Lingüística Computacional

Víctor Mijangos de la Cruz

IV. Modelos del lenguaje

Modelos del lenguaje

¿Cómo predecir lenguaje?

Las gramáticas formales permiten:

- Saber si una cadena pertenece al lenguaje (y parsearla).
- Generar las cadenas pertenecientes al lenguaje.

Sin embargo, las gramáticas formales tienen el problema de ser demasiado **estrictas**.

Por su parte, el lenguaje natural tiene la propiedad de ser **permisivo**.

Probabilidades de cadenas

Se propone introducir el cálculo de probabilidades a los métodos de NLP.

En particular, se conceptualizan dos problemas dentro del NLP (Jurafski, 2000):

- Determinar la probabilidad de una cadena $w^{(1)}, w^{(2)}, ..., w^{(T)} \in \Sigma^*$; esto es:

$$p(w^{(1)}, w^{(2)}, ..., w^{(T)})$$

- Dada una cadena $w^{(1)}, w^{(2)}, ..., w^{(t-1)} \in \Sigma^*$ determinar la probabilidad de la palabra siguiente; esto es:

$$p(w^{(t)} | w^{(t-1)}, ..., w^{(1)})$$

Probabilidades de cadenas

Estos problemas son importantes, puesto que:

- En las aplicaciones de NLP queremos generar cadenas que tengan alta probabilidad.
- Podemos determinar que tan probable es que una cadena pertenezca a un lenguaje.
- Nos permite predecir secuencias de cadenas.
- Esto también nos permite generar cadenas con probabilidades altas de pertenecer al lenguaje.

Inicio y final de la cadena

En los modelos estadísticos de este tipo, se introducen dos símbolos de primordial importancia:

- BOS (Beggining Of String): indica el inicio de una cadena.
- EOS (End Of String): inidca el final de una cadena.

Se puede iniciar una cadena arbitraria buscando:

$$\arg\max_{w} p(w|BOS)$$

Esta cadena se generará hasta que se encuentre el símbolo de EOS.

Probabilidad de cadenas

La probabilidad de una cadena $p(w^{(1)}, w^{(2)}, ..., w^{(T)})$ y las probabilidades condicionales $p(w^{(t)}|w^{(t-1)}, ..., w^{(1)})$ se relacionan de la siguiente forma:

$$p(w^{(1)}, w^{(2)}, ..., w^{(T)}) = \prod_{t=1}^{T} p(w^{(t)}|w^{(t-1)}, ..., w^{(1)}) \quad (1)$$

Esto reduce ambos problemas a un único problema: encontrar las probabilidades condicionales de las cadenas.

Podemos plantear el problema como sigue:

> *Dado el conjunto de cadenas Σ^* sobre un vocabulario (alfabeto) Σ, un modelo del lenguaje buscará asignar una probabilidad a cada una de las cadenas en Σ^*.*

Transductores probabilísticos

Muchos de los problemas del PLN se pueden ver como estimar este tipo de probabilidades.

En muchos casos, lo que se busca es pasar de un lenguaje L_1 a otro lenguaje L_2 (etiquetado, voz a texto, reconocimiento de caracteres, traducción automática). En ese caso, se puede modelar el problema como:

$$p(w^{(1)}, ..., w^{(T)}, s^{(1)}, ..., s^{(T')}) = \prod_{t=1}^{T} p(s^{(1)}, ..., s^{(T')} | w^{(1)} ..., w^{(T)}) p(w^{(1)}, ..., w^{(T)}) \quad (2)$$

Tal que $w^{(1)}, ..., w^{(T)} \in L_1$ y $s^{(1)}, ..., s^{(T')} \in L_2$. La probabilidad $p(w^{(1)}, ..., w^{(T)})$ define un modelo del lenguaje.

Modelo del lenguaje

Sin embargo, Σ^* tiene cardinalidad infinita. El objetivo es, a partir de un conjunto de datos empíricos, estimar una medida de probabilidad P sobre las cadenas formadas de Σ.

El modelo del lenguaje, puede entonces definirse como:

Modelo del lenguaje

Un modelo de lenguaje es una tupla $\mu = (\Sigma, P)$, donde Σ es el vocabulario (o alfabeto) y P es una medida de probabilidad sobre Σ^*.

El tipo de modelo del lenguaje dependerá de que tipo de estimador se use para P:

- Modelos del lenguaje de n-gramas (proceso estocástico, procesos de Markov)
- Modelos del lenguaje neuronales (redes neuronales)
  - Modelos del lenguaje causales (redes de una dirección)
  - Modelos del lenguaje bidireccionales (redes bidireccionales)

Modelos del lenguaje de *n*-gramas

Modelo de *n*-gramas

Una forma tradicional de determinar las probabilidades se basa en la teoría de **procesos estocásticos**, en particular en los **procesos de Markov**.

A esta forma de determinar el modelo del lenguaje se le conoce como **Modelo del Lneguaje de n-gramas**.

En general, se definen un número finito de probabilidades condicionales, a partir de los cuales se pueden calcular la probabilidad de cualquier elemento de Σ^*.

Procesos estocástico

Un proceso de Markov es un tipo de **proceso estocástico**. Un proceso estocástico es un conjunto de variables aleatorias que representan los estados de un sistema en un tiempo t:

$$\{X_t : t \geq 0\}$$

Los modelos del lenguaje pueden caracterizarse como procesos estocásticos:

$$X_1 = w^{(1)}, X_2 = w^{(2)}, ..., X_T = w^{(T)}$$

Procesos de Markov

Propiedad de Markov

Sea $X_0, X_1 ...$ un proceso estocástico con una función de probabilidad p. Se dice que el proceso estocástico tiene la propiedad de Markov si cumple:

$$p(X_t = x^{(t)} | X_{t-1} = x^{(t-1)}, ..., X_0 = x^{(0)}) = p(X_t = x^{(t)} | X_{t-1} = x^{(t-1)}) \quad (3)$$

Los **procesos de Markov** son procesos estocásticos que cumplen la propiedad de Markov; es decir, cada estado del proceso depende únicamente del estado inmediatamente anterior (Rincón, 2007).

Probabilidades conjuntas

A partir de la propiedad de Markov, la probabilidad de un proceso de Markov, puede determinarse como:

$$p(x^{(0)}, ..., x^{(T)}) = p(x^{(0)}) \prod_{i=1}^{T} p(x^{(t)}|x^{(t-1)}, ..., x^{(0)}) \quad (4)$$

$$= p(x^{(0)}) \prod_{t=1}^{T} p(x^{(t)}|x^{(t-1)}) \quad (5)$$

Las probabilidades $p(x^{(t)}|x^{(t-1)})$ se conocen como **probabilidades de transición** y $p(x^{(0)})$ como probabilidad **inicial**.

Homogeneidad

En general, las probabilidades de transición depende del estado en que se encuentren. Así transitar dele estado 3 al 4 tendrá una probabilidad distinta que transitar del 10 al 11.

Para simplificar el cálculo de las probabilidades de transición se asume que los procesos de Markov que describiremos son **homogeneos**; esto es, que cumplen:

$$p(X_t = x_j | X_{t-1} = x_i) = p(X_1 = x_j | X_0 = x_i)$$

Esto implica que la probabilidad de transición entre dos elementos es la misma sin importar el estado en que se está.

Matriz de transiciones

Asumiendo que tenemos procesos de Markov homogéneos, denotamos $p(x_j^{(t)}|x_i^{(t-1)}) = p(x_j|x_i)$, para cualquier t.

Si el proceso puede tomar N valores, entonces, las probabilidades que debemos obtener son N^2. Por tanto, las probabilidades se expresan en una **matriz de transición** de $N \times N$:

$$A = (p_{i,j}) = p(x_j|x_i)$$

	x_1	x_2	$\cdots$	x_N
x_1	$p_{1,1}$	$p_{2,1}$	$\cdots$	$p_{N,1}$
x_2	$p_{1,2}$	$p_{2,2}$	$\cdots$	$p_{N,2}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_N	$p_{1,N}$	$p_{2,N}$	$\cdots$	$p_{N,N}$

Probabilidades iniciales

Además de las transiciones, debemos saber qué tan probable es que un proceso inicie con un valor x_i. Ya que existen N valores, necesitamos N probabilidades. Éstas se guardan en un **vector de probabilidades iniciales**:

$$\Pi = (\pi_i) = p(x_i)$$

El proceso de Markov, se describe por A y Π, estos tienen las propiedades:

- Para todo $i, j = 1, ..., N$, $p_{i,j} \geq 0$,
- $\sum_{j=1}^{N} p_{i,j} = 1$
- $\sum_{i=1}^{N} \pi_i = 1$

Ejemplo: Proceso de Markov

Supóngase que se tiene un alfabeto binario $\Sigma = \{0, 1\}$. Determinar la probabilidad de todas las cadenas en Σ^*.

