red tras haber evaluado el gradiente sobre *todos los ejemplares* del entrenamiento. El cálculo del grandiente es más presiso puesto que se toman varios ejemplares en cuenta. Pero esto nos cuesta en el cómputo de la actualización de los parámetros. No es recomendable este entremiento cuando la red aún está lejos de los valores deseados. El algoritmo se da a continuación en la fig??.

Algoritmo 2 Entrenamiento en

```

1:  $H \leftarrow \text{PROPAGACIÓN HACIA ADELANTE}(X)$ 
2:  $\nabla_{\Theta} J \leftarrow \text{PROPAGACIÓN HACIA ATRÁS}(H, Y)$ 
3:  $\Theta \leftarrow \text{OPTIMIZA}(\Theta, \nabla_{\Theta} J)$ 

```

Figura 7.4 Algoritmo en lotes.

El entremiento en mini lotes en este se divide el conjunto completo de entrenamiento X en bloque más pequeños X_k y se entrena para cada uno de ellos. Este entremiento busca un equilibrio entre el entrenamiento en línea y el entramiento por lotes, dado que no siempre vamos a tener la taza ideal de apredizaje para que el gradiente sea preciso, pero tampoco la capacidad de cómputo para entrenar por lotes.

Algoritmo 3 Entrenamiento en minilotes.

```

1: for each  $X_k$  in  $X$  do
2:    $H_k \leftarrow \text{PROPAGACIÓN HACIA ADELANTE}(X_k)$ 
3:    $\nabla_{\Theta} J \leftarrow \text{PROPAGACIÓN HACIA ATRÁS}(H_k, Y_k)$ 
4:    $\Theta \leftarrow \text{OPTIMIZA}(\Theta, \nabla_{\Theta} J)$ 
5: end for

```

Figura 7.5 Algoritmo en minilotes.

Ahora veamos los pros y contra de cada entrenamiento en el entrenamiento por lotes sólo se realiza una actualización de los pesos en cada época de entrenamiento. Para que el cálculo del gradiente del error sea fiable, se necesitará un conjunto de entrenamiento

grande. El coste computacional del cálculo del gradiente será proporcional al tamaño del conjunto de entrenamiento. Si nuestro conjunto incluye millones de ejemplos, como suele ser habitual en big data, el coste de una iteración del algoritmo puede resultar muy costoso. Dado que, se necesitarán múltiples iteraciones para conseguir que el algoritmo converja, el coste computacional del entrenamiento de la red puede ser demasiado elevado.

Así tal vez no nos convenga hacer el cálculo para todos los ejemplares, entonces si tomamos una pequeña cantidad de los ejemplares tendríamos una estimación aproximada del gradiente para guiarnos. Cada vez que queramos darle un ajuste a los parámetros de nuestra red.

Ahora en el entrenamiento en línea, la estimación del gradiente, dependerá del ejemplo concreto que se escoja en cada momento, por lo que nos moveremos haciendo zigzag sobre la superficie de error. En problemas de clasificación, conviene evitar mostrarle a la red todos los ejemplos de una clase antes de pasar a los de la clase siguiente. Se recomienda mezclar bien los ejemplares para evitar cambios bruscos en el cálculo del error.

El aprendizaje online y el aprendizaje con minibatches son dos formas diferentes del gradiente descendente estocástico. El uso de minibatches suele funcionar mejor que el aprendizaje online pero si no disponemos de hardware paralelo, el aprendizaje online suele ser más rápido. Si usamos aprendizaje online, es decir $k = 1$, el uso de un único ejemplo de entrenamiento introducirá errores significativos en nuestra estimación del gradiente. En realidad, no necesitamos una estimación demasiado precisa, sólo una que nos permita movernos en una dirección adecuada, que nos facilite ir reduciendo el error de la red.

Normalización y normalización por lotes

Un factor importante a tomar en cuenta para un buen desempeño del entrenamiento son las magnitudes de los datos de entrada a la red. Entonces para mejorar el desempeño del algoritmo de optimización nos conviene pre-procesar los datos de entrada. Esto mediante una técnica que se llama normalización (*normalization* en inglés).

La técnica que hemos utilizado son técnicas de optimización basadas en el gradiente, la cual a una función de error se calcula el gradiente, este nos da la dirección del máximo descenso, para hacer aproximaciones discretas en los parámetros hasta que tratemos de llegar al mínimo de la función.

Para poder realizar lo anterior necesitamos que las magnitudes de los datos de entrada no estén muy dispersas, es decir que no nos encontremos con situaciones donde estemos manejando decimales para unos datos y para otros unidades de millones. En estos casos la función de error nunca va a terminar de ajustarse correctamente a estos datos, pues en ocasiones los parámetros para ajustar, van a avanzar de forma distorsionada. Dando la impresión que para unos datos el gradiente avanza muy rápido al centro, mientras que

para otros datos avanza muy lento, esto porque vamos a tener curvas de nivel demasiado elípticas (ver la Figura 7.6), entonces al tomar la dirección que nos da el gradiente, nos va a mover desplazarse muy violentamente en algunas y muy lento para otras. Cuando se intenta hacer descenso por el gradiente en estas regiones lo que va a pasar es que el vector va a oscilar muy violentamente y nos va a dificultar mucho llegar al mínimo.

Por el contrario, si las magnitudes con las que trabajamos son del mismo orden y contribuyen numéricamente de una manera más proporcionada al error, entonces vamos a tener curvas de nivel más circulares (ver la Figura 7.6). Cuando calculemos el gradiente, la perpendicular se aproximará durante más tiempo a la dirección de máximo descenso.

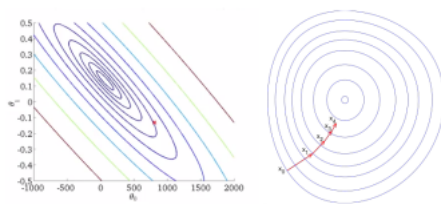


Figura 7.6 Izq: J la función de error con alta excentricidad. Der: J con curvas de nivel tendiendo a círculos

Entonces para asegurar un buen descenso por el gradiente, vamos a preferir siempre trabajar con curvas de nivel circulares, para esto vamos a usar la normalización.

La normalización consiste en centrar los datos aproximadamente en el intervalo $[-1.0, 1.0]$, este intervalo no es estricto para todas las redes basta con definir las magnitudes en un intervalo aceptable para tener curvas de nivel circulares.

Existen varias fórmulas para realizar la normalización, se sugiere la siguiente forma:

- Calcular *la media* μ_i y *varianza* σ_i^2 para cada característica i en los datos del conjunto de entrenamiento X .

★ Las formulas son las siguientes, con X_i la columna con la i -ésima característica en los datos de entrenamiento X .

$$\mu_i = \frac{1}{m} \sum_{i=1}^m x_i \quad (7.1)$$

$$\sigma_i^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2 \quad (7.2)$$

$$X_i = \frac{X_i - \mu}{\sigma^2} \quad (7.3)$$

- La media es el promedio de los datos de entrada.