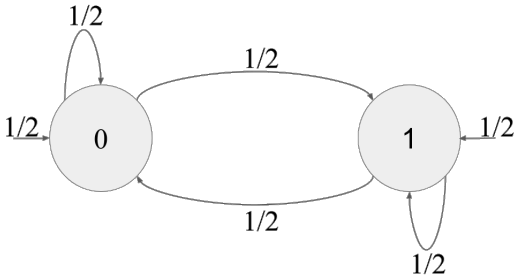
Asumiremos que las cadenas definen un proceso de Markov dado por la matriz de transición:

$$A = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \end{array}$$

Y el vector de probabilidades inciales:

$$\Pi = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \\ \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \end{array}$$

Ejemplo: Proceso de Markov



Gráfica del proceso de Markov definido por un alfabeto binario

La probabilidad de una cadena como 01110 está dada por:

$$p(01110) = p(0)p(1|0)p(1|1)p(1|1)p(0|1) = \frac{1}{32}$$

Procesos de Markov de orden n

Los procesos de Markov tienen una generalización:

Propiedad de Markov de orden n

Sea $X_1, ..., X_T$ una serie de variables aleatorias que pueden tomar valores $x_1, x_2, ..., x_N$ y con una función de probabilidad p. Se dice que las variables aleatorias tienen la propiedad de Markov de orden n si, para algún $n \in \mathbb{N}$, cumplen:

$$p(x^{(t)}|x^{(t-1)}, ..., x^{(1)}) = p(x^{(t)}|x^{(t-1)}, ..., x^{(t-(n-1))}) \tag{6}$$

.

De tal forma que la probabilidad de los procesos es:

$$p(x^{[1)}, x^{[2)}, ..., x^{[T)}) = \prod_{t=1}^{T} p(x^{(t)}|x^{(t-1)}, ..., x^{(t-n+1)})$$

Procesos de Markov de orden *n*

Los procesos de Markov tradicionales son de orden 2. Cuando hablamos de un orden mayor, debemos caracterizar las probabilidades de transición por medio de un *tensor de transiciones*:

$$A = (p_{i_1,i_2,\ldots,i_n}) = p(x_{i_n}|x_{i_1}, \ldots, x_{i_{n-1}})$$

Por ejemplo, para orden 3, tenemos que:

$$A = (p_{i,j,k}) = p(x_k|x_i, x_j)$$

Las probabilidades iniciales se determinarán como hemos señalado. El tensor A y el vector Π determinarán el proceso de Markov.

Lenguaje como procesos de Markov

Supóngase que se cuenta con un vocabulario Σ de palabras, de tal forma que las cadenas son de la forma $w^{(1)}w^{(2)}...w^{(T)}$. Debemos determinar:

$$p(w^{(1)}w^{(2)}...w^{(T)})$$

Podemos aproximar esta probabilidad, asumiendo la propiedad de **Markov de orden n**:

$$\begin{aligned} p(w^{(1)}w^{(2)}...w^{(T)}) &= \prod_{t=1}^{T} p(w^{(t)}|w^{(1)}...w^{(t-1)}) \\ &\approx \prod_{t=1}^{T} p(w^{(t)}|w^{(t-n+1)}...w^{(t-1)}) \end{aligned}$$

Aproximaciones estadísticas al lenguaje

Shannon (1948) propone niveles de aproximación a las estadísticas del lenguaje con base en los modelos estocásticos:

- **Aproximación de orden 0**: Dado $w \in \Sigma$, su probabilidad es $p(w) = \frac{1}{N}$.
- **Aproximación de orden 1**: Asume independencia de palabras, tal que $p(w^{(1)}, ..., w^{(T)}) = \prod_{i=1}^{T} p(w^{(i)})$.
- **Aproximación de orden 2**: Asume independencia de palabras, tal que $p(w^{(1)}, ..., w^{(T)}) = \prod_{i=1}^{T} p(w^{(i)}|w^{(i-1)})$.

Actualmente, estas aproximaciones se conocen como *n*-gramas, donde *n* es el número de elementos en la probabilidad.

Modelo del lenguaje de bigramas

A la aproximación de orden 2, se le conoce como **modelo de bigramas**. Este se define como:

Modelo del lenguaje de bigramas

Un modelo del lenguaje de bigramas es una tupla $\mu = (\Sigma, (A, \Pi))$, donde Σ es un vocabulario, $A = (a_{ij}) = p(w_j|w_i), w_i, w_j \in \Sigma$ es una matriz de probabilidades de transiciones y $\Pi = (\pi_i) = p(w_i), w_i \in \Sigma$ es un vector de probabilidades iniciales.

Y las probabilidades se determinan como:

$$p(w^{(1)}w^{(2)}...w^{(T)}) \approx \prod_{t=1}^{T} p(w^{(t)}|w^{(t-1)}) \tag{7}$$

Bigramas

Es común, obtener del corpus los pares de palabras (w_i, w_j) sobre los que se calcularán las probabilidades. Estos se conocen como **bigramas**:

Bi-grama

Sea $w^{(1)}w^{(2)}...w^{(T)}$ una cadena. Los bi-gramas sobre la cadena $w^{(1)}w^{(2)}...w^{(T)}$ son los pares $(w^{(t-1)}, w^{(t)})$ con $t = 1, ..., T$.

Las probabilidades de transición pueden calcularse por medio de la frecuencia de estos bigramas (si el bigrama no aparece en el corpus, se asume frecuencia 0):

$$p(w_j|w_i) = \frac{fr(w_i, w_j)}{fr(w_i)} \tag{8}$$

Estimación de modelo de bigramas

A partir de un **corpus de entrenamiento** hacer:

1. De las cadenas del corpus $w^{(1)}w^{(2)}...w^{(T)}$ obtener el vocabulario $\Sigma = \{w_1, w_2, ..., w_N\}$ de los tipos del corpus.
2. Añadir símbolos de inicio *BOS* y fin *EOS* a las cadenas $BOSw^{(1)}w^{(2)}...w^{(T)}EOS$
3. Obtener los bigramas:

$$(BOS, w^{(1)}), (w^{(1)}, w^{(2)}), ..., (w^{(T-2)}, w^{(T-1)}), (w^{(T-1)}, w^{(T)}), (w^{(T)}, EOS)$$

4. Obtener el vector de probabilidades iniciales como:

$$p(w_i) := p(w_i|BOS) = \frac{fr(BOS, w_i)}{fr(BOS)}$$

5. Obtener la matriz de transición (incluyendo EOS):

$$p(w_j|w_i) = \frac{fr(w_i, w_j)}{fr(w_i)}$$

Ejemplo: Modelo de bigramas

Considérese el siguiente corpus:

- El niño jugaba con el carrito.
- El niño jugaba.
- El niño salta.

El vocabulario obtenido de aquí es:

$$\Sigma = \{el, \text{niño}, jugaba, salta, con, carrito\}$$

Agregando los símbolos de inicio y final:

- BOS el niño jugaba con el carrito EOS
- BOS el niño jugaba EOS
- BOS el niño salta EOS

Ejemplo: Modelo de bigramas

De estas cadenas, obtenemos los bigramas por oración:

- (BOS, el), (el, niño), (niño, jugaba), (jugaba, con), (con, el), (el, carrito), (carrito, EOS)
- (BOS, el), (el, niño), (niño, jugaba), (jugaba, EOS)
- (BOS, el), (el, niño), (niño, salta), (salta, EOS)

Por ejemplo, la probabilidad inicial de "'el"' se calcula como:

$$\begin{aligned} p(el) &= p(el|BOS) \\ &= \frac{fr(BOS, el)}{fr(BOS)} \\ &= \frac{3}{3} = 1 \end{aligned}$$

Ejemplo: Modelo de bigramas

El vector de probabilidades iniciales Π queda como:

	BOS
el	1
niño	0
jugaba	0
salta	0
con	0
carrito	0

Ejemplo: Modelo de bigramas

Las probabilidades de transición, por ejemplo, para (el, niño) se calculan como:

$$p(\text{niño}|el) = \frac{fr(el, \text{niño})}{fr(el)} = \frac{3}{4} = 0.75$$

La matriz de transición A queda como (hemos considerado EOS en los renglones):

	el	niño	jugaba	salta	con	carrito
el	0	0	0	0	1	0
niño	0.75	0	0	0	0	0
jugaba	0	0.66	0	0	0	0
salta	0	0.33	0	0	0	0
con	0	0	0.5	0	0	0
carrito	0.25	0	0	0	0	0
EOS	0	0	0.5	1	0	1

Ejemplo: Modelo de bigramas

Nuestro modelo de bigramas queda determinado como $\mu = (\Sigma, (A, \Pi))$. A partir de este modelo podemos estimar las probabilidades de cualquier cadena. Por ejemplo, la probabilidad de la cadena "'el niño juega"' está dada por:

$$\begin{aligned} p(\text{el, niño, jugaba}) &= p(\text{el})p(\text{niño}|\text{el})p(\text{jugaba}|\text{niño}) \\ &= 1 \cdot \frac{3}{4} \cdot \frac{2}{3} \\ &= \frac{6}{12} \\ &= 0.5 \end{aligned}$$

Generación de cadenas

Dada una cadena, podemos predecir la probabilidad la palabra más probable como:

$$\hat{w} = \arg\max_i p(w_i^{(t)} | w^{(t-1)})$$

De igual forma, aplicando iterativamente este proceso podemos generar una cadena. Por ejemplo, con el modelo dado, tenemos que:

$$\arg\max_i p(w_i | BOS) = el$$

Ahora tomamos "'el"' como elemento de la cadena y obtenemos:

$$\arg\max_i p(w_i | el) = \text{niño}$$

Y aplicando una ves más:

$$\arg\max_i p(w_i | \text{niño}) = jugaba$$

Modelo de n-gramas

Los modelos de bigramas pueden generalizarse a partir de procesos de Markov de orden n. En primer lugar definimos **n-grama**:

n-grama

Dado una cadena $w^{(1)}w^{(2)}...w^{(T)}$, un n-grama es el conjunto de n elementos en la cadena

$$(w^{(t-n+1)}, w^{(t-n+2)}, ..., w^{(t-1)}, w^{(t)})$$

con $t = 1, ..., T$.