- La varianza es una medida de dispersión, para ver que tan separados están los datos unos de otros.
- La reasignación para cada X_i va a ser, la resta $X_i - \mu$, que va a centrar los datos alrededor del cero, y la división por la varianza σ^2 que va a encoger los intervalos de distancia entre los datos.
- Una vez que la red se entreno con datos normalizados, *es necesario almacenar las medias y varianzas* utilizadas durante el entrenamiento con el conjunto de entrenamiento X .
- Estos valores serán utilizados para normalizar datos nuevos que vayan a ser evaluados en la red. Esto para evitar que los nuevos datos usen magnitudes fuera del intervalo usado durante el entrenamiento de la red.

Definición 7.1

La *normalización* permite reducir la excentricidad en la función de error provocada por la disparidad entre los datos de entrada.

Entonces ya tenemos resuelta la situación de la disparidad entre los datos de entrada. Ahora tenemos que, al calcular los valores para las capas intermedias, los valores pueden cambiar sus rangos para las neuronas intermedias. Para esta situación tenemos la normalización por lotes (*batch normalization* en inglés).

La normalización por lotes aplica los beneficios de la normalización a las capas intermedias, haciendo que éstas den valores en intervalos no muy grandes para las siguientes capas.

Una ventaja que nos da esta herramienta es que los algoritmos de optimización podrán utilizar *tazas de aprendizaje más altas*, porque los brinco discretos del algoritmo no cambiarán drásticamente el comportamiento de la función de error durante un intervalo más largo.

Otra ventaja es que al normalizar entre capas, permite que cada capa calcule características distintas, independientemente que se otorguen ejemplares con mucho sesgo en una característica en especial. Permitiendo una mejor clasificación aún con datos nuevos.

Definición 7.2

La *normalización por lotes* consiste en:

1. Normalizar la salida de la capa de activación anterior restando su media y

dividiendo entre la desviación estándar.

2. Hacer descenso por el gradiente estocástico.
3. Dos parámetros: γ una desviación estándar y β una media para corrimiento.

Cuando se use el descenso por el gradiente estocástico (el descenso por el gradiente por lotes) modificará los pesos para optimizar J la función de error, y probablemente contrarreste el efecto de la normalización. Para evitarlo, se añaden dos parámetros, también entrenables: γ una desviación estándar y β una media para corrimiento, la idea es que el algoritmo tienda a modificarlos, γ para escalarlos y β para sesgar los datos, en lugar de a los pesos, de modo que los pesos produzcan un cómputo más estable.

Entonces la idea concreta es que dado un minilote B con x_i sus ejemplares, las entradas y_i para la capa siguiente se calculan con las siguientes formulas:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad (7.4)$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad (7.5)$$

$$x_i = \frac{X_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \quad (7.6)$$

$$y_i = \gamma x_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad (7.7)$$

con ε una constante agregada para mantener la estabilidad numérica.

Una vez que hayamos hecho la normalización y hecho la reasignación de datos, nos va a dar los nuevos y_i 's que van a entrar para participar en la combinación lineal que decide si se activan o no las neuronas de la siguiente capa. Los valores γ y β nos van a permitir desnormalizar los datos.

Así llegamos a la parte de *inferencia*, una vez entrenada la red con normalización y normalización por lotes. Vamos a querer usar la red para hacer inferencias, para esto vamos a calcular la media y la varianza sobre los *mini lotes de entrenamiento* la diferencia más grande con respecto a lo anterior la vamos a tener en el cálculo de la varianza, porque se utiliza la variante sin sesgos, estos valores se van a estimar con los mini lotes del conjunto de entrenamiento y estos van a quedar como *constantes fijas*. Cuando entren los nuevos valores que estamos tratando de evaluar con nuestra red neuronal, ya no vamos a calcular otra vez las μ y σ para las capas intermedias. Sino que vamos a utilizar:

$$E[x] = E_B[\mu_B] \quad (7.8)$$

$$\text{Var}[x] = \frac{m}{m-1} E_B[\sigma_B^2] \quad (7.9)$$

Que son los promedios con los mini lotes que entrenamos, entonces los valores modificados de los valores de activación de cada capa van a estar dados por:

$$y = x \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} + \quad (7.10)$$

Donde y toma el valor x de activación con una red neuronal normal después multiplicarlo por el parámetro γ (que ya tuvo que haber sido aprendido por el algoritmo de entrenamiento) se va a dividir entre esta desviación estándar pero calculada sobre los datos estadísticos el de lo que salió en esas capas intermedias cuando estábamos realizando el entrenamiento.

Entonces en resumen la idea es la siguiente, dada una capa de la red neuronal con sus valores de activación, antes de pasar a la siguiente capa tenemos que pasar por un proceso por un proceso, *la normalización* para ese proceso necesitábamos la media y la varianza. Sin embargo mientras estábamos entrenando esta media y esta varianza podían ser alterados porque estamos modificando los pesos eso modifique los valores que se estaban calculando aquí y por consiguiente estos dos se van a volver a modificar. Una vez que ese proceso ya terminó y ya decidimos que están fijos los pesos ya tenemos los valores de β y γ que vamos a utilizar, es cuando se va a proceder a calcular la última versión de las medias y las varianzas, con los datos de entrenamiento.

Cuando tengamos ya estos valores estables porque ya no estamos modificando nada en la red estos los vamos a guardar y se van a convertir en los valores que vamos a utilizar cuando queramos evaluar un dato nuevo sobre la red que ya está entrenada entonces ahora sí cuando llegue a un dato nuevo cada vez que pasemos por alguna de las capas intermedias vamos a tomar esos valores de activación y vamos a aplicar lo equivalente a la fórmula que teníamos antes esta que está aquí nada más que ahora vamos a utilizar estas constantes que se derivan del escalamiento que tuvimos que hacer cuando estábamos entrenando entonces esto que está acá va a ser el valor que va a entrar a la siguiente capa de la red en lugar de haber utilizado el valor de activación como y una vez que hayamos hecho esto entonces nuestra red también va a tener el mismo tipo de normalización para datos que no habían estado en el conjunto de entrenamiento y además pues ésta es normalización es ya no van a estar cambiando porque dependen nada más de los parámetros.

La mayoría de las apis para el desarrollo de redes neuronales ya tienen estos calculos programados por detrás y nosotros realmente lo único tenemos que hacer es programarla como una capa, que se le conoce como *capa de normal por lotes*.

Regularización

La última técnica a usar en este texto es la técnica de regularización, esta es usada para librarnos de dos problemas al momento de entrenar. Estos problemas son el *sobre ajuste* y el *ajuste pobre*, en inglés *overfitting* y *underfitting* respectivamente.

Recordemos que dado un problema de aprendizaje, se define un espacio de hipótesis, es decir, una familia de posibles funciones que vamos a utilizar para encontrar la función objetivo, que queremos aproximar.

El espacio de hipótesis puede ser tan grande o tan pequeño como la cantidad de funciones que estén contenidas en él. Así tenemos dos situaciones:

- Si el espacio de hipótesis, no contiene a la función objetivo, entonces por más que ejecutemos los algoritmos de entrenamiento, no va a ser posible llegar a una buena aproximación.

Ejemplo 7.1. Si la función objetivo tiene la forma de una parábola, y únicamente la intentamos aproximar con funciones lineales, es claro que nunca vamos a dar una buena aproximación.

- Si el espacio de hipótesis, es demasiado gigantesco, podemos llegar a tener problemas encontrando la respuesta correcta.

Las dos situaciones anteriores nos llevan a los dos problemas antes mencionados:

Ajuste pobre Cuando un modelo, no logra reducir suficientemente el error sobre el conjunto de entrenamiento (menos aún sobre el de validación) ver Figura 7.7.

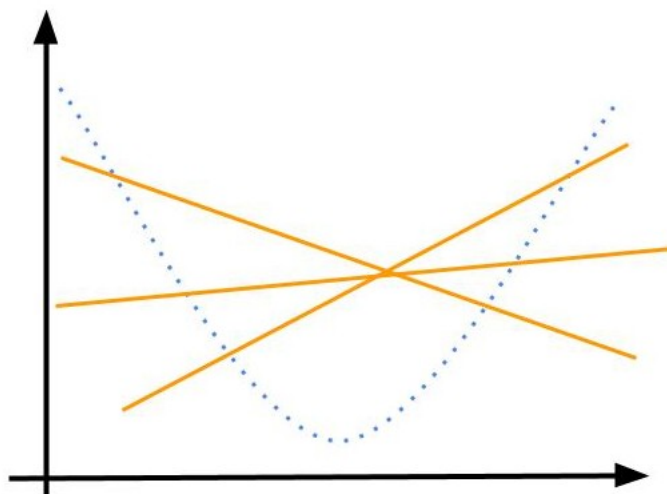


Figura 7.7 Ejemplo de ajuste pobre. La línea punteada es la función objetivo, las líneas naranjas el espacio de hipótesis.

Sobre ajuste Cuando un modelo, reduce muy bien el error con el conjunto de entrenamiento, pero con el conjunto de validación el error es muy alto, es decir, la hipótesis no generaliza a datos no vistos previamente, ver Figura 7.8.

Ejemplo 7.2. *Los datos del entrenamiento asemejan la función de un polinomio, entonces la red aprende solo a identificar los datos que pasen por la función, al momento de agregar datos nuevos estos probablemente no se comporten conforme al polinomio y nunca los clasifique adecuadamente.*

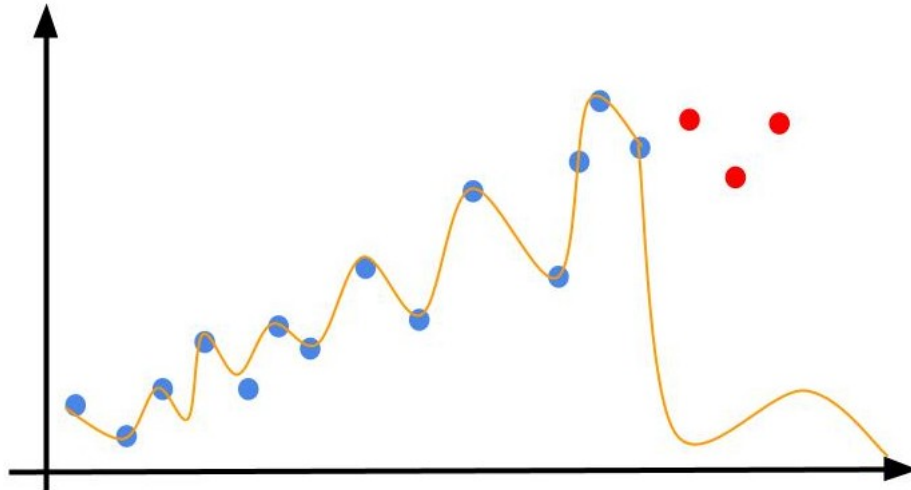


Figura 7.8 Ejemplo de sobre ajuste. La función aproximada de color naranja. Los datos durante el entrenamiento de color azul. Los datos nuevos de color rojo.

Estos dos problemas pueden suceder por dos razones distintas:

1. La cantidad de neuronas, dos casos:
 - **Faltan neuronas:** El espacio de hipótesis es muy simple para poder aproximar la función, es decir, tenemos un *ajuste pobre*.
 - **Sobran neuronas:** El espacio de hipótesis es muy expresivo. No obtiene las características de la función objetivo, sino que memorizó las etiquetas de los datos de entrada, es decir, tenemos un *sobreajuste*.
2. Faltan datos de entrenamiento, entonces independientemente del número de neuronas, el error en los datos de validación no baja, o incluso ni en los datos del entrenamiento.

Cuando nos sobran neuronas lo podemos detectar, porque al reducir neuronas, el desempeño de la red entrenada en el conjunto de entrenamiento, no empeora significativamente, y el desempeño con los datos de validación mejora.

Para automatizar la búsqueda en un espacio de hipótesis lo suficientemente expresiva vamos a usar la regularización.

Lo que vamos a hacer es modificar la función de pérdida, la nueva función va a estar integrada por dos componentes; la función de error (que ya teníamos), y un término de penalización sobre las magnitudes de los pesos de la red. Considerando estas dos anteriores, la función que estamos tratando de minimizar es la suma de esas dos.

El mínimo de esta función hallará un punto de compromiso tal que:

- reduce el error lo más posible
- solo crecen aquellos pesos que contribuyen mejor a reducir el error, compensando por la contribución de su magnitud creciente.

Es decir ahora tenemos una red en la cual colocamos desde el inicio muchas neuronas, lo cual tendería a producir un sobre ajuste. Lo que vamos a hacer es, penalizar los pesos de manera que el algoritmo de optimización haga que solamente aquellas conexiones que realmente están contribuyendo a reducir el error, tengan magnitudes significativas. Aquellas conexiones que no contribuyan entenderán a tener magnitudes muy pequeñas, casi como si no estuvieran ahí. El algoritmo de optimización hara que algunas de las conexiones queden muy débiles y básicamente ya no contribuyan. Entonces la hipótesis que se va a aprender va a ser más sencilla.

Dependiendo de qué tan rigurosos seamos con la penalización podemos hacer que, las únicas conexiones que pueden subir su magnitud en realidad sean muy pocas. También podría darse el problema de ajuste pobre, se tiene que ajustar bien el termino de penalización que lo vamos a denotar como λ .