Un **modelo del lenguaje de n-gramas** es una tupla $\mu = (\Sigma, (A, \Pi))$, tal que Σ es el vocabulario $\Pi = (\pi_i) = p(w_i)$ el vector de probabilidades iniciales y A un tensor de transiciones tal que:

$$A = (p_{i_1,..,1_n}) = p(w_{i_n}|w_{i_1}...w_{i_{n-1}})$$

Modelo de n-gramas

En un modelo de *n*-gramas, las probabilidades de las cadenas son de la forma:

$$p(w^{(1)}...w^{(T)}) = \prod_{t=1}^{T} p(w^{(t)}|w^{(t-n+1)}...w^{(t-1)})$$

Donde las probabilidades condicionales se pueden obtener como:

$$\begin{aligned} p(w^{(t)}|w^{(t-n+1)}...w^{(t-1)}) &= \frac{p(w^{(t-n+1)}...w^{(t)})}{p(w^{(t-n+1)}...w^{(t-1)})} \\ &= \frac{fr(w^{(t-n+1)}, w^{(t-n+2)}, ..., w^{(t-1)}, w^{(t)})}{fr(w^{(t-n+1)}, w^{(t-n+2)}, ..., w^{(t-1)}, w^{(t-1)})} \end{aligned}$$

Modelo de trigramas

Un caso particular es el modelo de 3-gramas o modelo de **trigramas**. Aquí el tensor de transición está dado por:

$$A = (p_{i,j,k}) = p(w_k|w_i w_j)$$

La probabilidad de cadenas es:

$$p(w^{(1)}...w^{(T)}) = \prod_{t=1}^{T} p(w^{(t)}|w^{(t-2)}w^{(t-1)})$$

Y las probabilidades condicionales pueden calcularse como:

$$p(w_k|w_i w_j) = \frac{fr(w_i, w_j, w_k)}{fr(w_i, w_j)}$$

Ejemplo: Modelo de trigramas

Considérese el siguiente corpus:

- Juan quería los tacos.
- Juan comía un pastel.

Del que se obtiene el vocabulario:

$$\Sigma = \{juan, queria, comia, los, un, tacos, pastel\}$$

Añadiendo los símbolos de inicio y de fin tenemos:

- BOS juan quería los tacos EOS
- BOS juan comía un pastel EOS

Ejemplo: Modelo de trigramas

Del cual podemos obtener los bigramas, para la primera cadena estos son:

(BOS, juan, quería), (juan, quería, los), (quería, los, tacos), (los, tacos, EOS)
(BOS, juan, comía), (juan, comía, un) (comia, un, pastel), (un, pastel, EOS)

Por ejemplo, la probabilidad de "‘ tacos"’ dado "‘quería los"’ es:

$$p(tacos|queria, los) = \frac{fr(queria, los, tacos)}{fr(queria, los)} = \frac{1}{1} = 1$$

En general, debemos clacular $7^3 = 343$ probabilidades de transición que componen el tensor A.

Ejemplo: Modelo de trigramas

La probabilidad de una cadena como "'juan comía un pastel"' se determinará como:

$$\begin{aligned} p(\textit{juan}, \text{comía}, \textit{un}, \textit{pastel}) &= p(\textit{juan})p(\text{comía}|\textit{BOS}, \textit{juan})p(\textit{un}|\textit{juan}, \text{comía})p(\textit{pastel}|\text{comía}, \textit{un}) \\ &= 1 \cdot 0.5 \cdot 1 \cdot 1 \\ &= 0.5 \end{aligned}$$

Sin embargo, se puede observar que:

$$p(\textit{juan}, \text{comía}, \textit{los}, \textit{tacos}) = 0$$

Es decir, existe un **overfiting**, por lo que oraciones aceptables se les asigna probabilidades nulas.

Dispersión

Los ceros en las probabilidades de transición son problemáticos para las aplicaciones de PLN.

Cuando en una matriz (o un tensor) existe una buena cantidad de ceros, se dice que es **dispersa**.

El objetivo es obtener matrices poco dispersas, de tal forma que toda transición tenga asignada una probabilidad. La probabilidad de transiciones no observadas debe ser relativamente pequeña. Pero que, para toda cadena:

$$p(w^{(t)}|w^{(1)}...w^{(t-1)}) > 0$$

A la corrección de estas probabilidades se le llama **smoothing** (o suavizamiento).

Corrección de frecuencias

Ya que las frecuencias están probabilidades como:

$$p(w^{(t)}|w^{(1)}...w^{(t-1)}) = \frac{fr(w^{(1)}...w^{(t)})}{fr(w^{(1)}...w^{(t-1)})}$$

Se puede agregar un factor $\lambda > 0$ para que todo valor sea mayor a 0. Sin embargo, debe recordarse que:

$$\sum_{w^{(t)}} p(w^{(t)}|w^{(1)}...w^{(t-1)}) = 1$$

Smoothing de Lidstone

Obsérvese que al agregar λ al dividendo se obtiene:

$$\sum_{w^{(t)}} \frac{fr(w^{(1)}...w^{(t)}) + \lambda}{fr(w^{(1)}...w^{(t-1)})} = \frac{fr(w^{(1)}...w^{(t-1)}) + \lambda \sum_{w^{(t)}} 1}{fr(w^{(1)}...w^{(t-1)})} \tag{9}$$

$$= \frac{fr(w^{(1)}...w^{(t-1)}) + \lambda N}{fr(w^{(1)}...w^{(t-1)})} \tag{10}$$

Entonces, para que la suma sea 1, se debe añadir λN al divisor. Este es el smoothing de Lidstone:
Dado un modelo de lenguaje, la probabilidad con un **smoothing de Lidstone** se define:

$$p(w^{(t)}|w^{(1)}...w^{(t-1)}) := \frac{fr(w^{(1)}...w^{(t)}) + \lambda}{fr(w^{(1)}...w^{(t-1)}) + \lambda N} \tag{11}$$

Donde N es el número de tipos en el corpus.

Smoothing Laplaciano

Un caso particular del smoothing de Lidstone es el **smoothing Laplaciano** (o Add One) donde $\lambda = 1$, de tal forma que:

$$p(w^{(t)}|w^{(t-1)} = \frac{fr(w^{(t-1)}w^{(t)}) + 1}{fr(w^{(t-1)}) + N}$$

Este smoothing es usado con frecuencia debido a que no requiere determinar el valor de λ.

Otras formas de determinar el smoothing es estimar el parámetro λ que mejor adapte.

Ejemplo: Smoothing Laplaciano

Considérese que se tiene el corpus siguiente (con símbolos de inicio y fin ya agregados):

- BOS el niño jugaba con el carrito EOS
- BOS el niño jugaba EOS
- BOS el niño salta EOS

Se tiene el vocabulario:

$$\Sigma = \{el, \text{niño}, jugaba, salta, con, carrito\}$$

Crearemos un modelo del lenguaje de *bigramas* con smoothing Laplaciano.

Ejemplo: Smoothing Laplaciano

Seguiremos los siguientes pasos:

1. Llenar el vector de iniciales y las transiciones con frecuencias de bigramas.
2. Sumar un uno a cada entrada del vector y la matriz.
3. Sumar las columnas y dividir cada entrada por la suma de esa columna.

Para la matriz A tenemos:

	el	niño	jugaba	salta	con	carrito
el	0	0	0	0	1	0
niño	3	0	0	0	0	0
jugaba	0	2	0	0	0	0
salta	0	1	0	0	0	0
con	0	0	1	0	0	0
carrito	1	0	0	0	0	0
EOS	0	0	1	1	0	1

Ejemplo: Smoothing Laplaciano

Añadiendo un 1 a cada entrada, obtenemos entonces:

	el	niño	jugaba	salta	con	carrito
el	1	1	1	1	2	1
niño	4	1	1	1	1	1
jugaba	1	3	1	1	1	1
salta	1	2	1	1	1	1
con	1	1	2	1	1	1
carrito	2	1	1	1	1	1
EOS	1	1	2	2	1	2

La suma de las columnas es:

	el	niño	jugaba	salta	con	carrito
+	11	10	9	8	8	8

Ejemplo: Smoothing Laplaciano

Finalmente, las probabilidades resultantes son:

	el	niño	jugaba	salta	con	carrito
el	1/11	1/10	1/9	1/8	2/8	1/8
niño	4/11	1/10	1/9	1/8	1/8	1/8
jugaba	1/11	3/10	1/9	1/8	1/8	1/8
salta	1/11	2/10	1/9	1/8	1/8	1/8
con	1/11	1/10	2/9	1/8	1/8	1/8
carrito	2/11	1/10	1/9	1/8	1/8	1/8
EOS	1/11	1/10	2/9	2/8	1/8	2/8

Ejemplo: Smoothing Laplaciano

Por ejemplo, para el vector de probabilidades iniciales, Π se tienen:

	BOS
el	3/8
niño	1/8
jugaba	1/8
salta	1/8
con	1/8
carrito	1/8

Ejemplo: Smoothing Laplaciano

Ahora es posible calcular una probabilidad mayor a 0 para toda cadena en σ^*. Por ejemplo:

$$\begin{aligned} p(carrito, con, jugaba) &= p(carrito)p(con|carrito)p(jugaba|con) \\ &= \frac{1}{8}\frac{1}{8}\frac{1}{8} \\ &= \frac{1}{512} \approx 0.0019 \end{aligned}$$

Por su parte:

$$\begin{aligned} p(el, nino, jugaba) &= p(el)p(nino|el)p(jugaba|nino) \\ &= \frac{3}{8}\frac{4}{11}\frac{3}{10} \\ &= \frac{36}{880} \approx 0.04 \end{aligned}$$

Interpolación

Otra forma de suavizar las probabilidades es combinar varias medidas de probabilidad, garantizando que el resultado es una medida de probabilidad.