La función de perdida se va actualizar de la siguiente forma:

$$J(\Theta) = \text{Error}(\Theta) + \frac{\lambda}{2m} \sum_{\Theta} \theta^2 \quad (7.11)$$

$$\nabla_{\theta} J^{(1)} = \nabla \text{Error}_{\Theta}^{(1)} + \frac{\lambda}{m} \theta \quad (7.12)$$

Entonces vamos a tener una gráfica parecida a la siguiente:

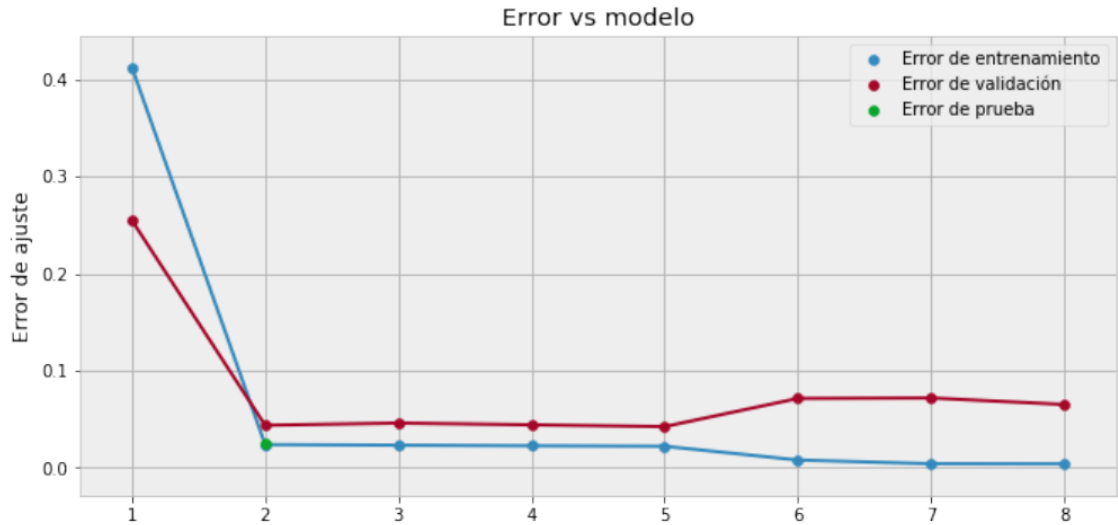


Figura 7.9 Sobreajuste.

En la ecuación 7.11 que del lado izquierdo tenemos nuestra función de error (podría ser una entropía cruzada, diferencias cuadrado), el segundo término es la penalización, la más común es sumar los todos los cuadrados de las magnitudes, de todos los pesos que tienen la red, este término va a provocar que la función suba de valor, si estamos utilizando pesos con magnitudes grandes.

La manera de minimizar los valores, es dejar subir solamente los valores de aquellos pesos que ayuden a compensar, bajando la penalización. La penalización está regida por el parámetro lambda, entre más grande sea lambda, se observa en la Figura 7.9 que se incrementa el valor de esta función, por culpa de los pesos. Entonces, si la lambda es muy pequeña el término dominante va a ser la función original de error y poca penalización por tanto muy poco efecto, por el hecho de haberle subido la magnitud de los pesos, vamos a obtener un sobreajuste. Esto equivaldría a tener nuestra red neuronal con el montón de neuronas que le pusimos, no hay penalización y podríamos tender precisamente al problema del sobreajuste.

Entonces vamos al lado derecho de la gráfica Figura 7.9. El error en el entrenamiento se está penalizando tanto por los pesos que el algoritmo de optimización, minimizó los pesos y casi no corrigió el error, entonces tenemos un ajuste pobre. En ese caso nuestra red no está modelando bien nuestra función, tenemos un error bastante alto y la función de validación también está creciendo.

En la parte de en medio de la gráfica Figura 7.9 es donde se obtiene el equilibrio, entre la cantidad de neuronas adecuadas y la cantidad de pesos significativos, que nos va a minimizar o eliminar el problema de un pobre o un sobre ajuste.

Descarte

Otra estrategia para regularizar es la deserción o descarte de algunas neuronas (*dropout* en inglés). Durante el entrenamiento, una cierta cantidad de salidas de capas ocultas se ignoran *aleatoriamente* (se les da valor final de 0) con probabilidad $1 - p$, es decir algunas neuronas de las capas ocultas se activan con un probabilidad p .

En esta técnica no se modifica la función de error, si no se modifica propiamente la red, ver Figura 7.10. En cada época, durante el entrenamiento se puede volver a elegir las neuronas a descartar, o no. La capa alterará su conectividad y buscará caminos alternativos para transmitir la información en la siguiente capa. Como resultado, cada actualización de una capa durante el entrenamiento se realiza con una configuración diferente de la capa. Así se puede simular el entrenamiento de una gran cantidad de redes neuronales con diferentes arquitecturas en paralelo.

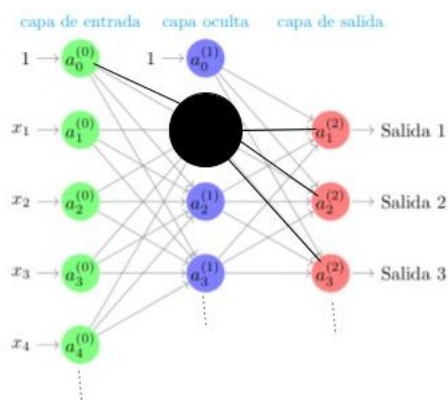


Figura 7.10 Dropout (desarrollado por Geoffrey Hinton y sus estudiantes en la Universidad de Toronto). Se descartan neuronas con sus conexiones aleatoriamente, dentro de la red.

Al momento de pasar los datos de prueba, todas las neuronas están activas. Pero se han reducido en p , esto sucede porque después de la eliminación en el entrenamiento, las siguientes capas recibirán valores más bajos. Sin embargo, en la fase de prueba, mantendremos todas las unidades, por lo que los valores serán mucho más altos de lo esperado, es por eso que necesitamos reducirlos, por un factor igual a la tasa de deserción p , para equilibrar el hecho de que hay más unidades activas que en tiempo de entrenamiento. Para información del tema puede leer el capítulo cuatro, sección 4.4.3 Adding dropout, Deep Learning With Python, 2017.

8 | Caso de análisis e interpretación

Red Hinton árbol familiar con numpy (entrenamiento)

En esta sección vamos a ver un ejemplo acerca de cómo utilizar a las redes neuronales, es un ejemplo particular pues sirve para calcular relaciones simbólicas, es un ejemplo propuesto por Geoffrey Hinton, en el artículo [Aprendiendo representaciones distribuidas de conceptos](#), donde el objetivo es, que una red neuronal aprenda el parentesco entre familiares. Para la creación de esta, se ayuda de árboles familiares como el que se ve en la Figura ??.

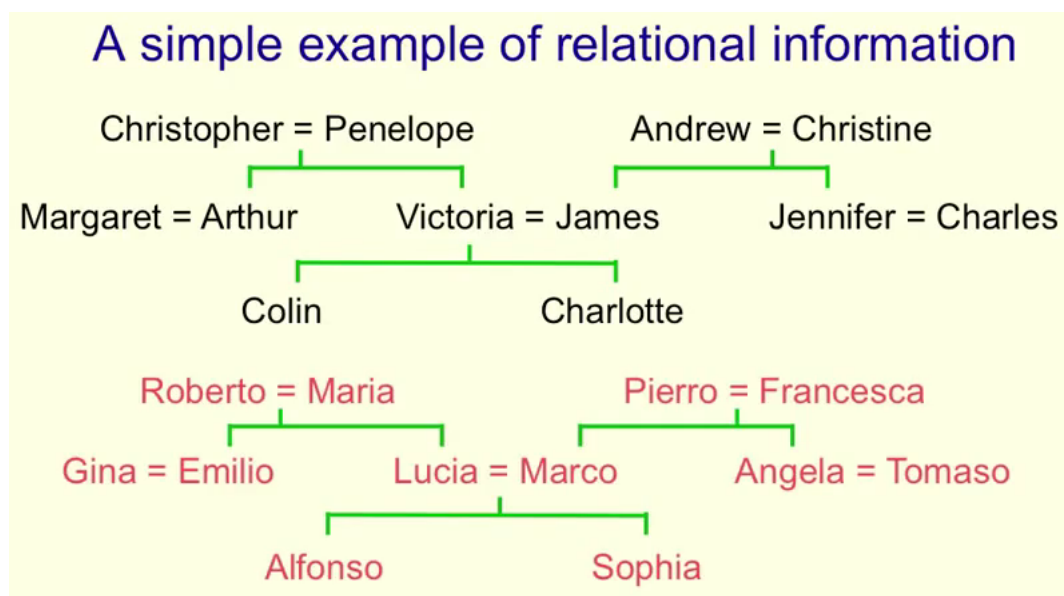


Figura 8.1 Ejemplo de dos árboles genealogicos de familias nucleares.

En el árbol anterior se pueden apreciar que existen las siguientes 12 relaciones entre personas; hijo, hija, sobrino, sobrina, padre, madre, tío, tía, hermano, hermana, esposo, esposa. Y vamos a tener la siguientes proposiciones para denotarlas se la siguiente forma;

- (Colin tiene-padre James)

- (Colin tiene-madre Victoria)
- (James tiene-esposa Victoria)

La tarea objetivo: *Que la red siguiente indique la segunda persona de la terna, dados los valores en las dos primeras posiciones.* Si tenemos los valores [nombre entrada] [tiene-relación] [nombre salida], con los dos primeros datos nos debe dar el tercer dato. Por ejemplo si nos preguntamos si Victoria tiene esposo, debería darme como respuesta James. Puede ser que la respuesta no sea única, por ejemplo si pregunto si Charlotte tiene tío hay dos respuestas posibles Arthur y Charles. No esperamos que la red haga nada lógico cuando no existe la relación. Solamente la probaremos con las relaciones que estén definidas en el árbol y por consiguiente en nuestro conjunto de entrenamiento.

Entonces lo primero que tenemos que hacer es ver cómo codificar a las personas y a las relaciones, retomando el concepto de one hot encoding, es lo que vamos a hacer. Al ver la Figura ?? vamos a tomar en la parte de abajo de la red neuronal que representa a las entradas, las codificaciones para la persona y el tipo de relación, lo que vamos a tener a la salida van a ser neuronas que representan a la segunda persona es la que está relacionada con la primera y es la que podemos la que podemos preguntar.

Lo que decidieron hacer en esta red, es no introducir sesgos inicio, pero hacer que la red por medio de su entrenamiento descubra una codificación compacta, donde se codifique juntos o con algún elemento en común, aquellos elementos que si tienen cosas en común, a esto se le llama hacer un auto codificador un encoder.

El diseño de la red por capa se propone de la siguiente forma:

- Capa uno:
 - ★ 24 neuronas de entrada (una para cada persona).
 - ★ 12 neuronas de entrada (una para cada relación).
- Capa dos:
 - ★ 6 neuronas conectadas con las 24 personas.
 - ★ 6 neuronas conectadas con las 12 relaciones.
- Capa tres:
 - ★ 12 neuronas conectadas a todas las neuronas en la capa 2.
- Capa cuatro:
 - ★ 6 neuronas conectadas a todas las neuronas en la capa 3.
- Capa cinco:

- ★ 24 neuronas de salida, una para cada persona relacionada con la de entrada.

Las datos de entrada son: 112 proposiciones, 100 utilizadas para entrenamiento, con 1500 iteraciones.

La estructura de esta red la Figura 8.2 en observen que no es una capa completamente conectada con la siguiente sino que ambos tipos de datos dan origen a dos regiones distintas, primero tenemos a la persona script en one-hot encoding por otro lado tenemos a la red, sin embargo vamos a tratar de y representar mejor esos datos y tratar de capturar si tienen algo en común entre ellas y eso lo vamos a lograr haciendo que la red misma descubra una forma compacta de representar a esas personas.

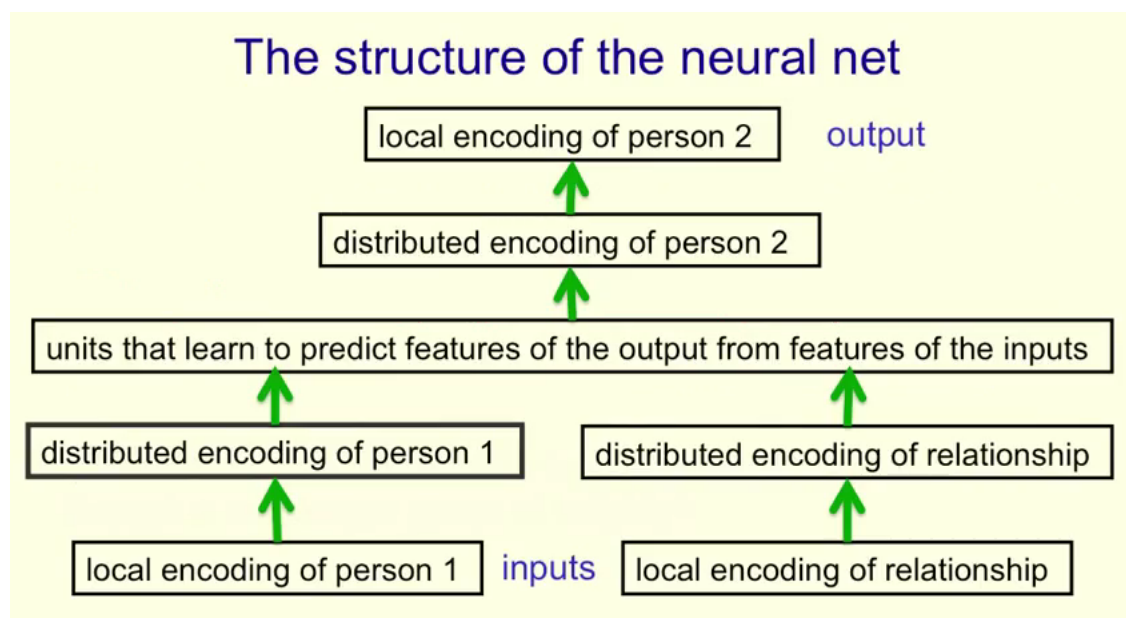


Figura 8.2 Estructura propuesta para la red neuronal de parentesco familiar.

Empezamos con 24 neuronas de entrada, después vamos a poner una mini capa, que solamente afecta, a estas neuronas y va a tener 6 neuronas eso quiere decir que la red neuronal tendrá que descubrir alguna manera de representar a las 24 personas con solamente 6 bits. Esta codificación le ayuda a predecir correctamente la persona que estamos buscando. Lo mismo vamos a hacer con el tipo de relaciones, había 12 en las neuronas de entrada, le vamos a reducir a 6 neuronas. Una vez que la red ha pasado por este cuello de botella, ha codificado así nuestros datos simbólicos, vamos a tener es una capa donde, ahora si conectamos todos contra todos es como si estas dos capas de pronto se fueran a pegar y se volvieran una. Estos van a estar conectados con todos estos esto es realmente lo que correspondería a nuestra capa de siempre nuestra capa oculta. Después vamos a tener que hacer el proceso inverso, en principio esta red de capa nos va a permitir encontrar qué relación tienen dos personas, pero si queremos poder predecir la otra relación entonces necesitamos partir de lo que encontremos.