Debido que el espacio de probabilidades es **convexo**, se puede tomar $\lambda \in [0, 1]$ combinar dos medidas de probabilidad p y q de tal forma que:

$$\hat{p}(x) = \lambda p(x) + (1 - \lambda) q(x)$$

Se comprueba que es una medida de probabilidad:

$$\begin{aligned} \sum_x \hat{p}(x) &= \sum_x (\lambda p(x) + (1 - \lambda) q(x)) \\ &= \lambda \sum_x p(x) + (1 - \lambda) \sum_x \\ &= \lambda + 1 - \lambda = 1 \end{aligned}$$

Interpolación

De forma más general, sean p_i, $i \in \mathcal{N}$, medidas de probabilidad, entonces:

$$\hat{p}(x) = \sum_i \lambda_i p_i(x)$$

Condicionado a que $\sum_i \lambda_i = 1$. Esta medida se conoce como **interpolación lineal**.

De igual forma, podemos comprobar que se trata de una medida de probabilidad:

$$\begin{aligned}\sum_x \hat{p}(x) &= \sum_x \sum_i \lambda_i p_i(x) \\ &= \sum_i \lambda_i \sum_x p_i(x) \\ &= \sum_i \lambda_i = 1\end{aligned}$$

Interpolación y modelos del lenguaje

La interpolación lineal se puede utilizar para combinar modelos del lenguaje de n-gramas de distinto orden:

Interpolación lineal

Dado un n-grama $w^{(t-n+1)}, ..., w^{(t)}$ de elementos de un alfabeto Σ y dadas las distribuciones de los $(n-k)$-gramas, con $k \in \{0, ., n-1\}$, la distribución con interpolación lineal se define como:

$$\hat{p}(w^{(t)}|w^{(t-n+1)}, ..., w^{(t-1)}) := \sum_{k=0}^{n-1} \lambda_k p(w^{(t)}|w^{(t-n+1-k)}, ..., w^{(t-1)}) \quad (12)$$

donde $\lambda_k \in \mathbb{R}$ son parámetros que cumplen la igualdad $\sum_{k=0}^{n-1} \lambda_i = 1$.

Ejemplo: Interpolación

Supóngase que se busca la probabilidad de la cadena "'el gato come"' y se tienen las siguientes frecuencias de *n*-graamas ($n \leq 3$):

$$fr(el, gato, come) = 2, fr(el, gato) = 6, fr(gato, come) = 5, fr(gato) = 25, fr(come) = 20$$

Calcularemos las probabilidades de transición con interpolación lineal comenzando por 3-gramas como:

$$\hat{p}(w^{(3)}|w^{(1)}w^{(2)}) = \lambda_1 p(w^{(3)}|w^{(1)}w^{(2)}) + \lambda_2 p(w^{(3)}|w^{(2)}) + \lambda_3 p(w^{(3)})$$

Ejemplo: Interpolación

Debemos calcular las probabilidades de n-gramas con $n = 1, 2, 3$. Para el caso de **1-gramas**:

$$p(come) = \frac{fr(come)}{N} = \frac{1}{5}$$

Para **bigramas** tenemos:

$$p(come|gato) = \frac{fr(gato, come)}{fr(gato)} = \frac{1}{5}$$

Y para **trigramas**:

$$p(come|el, gato) = \frac{fr(el, gato, come)}{fr(el, gato)} = \frac{1}{3}$$

Ejemplo: Interpolación

Ahora, debemos seleccionar λ_k, $k = 1, 2, 3$, que cumplan que su suma sea igual a 1. Podemos tomar:

$$\lambda_0 = 0.5, \lambda_1 = 0.3, \lambda_2 = 0.2$$

Entonces:

$$\begin{aligned}
\hat{p}(come|el, gato) &= \lambda_0 \cdot p(come|el, gato) + \lambda_1 \cdot p(come|gato) + \lambda_2 \cdot p(come) \\
&= 0.5 \cdot p(come|el, gato) + 0.3 \cdot p(come|gato) + 0.2 \cdot p(come) \\
&= 0.5\frac{1}{3} + 0.3\frac{1}{5} + 0.2\frac{1}{5} \\
&\approx 0.266
\end{aligned}$$

Evaluación de modelos

Una medida de probabilidad sobre *n*-gramas depende de:

1. El *n* que se elija: el tamaño de los *n*-gramas.
2. El método de smoothing seleccionado.

¿Cómo podemos determinar cuál medida es mejor?

Necesitamos evaluar nuestro modelo, para esto necesitamos:

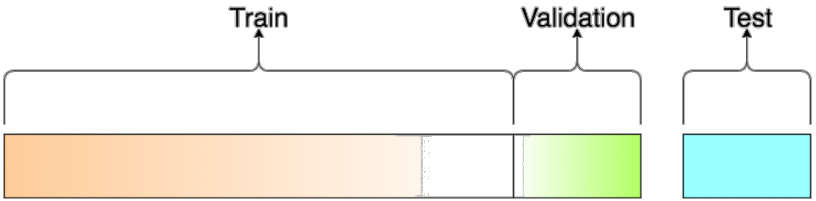
- Un corpus de evaluación. Que no haya sido visto para inferir el modelo.
- Una métrica de evaluación.

Corpus de evaluación

Cuando se construye un modelo del lenguaje, es común que se divida el corpus total de la siguiente forma:

- 70% para generar el modelo $\mu = (\Sigma, (A, \Pi))$:
  - 60% para entrenamiento.
  - 10% para validación (estimar hiperparámetros).
- 30% para evaluación.

NOTA: El corpus de evaluación no debe ser visto nunca en el proceso de generar el modelo.



Out Of Vocabulary

El corpus de entrenamiento y el de evaluación son conjuntos disjuntos, lo que puede provocar que el vocabulario Σ **no contenga algunas palabras** que se presenten fuera del vocabulario. Estas palabras se conocen como **Out Of Vocabulary** (OOV).

Se asume que las OOVs son elementos con poca probabilidad, y hay algunas formas de lidiar con ellas:

1. Asignar $p(OOV|w) = \frac{1}{N} = p(w|OOV)$ para todo $w \in \Sigma$ y donde $|\Sigma| = N$.
2. Asignar $p(OOV|w) = 1 = p(w|OOV)$, de tal forma que no afecte el producto.
3. Si $fr(w) \leq r$, para algún $r \geq 1$, entonces sustituir w por el símbolo OOV y estimar las probabilidades así.

Métricas de evaluación

El entrenamiento de un modelo del lenguaje de *n*-gramas es **no supervisado**. Una métrica como la **entropía empírica** puede usarse para evaluar:

$$H_E(P) = -\frac{1}{T}\sum_{t=1}^{T} \log_2 p(w^{(t)}|w^{(t-n+1)}...w^{(t-1)})$$

Asumimos que el corpus de evaluación es una cadena de longitud T. Aquí $p(w^{(t)}|w^{(t-n+1)}...w^{(t-1)})$ son las probabilidades de transición del modelo de *n*-gramas y cuando $t-1=0$ son probabilidades iniciales. Es decir, se evalúa $P=(A,\Pi)$.

Perplejidad

Una métrica más comúnmente utilizada es la **perplejidad**. Esta se ha adoptado como un estándar para evaluar modelos del lenguaje:

Perplejidad

La perplejidad Px de un modelo μ de tamaño T se define como:

$$Px(\mu) := \mu^{-1/T} \tag{13}$$

En el caso de los modelos de n-gramas esta se mide como:

$$Px(P) = \Big(\prod_{t=1}^{T} \frac{1}{p\big(w^{(t)} | w^{(t-n+1)} \dots w^{(t-1)}\big)} \Big)^{\frac{1}{T}}$$

La perplejidad está relacionada con la entropía, es fácil ver que $Px(P) = 2^{H_E(P)}$

Evaluación de diferentes modelos

Jurafsky (2000) muestra evaluaciones de diferentes modelos de *n*-gramas. Bajo un modelo entrenado con 38 millones de palabras y evaluado con 1.5 millones, la perplejidad es:

	1-grama	2-grama	3-grama
Px	962	170	109

Perplejidad claculada para diferentes modelos de *r*-gramas (Jurafski, 200).

Modelos del lenguaje neuronales

Modelos tradicionales

Un modelo (estadístico) del lenguaje es $\mu = (\Sigma, P)$, tal que Σ es vocabulario y P medida de probabilidad sobre Σ^*.

En los **modelos de n-gramas**, la probabilidad de las cadenas se estima por factores condicionales que toman en cuenta una historia **limitada**:

$$p(w_1 w_2 ... w_n) \approx p(w_1) \prod_{i=2}^{n} p(w_i | w_{i-1} ... w_{i-n+1})$$

La estimación depende del modelo de Markov, determinado por un vector de inicials Π y un tensor de probabilidades de transición A.