Se va a reconstruir las características de la segunda persona, (se va a hacer el proceso inverso) vamos a tener también un codificador pero trataremos de predecir a la persona correcta con solamente seis neuronas, es decir, la segunda persona de la relación va a estar codificada en términos de los vínculos que tenga con las otras personas que existen en el conjunto. Una vez que ya logramos obtener esa codificación, tenemos que decodificarla y representarla nuevamente con 24 neuronas en la capa de salida, una por cada persona. Si logramos lo anterior nuestro código compacto de la penultima capa se puede descomprimir sin pérdida en la respuesta.

Las seis neuronas ocultas en el cuello de botella conectado a la entrada representa las características de las personas que son útiles para predecir la salida, tales como, la rama del árbol genealógico al que pertenece. La capa central aprende cómo las características predicen otras características. Por ejemplo, la persona de entrada es de generación 3 y la relación requiere que la respuesta sea una generación más, entonces implica que la persona de salida es de la generación 2.

Codificación de los datos de entrada

Lo primero que tenemos que hacer es, conseguir todos los datos para alimentar a nuestra red y codificarlos como acabamos de ver. Para eso tenemos un archivo escrito a mano (ver Figura 8.3) donde se hicieron tuplas con todas las relaciones posibles y se describe quién está relacionado con quién, al menos en una de las direcciones. En las relaciones que son simétricas se reutiliza la información para que sea menos tedioso de leer, entonces si ya tenemos las relaciones de X tiene padre Y, también tenemos las relaciones de Y tiene hijo X.

Una vez que tenemos todas estas duplas las podemos cargar dentro de nuestro archivo donde hagamos la red. Es en ese archivo donde verificaremos que tenemos las 111 combinaciones, pero solamente 104 son entradas distintas, (recordemos el caso confuso, de los dos tíos).

Ya te que nos aseguramos que este todo bien relacionado podemos hacer las entradas que recibirá la red, que son solo los dos primeros datos de la tupla, ([nombre],[tiene-relacion]).

A continuación tenemos que revisar las salidas, que serían todas las personas posibles, y todas las relaciones. Esto con one-hot encoding, se codifica a las personas posibles en un solo vector y a cada una se le asigna un lugar en este, así si James fue nuestra tercera persona en codificarse este ocupará el tercer lugar, y el vector se verá así $[0, 0, 1, 0, \dots, 0]$. Hacemos lo mismo para las relaciones, son solo 12 relaciones así que si tenemos la relación tiene-Tío, después de tiene-Padre y este fue el primero en codificarse, el vector para tiene-Tío se verá así $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.

La primera persona va a corresponder con la primera neurona, la segunda persona con la segunda neurona, la tercera con la siguiente neurona y así sucesivamente.

```

# Padres
padres = np.array([
    ('Arthur', 'tiene-padre', 'Christopher'),
    ('Victoria', 'tiene-padre', 'Christopher'),
    ('James', 'tiene-padre', 'Andrew'),
    ('Jennifer', 'tiene-padre', 'Andrew'),
    ('Colin', 'tiene-padre', 'James'),
    ('Charlotte', 'tiene-padre', 'James'),
    ('Emilio', 'tiene-padre', 'Roberto'),
    ('Gina', 'tiene-padre', 'Roberto'),
    ('Marco', 'tiene-padre', 'Pierro'),
    ('Angela', 'tiene-padre', 'Pierro'),
    ('Alfonso', 'tiene-padre', 'Marco'),
    ('Sophia', 'tiene-padre', 'Marco')
])

# Madres
madres = np.c_[padres[:,0], np.repeat('tiene-madre', len(padres)),
               padres[:,2]]
for i in range(len(padres)):
    madres[i][2] = esposas[esposas[:,0] == madres[i][2]][0,2]

# Hijos
temp = padres[:,2]
hijos = np.c_[temp[:,2], np.repeat('tiene-hijo', len(temp)), temp[:,
0]]
temp = madres[:,2]
hijos = np.vstack((hijos, np.c_[temp[:,2], np.repeat('tiene-hijo',
len(temp)), temp[:,0]]))

```

Figura 8.3 Fragmento del archivo, con los datos utilizados para el ejercicio.

Vamos a dividir las muestras en dos conjuntos, el conjunto de entrenamiento para juntar los pesos y un conjunto de prueba para averiguar si es capaz de hacer predicción. Para crear estos conjuntos vamos a utilizar una función que revuelva aleatoriamente, todas las relaciones posibles, utilizaremos 100 relaciones al entrenamiento y 4 para el de prueba. Como la arquitectura de la red ya está fija y lo único que vamos a hacer es tratar de ajustar los pesos.

Un tip, siempre la primera vez que se programe una red, se fija la semilla (para los muestreos aleatorios), lo que esto logrará hacer es que siempre que corre en el programa va a salir exactamente el mismo resultado. Al programar redes neuronales en varios pasos, como vamos a utilizar muestreos aleatorios, se piden números al azar. Si prueban en valores y funciona muy bien la próxima vez que ejecuten ese código no van a volver a obtener el mismo resultado y eso puede llegar a ser frustrante o puede complicar el proceso de depuración e incluso el proceso de recuperar una red que estaba funcionando bien.

No hay segos para esta red, porque en el ejercicio no tendríamos porque predisponernos a suponer algo y fue así como lo plantearon, funcionó bien.

Alimentación hacia adelante

Vamos a ver ahora cómo se lleva a cabo la alimentación hacia adelante, recordemos que tenemos que calcular dos cosas, la combinación lineal de los valores de activación de

la capa anterior 8.1 y sobre ella aplicarle la función de activación 8.2.

$$a_j^{(l+1)} = g \left(\sum_j \Theta_{i,j}^{(l)} a_j^{(l)} \right) \quad (8.1)$$

$$g = \frac{1}{1 + e^{-x}} \quad (8.2)$$

Esto se puede hacer en manera matricial multiplicando en la matriz de pesos por los valores de activación, donde $a_j^{(l)}$ es el valor de activación de la neurona j en la capa l y $\Theta_{i,j}^{(l)}$ es el peso del arista que conecta a la neurona $a_j^{(l)}$ con la neurona $a_i^{(l+1)}$.

$$\begin{bmatrix} a_0^{(l+1)} \\ a_1^{(l+1)} \\ \dots \\ a_n^{(l+1)} \end{bmatrix} = g \left(\begin{bmatrix} \theta_{10}^{(l)} & \dots & \theta_{1n}^{(l)} \\ \dots & & \dots \\ \theta_{n0}^{(l)} & \dots & \theta_{nn}^{(l)} \end{bmatrix} \begin{bmatrix} a_0^{(l+1)} \\ a_1^{(l+1)} \\ \dots \\ a_n^{(l+1)} \end{bmatrix} \right)$$

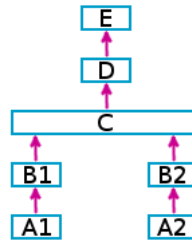


Figura 8.4 Arquitectura de la red.

El número de conexiones que se sugieren son las siguientes, teniendo en cuenta la arquitectura de la red que la podemos ver en la Figura 8.4:

- $WA1 \rightarrow (6 * 24) = 144$
- $WA2 \rightarrow (6 * 12) = 72$
- $WB1 \rightarrow (12 * 6) = 72$
- $WB2 \rightarrow (12 * 6) = 72$
- $WC \rightarrow (6 * 12) = 72$
- $WD \rightarrow (24 * 6) = 144$

En la primera capa tenemos dos bloques de neuronas, el A1 para meter los datos de las 24 personas y el A2 para meter las 12 relaciones posibles, estos van a tener de salida 6 neuronas cada bloque para los encoders. Las capas B1 y B2 son las encargadas de comprimir la información, los encoders y estos hacen una conexión completa con la capa C, con 72 pesos por cada capa. La capa C es la encargada de ver las características, como la información que tiene esta comprimida en 6 bits, ahora va a tener una salida de 12 bits para poder decodificar los datos, es decir tiene $6 \times 12 = 144$ pesos. La capa D es la encargada de decodificar los 6 bits que salen del bloque C en 24 bits que es lo que necesitamos, uno para cada persona, entonces el bloque D tiene $24 \times 6 = 144$ pesos.

Así tenemos un total de 576 pesos, cuyo valor debe ser ajustado.

Entonces en el algoritmo de propagación hacia delante, lo que tenemos es el producto de los pesos por los valores de activación y después la función logística, para calcular los valores de activación de la siguiente capa. vemos que primero se realice esta operación nada más para el auto encoder del lado izquierdo. Después hacemos lo mismo pero para las entradas y pesos que conectan la capa A2 con B2.

Una vez que obtuvimos ambos resultados unimos ambos elementos para que así se conecten todos contra todos en la capa C y hacemos de nuevo el producto de los pesos por los valores de activación y después la función logística, se procede igual con D y E con sus respectivos valores.

Vamos a guardar todas las variables de salida para usarla en la retropropagación. De este modo tendremos acceso a los valores de activación en las combinaciones lineales después de haber ejecutado.

Para la red, vamos a tener los métodos indispensables, un constructor donde podríamos asignar de pesos que ya hubiéramos sacado de algún otro lado. Necesitamos un método que nos permita colocar todas las matrices de pesos en un solo vector entonces lo que hace es aprender todas estas matrices de pesos y ponerles una encima de la otra esta es una forma de tener los parámetros del algoritmo de entrenamiento como un vector.

Después necesitamos una función auxiliar que lo que se encarga es de reconstruir todas estas matrices a partir de un vector, estos son métodos interesantes para poder probar qué es lo que estamos haciendo, por ejemplo hacer algo que jamás se debe de hacer podemos inicializar nuestra red poniendo solamente unos en las matrices de pesos. Esto jamás se debe de hacer cuando intenten entrenar una red porque no va a aprender absolutamente nada, si todos los pesos son iguales, todas las componentes del cliente son iguales y todo el tiempo va a empeorar o va a mejorar para algunos ejemplares pero se va a descomponer para otros. entonces este método nos sirve para ver que cuando programamos el fit forward hayamos hecho algo que tenga sentido.

8. Caso de análisis e interpretación

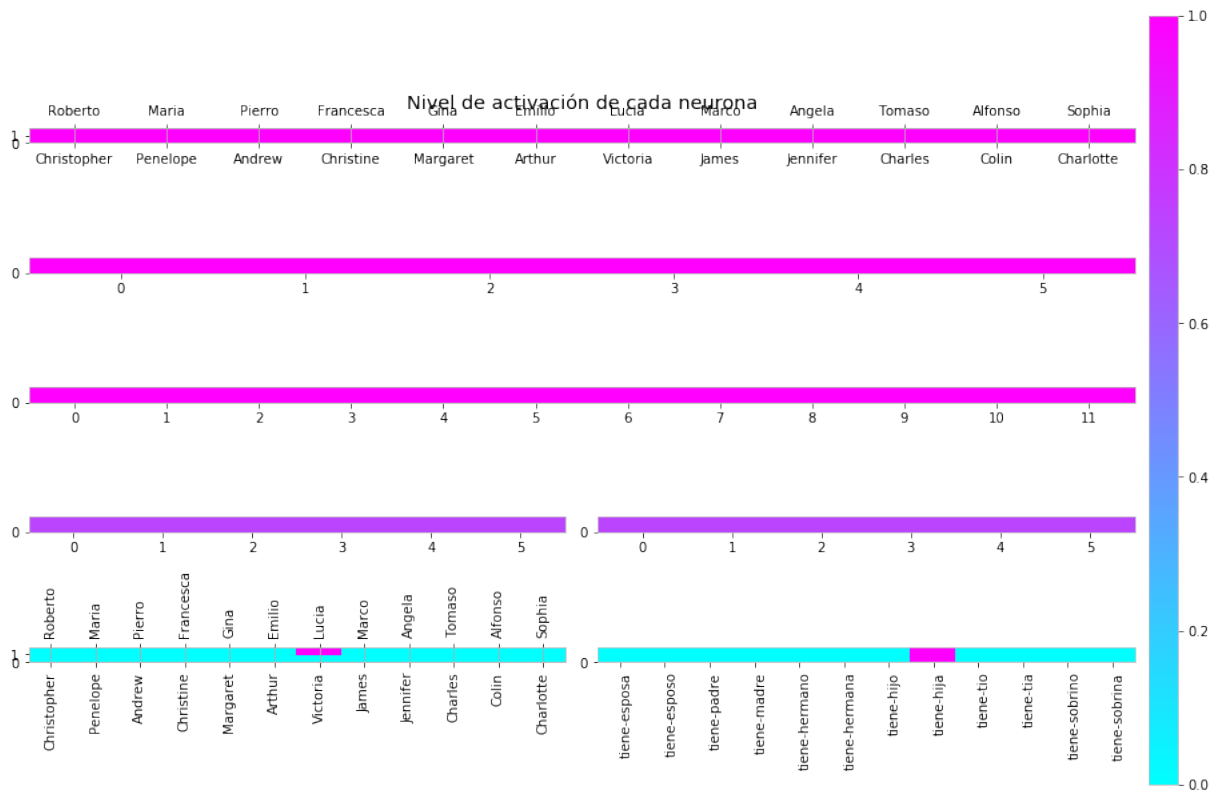


Figura 8.5 Red inicializada con todos pesos en un mismo valor. Si todos los pesos de la red son idénticos no tendremos una actualización de pesos a través de las capas, provocando que nunca llegue a dar respuestas. Esta red nunca aprendería.