Problemas de los modelos de n-gramas

Los modelos de n-gramas, sin embargo, cuentan con diferentes problemas:

- Las probabilidades condicionales requieren de **smoothing** para no sobre-ajustar. No es fácil determinar cuáles son los **parámetros óptimos** del smoothing.
- Tienen **memoria acotada**: eventos/tokens lejanos a la palabra actual no influyen en esta. Esto no es una propiedad del lenguaje natural.
- Son **estáticos**, la longitud de la historia es fija.
- El desempeño depende de factores como el tamaño de n y el tipo de smoothing.

Modelos del lenguaje neuronales

Joshua Bengio (2003) propone estimar los modelos del lenguaje a partir de las **redes neuronales**. Sus objetivos son:

1. Asociar cada palabra en el vocabulario con un **vector distribuido** en $\mathbb{R}^d$.
2. Expresar la función de **distribución conjunta** de las secuencias de palabras por medio de estos vectores.
3. Aprender de manera **simultánea** los vectores distribuidos y los parámetros de la función de probabilidad.

Introducción a las redes FeedForward

Redes neuronales y probabilidad

las redes neuronales fungen como **estimadores** de probabilidad que nos servirán para asignar probabilidades a cadenas.

Las redes neuronales han demostrado ser capaces de aproximar con un buen nivel de exactitud las funciones de distribución de objetos.

Se pueden definir diferentes modelos del lenguaje basados en las redes neuronales:

- **Modelo neuronal (causal) de n-grama**: utiliza una red FeedForward para estimar probabilidades (Bengio, 2003).
- **Modelo neuronal recurrente**: Son modelos basados en RNNs que conservan memoria a largo plazo.
- **Modelo neuronal bidireccional**: Se basan en RNNs bidireccionales o transformer, consideran los elementos precedentes y subsequentes.
- **Modelo neuronal atencional**: Enriquecen la estimación con mecanismos de atención.

Clasificación

Las redes neuronales se basan en el **perceptrón**, que es un modelo de clasificación binaria. Una **clasificación binarias** busca una función:

$$f: \mathbb{R}^d \to \{0, 1\}$$

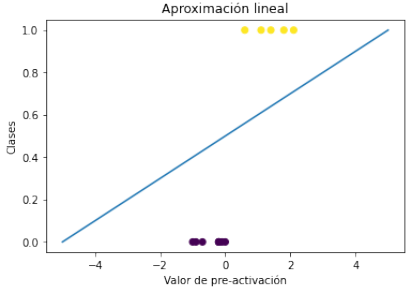
Dado un vector (un objeto caracterizado por d variables) buscamos saber si pertenece a una clase (1) o no (0).

Estas clasificación binaria puede pensarse como la **interacción de una neurona** que se activa o no se activa.

Clasificación

Al igual que en la regresión lienal, se puede tomar el conjunto supervisado y visualizarlo:

$$\mathcal{S} = \{(x, y) : \mathbb{R}^d, y \in \{0, 1\}\}$$



Sin embargo, una línea (o hiperplano) no se ajustarán adecuadamente a los datos.

Clasificación

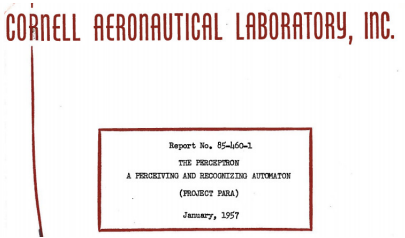
Necesitamos una función que se ajuste a los datos. Notamos:

- Existen dos clases. Por tanto, f debe tomar valores 0 ó 1, únicamente.
- Se espera que exista b tal que:

$$f(x) = \begin{cases} 1 & \text{si } f(x) > b \\ 0 & \text{si } f(x) \leq b \end{cases}$$

Si esta última condición se cumple, se dirá que los datos son **separables**. Cuando f es lineal se dice que son **linealmente separables**.

¿Qué es el perceptrón?


CORNELL AERONAUTICAL LABORATORY, INC.

Report No. 85-460-1

THE PERCEPTRON

A PERCEIVING AND RECOGNIZING AUTOMATON

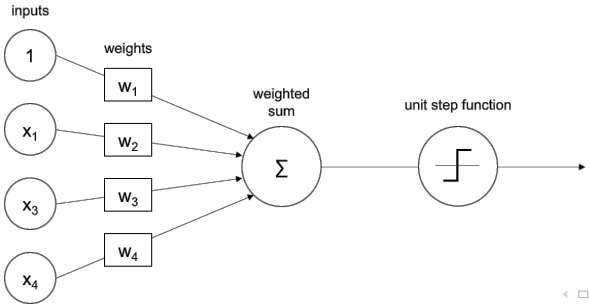
(PROJECT PARA)

January, 1957

- El perceptrón es un método para solucionar el problema de la clasificación binaria.
- La idea básica del perceptrón fue planteada por McCulloch y Pitts (1943) y después establecida algoritmicamente por Frank Rosenbluth (1957).
- La idea general del perceptrón es emular el funcionamiento de una neurona.

Perceptrón

- El perceptrón puede verse como el nodo de una gráfica dirigida que recibe como entrada un conjunto de rasgos (estímulos).
- Estos rasgos se codifican como un vector en $\mathbb{R}^d$.
- Los rasgos son ponderoados a través de pesos.
- Los valores de los rasgos ponderados se suman.
- La neurona se activa o no a partir de una función de activación.



Formalización del perceptrón

El perceptrón se puede ver como una función $f: \mathbb{R}^d \to \{0, 1\}$ dada por:

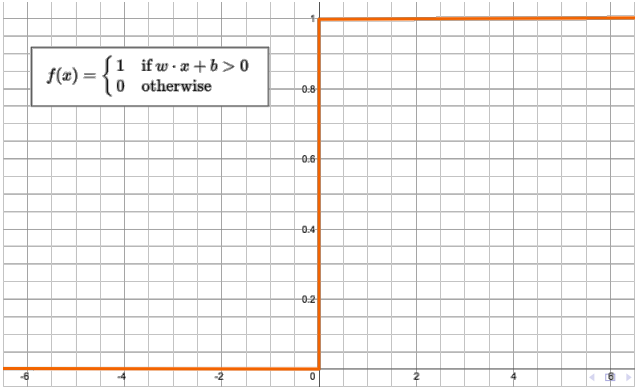
$$f(x) = \begin{cases} 1 & \text{si } \sum_{i=1}^{d} w_i x_i + b > 0 \\ 0 & \text{si } \sum_{i=1}^{d} w_i x_i + b \leq 0 \end{cases} \tag{14}$$

donde $w \in \mathbb{R}^d$ es un vector de pesos, $b \in \mathbb{R}$ es un bias o sesgo.
El sesgo b puede verse como un **umbral** que se debe superar para que se active la neurona.

Perceptrón y clasificación binaria

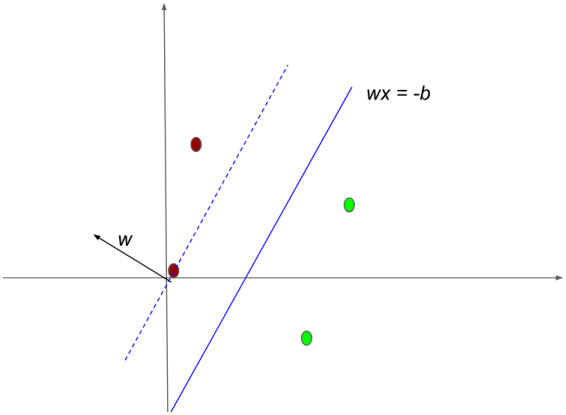
El perceptrón es capaz de clasificar datos en dos clases cuando estos datos son linealmente separables.

El perceptrón tiene semejanzas con la regresión lineal, pero, en tanto clasificación, puede visualizarse como una función escalonada.



Representación geométrica del perceptrón

Otra forma de visualizar el perceptrón es a partir de una línea (o hiperplano) que produce una separación lineal en el espacio de los datos.



Perceptrón: problemas lógicos

El perceptrón es capaz de resolver problemas lógicos del siguiente tipo:

$\mathbf{x_1}$	$\mathbf{x_2}$	**OR**	**AND**	**NOT** (x_1)
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

Perceptrón: problemas lógicos

La solución del problema OR es

$$w = \begin{pmatrix} 1 & 1 \end{pmatrix}, b = -0.5$$

La solución del problema AND es:

$$w = \begin{pmatrix} 1 & 1 \end{pmatrix}, b = -1.5$$

La solución del problema NOT (x_1) es:

$$w = \begin{pmatrix} -1 & 0 \end{pmatrix}, b = 0.5$$

Sin embargo, surge la pregunta ¿cómo determinar estos parámetros para cualquier clasificación binaria?

Parametros del perceptrón

El conjunto de paŕametros del perceptrón está determinado por:

$$\theta = \{w, b : w \in \mathbb{R}^d, b \in \mathbb{R}\}$$

El objetivo será encontrar una función $R(\theta)$ que, al minimizarla, obtenga los parámetros que clasifiquen adecuadamente los ejemplos X.

Sin embargo, la función de decisión del perceptrón **no es derivable**, por lo que no se pueden utilizar métodos basados en gradiente.

Aprendizaje en el perceptrón

Podemos observar cómo se comporta el perceptrón cuando comete un error. Supóngase $w = \begin{pmatrix} 1 & 1 \end{pmatrix}$ como antes, y $b = 0.5$. En el problema OR, tenemos:

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{wx + b}$	$\mathbf{f(x)}$	$\mathbf{y}$	$\mathbf{f(x) - y}$
0	0	0.5	1	0	1
0	1	1.5	1	1	0
1	0	1.5	1	1	0
1	1	2.5	1	1	0

La diferencia entre $f(x)$ y el valor esperado y es 0 cuando acierta y 1 cuando se equivoca. Si restamos esta diferencia obtendremos el valor de b que buscamos (-0.5).

Algoritmo del perceptrón

Dado un conjunto de datos $x^{(1)}, x^{(2)}, ..., x^{(N)}$ el objetivo del algoritmo del perceptrón es encontrar los parámetros $w \in \mathbb{R}^d$ y $b \in \mathbb{R}$.
Deben observarse el comportamiento de los errores:

- Si la neurona debio activarse, pero no lo hizo ($f(x) > 0$), entonces los valores de w deben ascender.
- Si no debió activarse, pero lo hizo ($f(x) \leq 0$), entonces los valores de w deben descender.

La variación de los pesos debe de variar con respecto a la regla:

$$\Delta w_i = -\eta\big(f(x^{(k)}) - y^{(k)}\big)x_i^{(k)} \tag{15}$$

$y^{(k)}$ es la activación esperada para el ejemplo $x^{(k)}$.
η se conoce como el rango de aprendizaje (pondera el valor del cambio en cada paso).

Algoritmo del perceptrón

Para el bias, se toma $x^{(k)} = \begin{pmatrix} x_1 & \cdots & x_d & 1 \end{pmatrix}^T$ el Podemos resumir el algoritmo del perceptrón como sigue:

Inicialización: Se escoge aleatoriamente un vector $w = \begin{pmatrix} w_1 & \cdots & w_d \end{pmatrix}^T \in \mathbb{R}^d$

Entrenamiento: Por cada iteración, y para todo $x^{(k)}$ hacer:

1. Calcular

$$f(x) = \begin{cases} 1 & \text{si } w \cdot x^{(k)} > 0 \\ 0 & \text{si } w \cdot x^{(k)} \leq 0 \end{cases}$$

2. Actualizar los pesos como:

$$w_i \leftarrow w_i + \nabla w_i$$

Finalización El algoritmo termina cuando el error es 0 o después de T iteraciones.

Ejemplo de aplicación

Supóngase que se tienen los siguientes datos:

$$X = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{pmatrix} \quad Y = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Para la inicialización, tomaremos los valores $w = \begin{pmatrix} 1 & 1 \end{pmatrix}^T$ y $b = 1$.

Ejemplo de aplicación

Una primera aplicación muestra que:

$$f(X) = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}^T$$

Aplicando la función de error, tenemos, por ejemplo que ($\eta = 1$):

$$\begin{aligned} w_1 &\leftarrow w_1 - \eta \sum_{k=1}^{4} [f(x^{(k)}) - y^{(k)}] x_1^{(k)} \\ &\leftarrow 1 - [(1-1)1 + 0 + 0 + (1-1)1] = 1 \end{aligned}$$

$$\begin{aligned} w_2 &\leftarrow w_1 - \eta \sum_{k=1}^{4} [f(x^{(k)}) - y^{(k)}] x_2^{(k)} \\ &\leftarrow 1 - [0 + (1-0)1 + 0 + (1-1)1] = 1 - 1 = 0 \end{aligned}$$

Ejemplo de aplicación

Resumiendo el algoritmo, obtenemos:

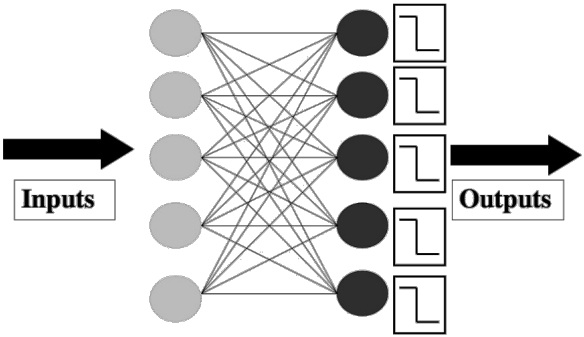
Iteración 1 $w = \begin{pmatrix} 1 & 0 \end{pmatrix}^T, b = -1$ / $f(X) = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}^T$

Iteración 2 $w = \begin{pmatrix} 3 & 1 \end{pmatrix}^T, b = 1$ / $f(X) = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}^T$

Iteración 3 $w = \begin{pmatrix} 3 & 0 \end{pmatrix}^T, b = -1$ / $f(X) = \begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix}^T$

Generalizaciones del perceptrón

La salida del perceptrón puede ser un vector $f(x) \in \{o, 1\}^m$, entonces, los parámetros son $w \in \mathbb{R}^{d \times m}$ y $b \in \mathbb{R}^m$:

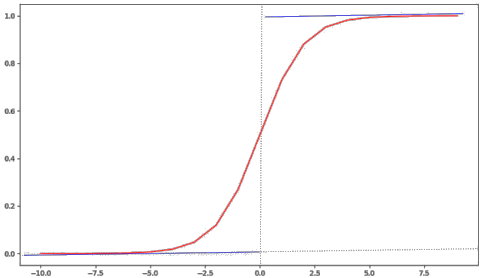


Generalizaciones del perceptrón

Las funciones indicadores ($f : X \to \{0, 1\}$) pueden ser aproximada por una función $\sigma : X \to [0, 1]$, llamada sigmoide y definida como:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{16}$$

En una red neuronal a es la preactivación $a = w \cdot x + b$.



Generalizaciones del perceptrón

- La función sigmoide así definida puede verse como una función de probabilidad, donde $p(X = 1) = \sigma(a)$ y $p(X = 0) = 1 - \sigma(a)$. Por lo que el perceptrón puede verse como un método de inferencia ($X \sim Ber[\sigma(a)]$).
- Si la función indicadora toma los valores -1 y 1, se puede utilizar la aproximación:

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

El perceptrón como función lógica

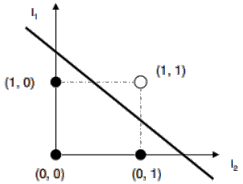
El perceptrón surge como un algoritmo para solucionar problemas lógicos.
Por tanto, es capaz de solucionar problemas **AND**, **OR** o **NOT**.
Pero no puede resolver un problema de tipo **XOR**.

x_1	x_2	**OR**	**AND**	**NOT** (x_1)	**XOR**
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	0	1
1	1	1	1	0	0

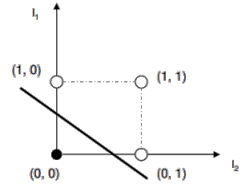
Problemas con el perceptrón

El perceptrón es un método de separación lineal. Por tanto su capacidad es limitada.

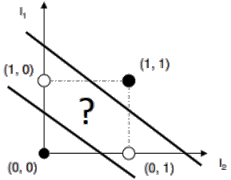
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1



XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



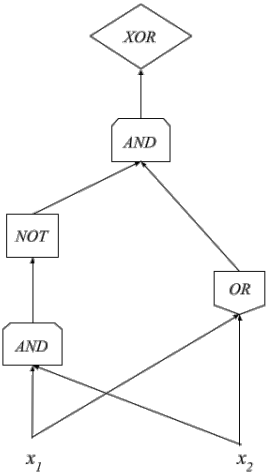
¿Cómo solucionar el problema XOR?

El problema XOR es un problema complejo, compuesto de elementos simples:

$$XOR(x_1, x_2) = AND(NOR(AND(x_1, x_2)), OR(x_1, x_2))$$

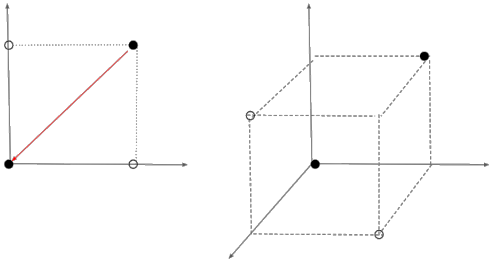
Podemos aplicar perceptrones de forma compuesta para solucionar este problema. Esto conforma una estructura gráfica más compleja.

En la última "capa" se trabaja con dos entradas: los puntos originales ahora son linealmente separables.



Otras soluciones al problema XOR

La idea es modificar los datos para que estos sean separables por un plano o hiperplano.
En otras palabras, buscamos transformar los datos. Esto se puede hacer de distintas formas.



Solución al problema XOR

- El problema XOR puede resolverse por la composición de diferentes perceptrones.
- En **términos lógicos** el problema XOR es la composición de problemas lógicos básicos.
- En **términos geométricos** se transforman los datos a un espacio donde sean linealmente separables.
- Se pueden utilizar **transformaciones lineales**; sin emabrgo éstas **son insuficientes** en muchos casos.

¿Cómo separar un problema XOR?

Por ejemplo, se puede tomar la transformación lineal:

$$L = \begin{pmatrix} 1 & 0 \\ -1 & -1 \end{pmatrix}$$

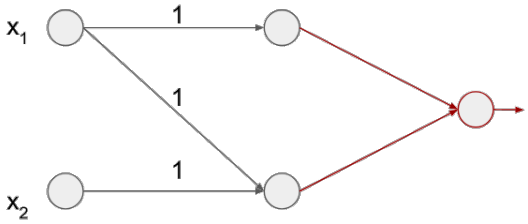
Esta tranformación hace que los datos sean

$$XL = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & -1 \\ 1 & -1 \\ 1 & 0 \end{pmatrix}, Y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Que se puede resolver con un perceptrón con parámetros $w = \begin{pmatrix} 0 & -1 \end{pmatrix}, b = -1$

Incorporación de la transformación en la red neuronal

La transformación L se puede expresar por medio de un grafo dirigido, particularmente como un perceptrón.