Entonces la siguiente fase es cuando nos mandan un conjunto de datos y queremos evaluarlos a través de la red de la función de la red y obtener los valores de salida. Como de estos datos de entrenamiento que separan el conjunto en personas y relaciones, se manda a llamar el método feedforward, con los datos del conjunto de prueba y finalmente una vez que fueron programadas estas funciones vamos a probar si realmente está funcionando el primer paso pues es crear una instancia del objeto, asignarle los unos que podemos para decir básicamente como puede verse y tratar de aplicar el feed forward.

Necesitamos visualizar qué es lo que pasó y esto es lo que hacemos con una función auxiliar que permite graficar lo que está sucediendo, ese está en el archivo plot.py del zip de Hinton del curso. Así que vamos a poder ver lo que estamos haciendo en la parte de abajo de esta gráfica Figura 8.6 podemos ver a las personas podemos ver las relaciones y los colores azules indican ceros. Entonces en un principio cuando pusimos los pesos con el mismo valor, dadas las entradas que tenía que decir que tenía alta probabilidad de tener relación con todos o ninguno, y cuando la red ya está entrenada nos debe indicar solo una celda en color azul que es a la que pertenece.



Figura 8.6 Red con feedforward diferentes pesos, pero sin entrenar.

Entrenamiento

Momentum y decaimiento de los pesos

Ejecución del entrenamiento

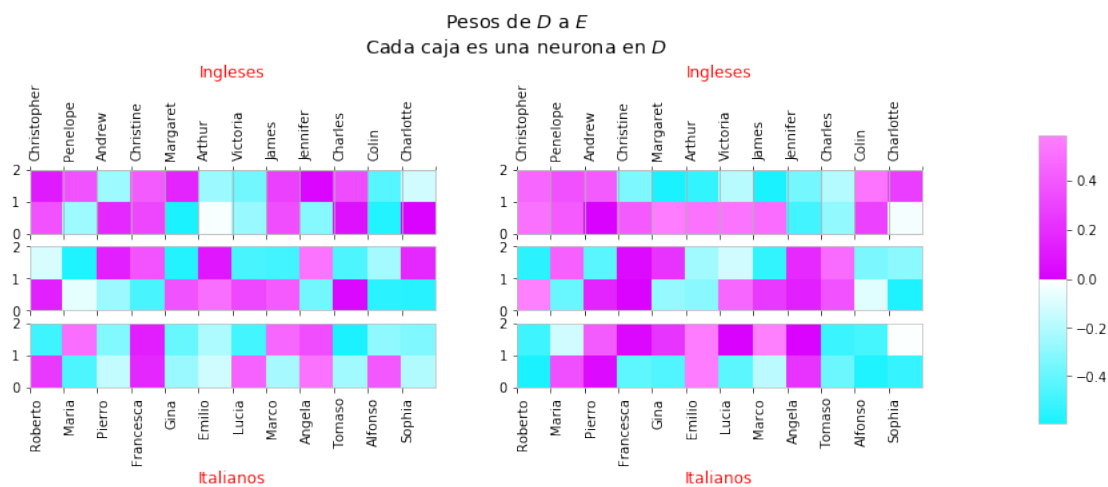


Figura 8.7 Resoluciones.

8. Caso de análisis e interpretación

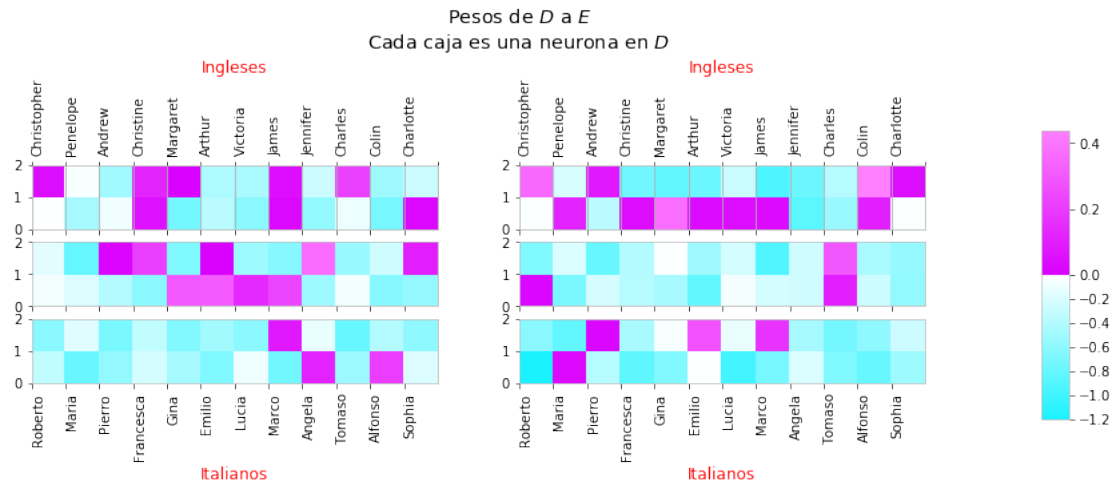


Figura 8.8 Resoluciones.

Red Hinton árbol familiar con pytorch

Vacio.

9 | Entrenamiento con genéticos

Algoritmos genéticos

Un algoritmo genético, consiste en “evolucionar” una población de individuos, cada uno de los cuales representa una posible solución. Esa población se somete a la evolución biológica es decir mutaciones y recombinaciones genéticas. De generación en generación, los individuos se seleccionan de acuerdo con una función de adaptación o *fitness*, en función de la cual se decide qué individuos sobreviven los más aptos y son descartados los menos aptos.

La selección se realiza siempre de forma probabilística. Un individuo es más probable que se seleccione si tiene un mejor valor de fitness, aunque cualquier individuo, por malo que sea, tiene una probabilidad de selección estrictamente mayor que cero. En ocasiones, si no queremos que una muy buena solución encontrada en una generación intermedia de la evolución se pierda por el camino, podemos introducir cierto grado de elitismo en la selección. Podemos hacer que el mejor individuo de la población (o los k mejores) siempre sobrevivan, algo que aleja a los algoritmos genéticos del mundo real, donde una generación termina siempre reemplazando por completo a las anteriores.

Neuroevolución

Vacio.

Antecedentes: Aprendizaje por refuerzo en videojuegos

Vacio.

Arquitectura para estimar la función de recompensa

Vacio.

Entrenamiento

Vacio.

10 | Mapeos autoorganizados

Introducción

Vacio.

Aprendizaje no supervisado

Vacio.

Mapeos autoo-organizados

Vacio.

Kohonen

Vacio.

11 | Redes Neuronales Convolucionales

Convolución

Redes Convolucionales

Softmax

MNIST

PARTE III

Redes con ciclos

12 | Redes Neuronales Recurrentes

Derivadas ordenadas

Retropropagación en el tiempo

Sistemas dinámicos y despliegue del grafo

Arquitectura recurrente universal

Función de error

Forzamiento del profesor

13 | Atención

14 | LSTM

15 | GRU

16 | Casos de análisis: etiquetado de palabras y conjugación de verbos

PARTE IV

Redes no dirigidas

17 | Redes de hopfield

Entrenamiento

18 | Máquinas de Boltzman

Entrenamiento

Partículas y partículas de fantasía

Máquinas de Boltzman Restringidas

19 | Redes adversarias

GANs

A | Ecuaciones diferenciales