La parte roja representa el perceptrón que trabaja con los datos transformados.

Capas ocultas

Cada nodo en una capa oculta se conoce como **unidad oculta**. Cada unidad oculta puede verse como un perceptrón, que toma como entrada el vector de la capa anterior.

Los nodos intermedios en esta estructura conforman una capa oculta. Por tanto, podemos pensar una red neuronal con una capa oculta a partir de las funciones:

$$h(x) = g(W^{(1)}x + b^{(1)}) \tag{17}$$

$$f(x) = \phi(W^{(2)}x + b^{(2)}) \tag{18}$$

Donde $W^{(1)} \in \mathbb{R}^{d \times m}$, $b^{(1)} \in \mathbb{R}^m$, $W^{(2)} \in \mathbb{R}^{m \times o}$ y $b^{(2)} \in \mathbb{R}^o$ (o es el número de salidas). Además, g y ϕ son funciones de activación (se busca que sean **derivables**).

Redes neuronales pre-alimentadas

Una red neuronal pre-alimentada (o *feedforward NN*) es una red con K capas ocultas. Cuenta con:

1. Una pre-activación para cada capa $0 < k \leq K$, dada por:

$$a^{(k)}(x) = W^{(k)} h^{(k-1)}(x) + b^{(k)}$$

2. Una función de activación para cada capa $0 < k \leq K$:

$$h^{(k)}(x) = g(a^k(x))$$

3. Una activación de salida:

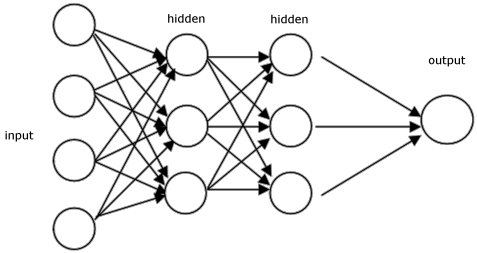
$$f(x) = \phi[W^{(K+1)} h^{(K)}(x) + b^{(K+1)}]$$

Redes neuronales pre-alimentadas

Las dimensiones de las matrices en las diferentes capas son:

1. En la capa de entrada, $W^{(1)}$ es una matriz de $d \times m_1$, con d dimensión de los vectores de entrada.
2. Para la k-ésima capa, $W^{(k)}$ es de $m_{k-1} \times m_k$.
3. En la capa de salida, $W^{(K+1)}$ es una matriz de $m_K \times o$, con o dimensiones de salida.

La **arquitectura** de una red neuronal feedforward específica el número de capas ocultas y las unidades de cada una (K y $m_1, ..., m_K$).



Funciones de pre-activación y activación

Dentro de un red feedforward se hablará de dos "tipos de funciones":

- **Funciones de pre-activación**: son funciones afines (lineales más una traslación) de la forma:

$$W^{(k)} h^{(k-1)}(x) + b^{(k)}$$

  Con $0 < k \leq K$. La concatenación de funciones afínes produce otra función afín.

- **Funciones de activación**: Permiten definir capas no lineales al aplicar una función no lineal $g : \mathbb{R}^{m_{k-1}} \to \mathbb{R}^{m_k}$ a la pre-activación. Para aplicar optimización por gradiente, g debe ser derivable.

Funciones de activación

Para las capas ocultas, se pueden determinar diferentes funciones de activación g:

- Sigmoide:

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

- Tangente hiperbólica:

$$\tanh(a) = \frac{e^{a} - e^{-a}}{e^{x} + e^{-a}}$$

- Softplus:

$$S(a) = \log(1 + e^{a})$$

- Rectified Linear Unit:

$$ReLU(a) = \max\{0, a\}$$

Funciones de activación

Estas funciones de activación son comúnes porque son fácilmente derivables. Sus **derivadas** son:

- Sigmoide:

$$\sigma'(a) = \sigma(a)(1 - \sigma(a))$$

- Tangente hiperbólica:

$$\tanh'(a) = 1 - \tanh^2(a)$$

- Softplus:

$$S(a) = \sigma(a)$$

- Rectified Linear Unit:

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

Activación en la capa de salida

La **capa de salida** muestra una función de activación particular, pues es la función que determinará la solución al problema.

Podemos proponer tres funciones de activación que dependen del problema a resolver.

1. **Función sigmoide:** Si se tiene una sola neurona de salida. Determina una probabilidad $p(Y = 1) = \sigma(a)$, donde a es la única salida.
2. **Función Softmax:** Generalización de la sigmoide. Se utiliza cuando se cuentan con diferentes salidas: $Sofmax(a_j)$, con $j \in 1, 2, ..., o$.
3. **Función lineal:** Cuando se trata de un problema de regresión (por ejemplo, en redes AutoEncoders).

Función Sotfmax

La función Softmax es la más comúnmente utilizada como activación en las capas de salida de redes neuronales.

Supóngase que se cuenta con $a_1, a_2, ..., a_o$ neuronas de salida, la función Softmax para la salida a_j se define como:

$$Softmax(a_j) = \frac{e^{a_j}}{\sum_{i=1}^{o} e^{a_i}}$$

La función Softmax define una función de probabilidad sobre las unidades de salida.

Para la clasificación en las clases de salida, tenemos que

$$\hat{y} = \arg\max_j \{Softmax(a_j)\}$$

Derivada de la función Softmax

La función Softmax puede verse como una función de $\mathbb{R}^o$ a $\mathbb{R}^o$ (vector de preactivación a vector de activaciones de salida).
Para obtener la derivada parcial de la función Softmax, denotemos ψ_j a la salida j.
Simplificaremos derivando sobre el logaritmo:

$$\frac{\partial \log \psi_j}{\partial a_i} = \frac{1}{\psi_j} \frac{\partial \psi_j}{\partial a_i}$$

De tal forma que:

$$\frac{\partial \psi_j}{\partial a_i} = \psi_j \frac{\partial \log \psi_j}{\partial a_i}$$

De esta forma, bastará obtener la derivada de $\log \psi_j$ para obtener la derivada de la función Softmax.

Derivada de la función Softmax

Por las propiedades del logaritmo, tenemos que:

$$\log \psi_j = \log \frac{e^{a_j}}{\sum_k e^{a_k}} = a_j - \log \sum_k e^{a_k}$$

Derivando esta expresión obtenemos que:

$$\frac{\partial \log \psi_j}{a_i} = \frac{\partial a_j}{\partial a_i} - \frac{\partial \log \sum_k e^{a_k}}{\partial a_i}$$

Del segundo elemento, tenemos que:

$$\frac{\partial \log \sum_k e^{a_k}}{\partial a_i} = \frac{1}{\sum_k e^{a_k}} \frac{\partial \sum_k e^{a_k}}{\partial a_i} = \frac{e^{a_i}}{\sum_k e^{a_k}} = \psi_i$$

Derivada de la función Softmax

Falta derivar el factor $\frac{\partial a_j}{\partial a_i}$. Este caso sencillo tenemos que:

$$\frac{\partial a_j}{\partial a_i} = \delta_{i,j}$$

Donde $\delta_{i,j}$ es la delta de Kronecker definida como:

$$\delta_{i,j} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Por tanto, la **derivada de la función Softmax** sobre la *i*-ésima salida es:

$$\frac{\partial \psi_j}{\partial a_i} = \psi_j(\delta_{i,j} - \psi_i) \tag{19}$$

Donde $\psi_j = Softmax(a_j)$ es la salida *j*-ésima de la red.

¿Por qué nos gustan tanto las redes neuronales?

Las redes neuronales FeedForward son una herramienta de gran importancia, pues permiten aproximar funciones $f: \mathbb{R}^d \to \mathbb{R}^o$.

Teorema: Aproximador uiversal

Una red neuronal f con m unidades ocultas (una sola capa) puede aproximar tanto como se quiera cualquier función Borel-medible.

El problema al que nos enfrentamos es elegir la arquitectura adecuada para nuestro problema y, más aún, tener una muestra suficientemente rica para lograr la aproximación.

Relevancia de las redes neuronales

Algunas **ventajas** del uso de redes son:

- Una red con una sola capa puede aproximar puede aproximar adecuadamente cualquier función de interés.
- Son capaces de encontrar representaciones abstractas que permitan una clasificación adecuada.
- Alta flexibilidad, permiten un buen número de arquitecturas.
- Requieren poco conocimiento del problema.

Algunas **desventajas** son:

- Requieren una buena cantidad de datos para converger adecuadamente.
- Requieren poder de procesamiento alto. Muchas veces poco accesible.
- Son poco interpretables.

Flexibilidad de las redes neuronales

Las redes FeedForward son una arquitectura **simple**.

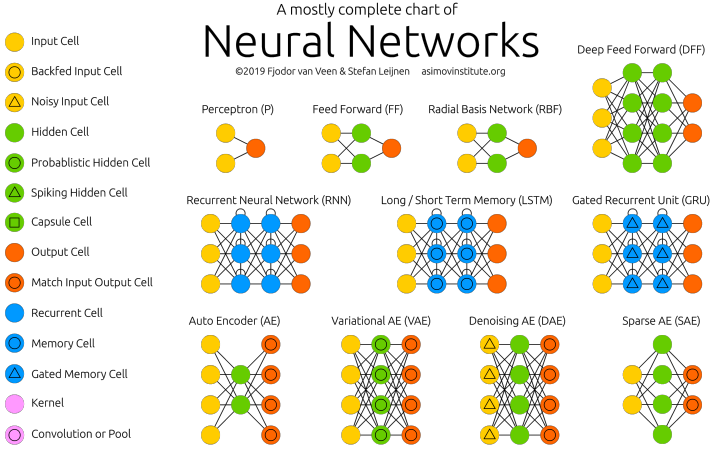
Pero éstas pueden verse como una particularidad de las **redes recurrentes**.

Asimismo, existen diferentes formas de conceptualizar una red, lo que determina diferentes algoritmos basados en redes neuronales.

Esto permite que las redes puedan configurarse de diferentes formas, permitiendo crear estructuras de redes que solucionen diferentes problemas.

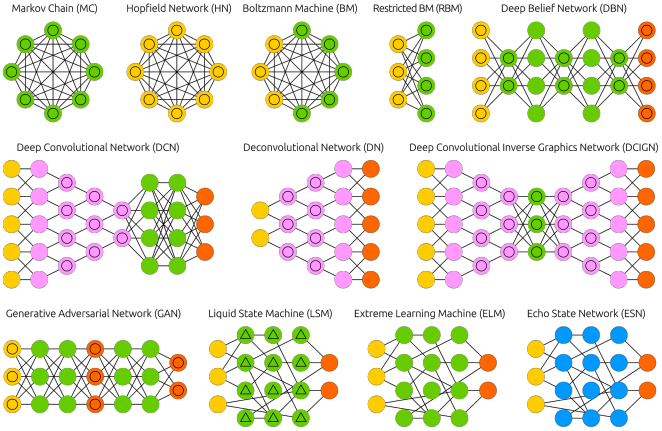
Familia de redes neuronales

Algunas arquitecturas conocidas son:



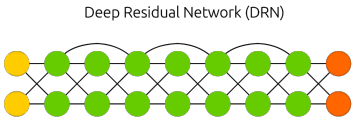
Familia de redes neuronales

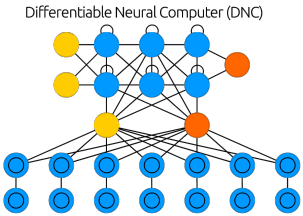
Otras arquitecturas son:

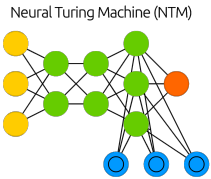


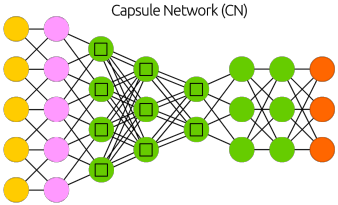
Familia de redes neuronales

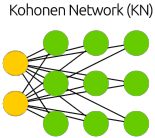
Otras arquitecturas son:

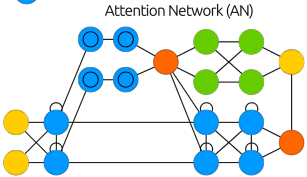












Ingredientes de una red neuronal

Para construir una red neuronal, entonces, se requiere:

- Un conjunto de datos de entrada X.
- Determinar un tipo de red neuronal que se acople al problema.
- Una función de salida ϕ (Softmax, sigmoide, escalón, ...).
- Una arquitectura $\{K, m_1, ..., m_K\}$ (tal que $|\theta| = \sum_{k=1}^{K} m_{k-1} \cdot m_k + \sum_{k=1}^{K} m_k$).
- Funciones de activación.
- Una función de riesgo $R(\theta)$, para el aprendizaje de parámetros.

Ejemplo de red neuronal

Supóngase que se tiene un conjunto de datos supervisados:

$$\mathcal{S} = \{(x, y) : x \in \mathbb{R}^5, y \in \{1, 2, 3\}\}$$

Buscamos clasificar los ejemplos en 3 clases.
Suponemos que los datos **no son linealmente separables**. Por tanto, requerimos de una red **FeedForward**.

Arquitectura de la red

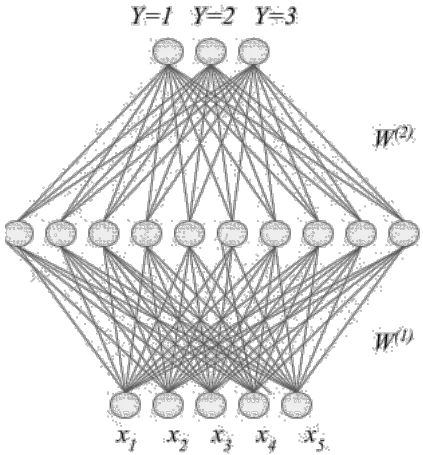
Definiremos nuestra red con las siguientes propiedades:

1. Una capa oculta con 10 unidades. Esto define una matriz $W^{(1)} \in \mathbb{R}^{10\times 5}$ y un bias $b^{(1)} \in \mathbb{R}^{10}$.
2. Una función de activación en la capa oculta $\tanh$.
3. Una función de salida dada por Softmax (por tanto, se trata de una probabilidad de clase).

La salida de la red contará con 3 nodos (uno para cada clase) determinados po:

$$\phi_y(x) = p(Y = y|x)$$

Implementación de la red neuronal



Dado un ejemplo de entrada x, la red calculará:

- Capa oculta ($\mathbb{R}^5 \rightarrow \mathbb{R}^{10}$):

$$h = \tanh(W^{(1)}x + b^{(1)})$$

- Pre-activación de la capa de salida (para una sola clase):

$$a = W^{(2)}h + b^{(2)}$$

- Activación en la capa de salida (probabilidad de una clase):

$$\phi_y(x) = \frac{e^{a_y}}{\sum_{y'} e^{a_{y'}}}$$

Implementación de la red neuronal

Finalmente, se determinará que x pertenece a una clase a partir de la regla:

$$\hat{y} = \arg\max_{y} \phi_y(x)$$

A todo este paso se le conoce como **Forward**.
La red así constituida tiene un **número de parámetros**

$$|\theta| = (5 \times 10 + 10) + (10 \times 3) + 3 = (50 + 10) + (30 + 3) = 83$$

El problema de aprendizaje consiste en encontrar los parámetros óptimos:

$$\theta = \{W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}\}$$

¿Cómo se puede realizar esto en redes de este tipo?

Problema de aprendizaje

Las redes neuronales, como otros problemas de aprendizaje, pueden verse como una función $\phi : X \rightarrow Y$, tal que:

$$\phi(x) = \hat{y}$$

Tal que $\hat{y}$ representa una predicción que busca ser cercana a una categoría de supervización y.

El **problema de aprendizaje** se puede plantear como:

> *El proceso de aprendizaje consiste de escoger una función apropiada ϕ^* de un conjunto de funciones $\{\phi : X \rightarrow Y\}$.*

Parámetros

Las funciones que nos interesan son aquellas que dependen de un conjunto de parámetros (pesos). Es decir, aquellas funciones:

$$\phi : X \times \Theta \rightarrow Y$$

donde $\Theta = \{\theta\}$ es un conjunto de parámetros.

En las redes neuronales, estos parámetros representans los pesos de la red:

$$\theta = \{W^{(1)}, W^{(2)}, ..., W^{(K)}, b^{(1)}, b^{(2)}, ..., b^{(K)}\}$$

El **problema de aprendizaje** se reduce a encontrar el conjunto de parámetros $\theta \in \Theta$ que nos den la función adecuada $\phi(x; \theta) = \hat{y} \approx y$.

Búsqueda de parámetros

En el caso del perceptrón, los parámetros se obtienen por una regla simple (algoritmo del perceptrón).

Requerimos de una forma general de obtener los parámetros sin importar el tipo y la arquitectura de una red.

Determinado un tipo y arquitectura neuronal, se define un **espacio de parámetros** con la propiedad de ser **continuo**. Denotaremos un conjunto de parámetros como:

$$\theta \in \Theta$$

Emmpirical risk minimization

Para encontrar el conjunto de parámetros óptimo $\hat{\theta}$ pasamos del problema de aprendizaje a un problema de **optimización**.

Función de riesgo

A una función $R : \Theta \rightarrow \mathbb{R}$, donde Θ es un espacio de soluciones, se le denomina función de riesgo (función objetivo/costo).

Hablamos del problema de **minimización del riesgo empírico** cuando, a aprtir de datos de entrenamiento, buscamos:

$$\hat{\theta} = \arg\min_{\theta \in \Theta} R(\theta)$$

Funciones de riesgo

La **función de riesgo** puede definirse de cualquier forma que nos ayude a solucionar el problema (asumismo $Y \sim q$).

Para las redes neuronales es común utilizar alguna función del siguiente tipo (ϕ distribución estimada por la red):

- Divergencia KL (GANs, Variational AutoEnconders):

$$R(\theta) = \mathbb{E}_q[\ln \frac{q(x)}{\phi(x;\theta)}]$$

- Entropía cruzada (FeedForwards, Recurrentes):

$$R(\theta) = \mathbb{E}_q[\ln \phi(x;\theta)]$$

- Error cuadrático medio (AutoEncoders, Denoising AutoEncoders):

$$R(\theta) = \frac{1}{2}\mathbb{E}[(y - \phi(x;\theta))^2]$$

Función de riesgo en redes FeedForward

En las redes FeedForward, la función $\phi(x) = \phi(x; \theta)$ constituye la red (forward).

Este tipo de redes se aplican a problemas **supervisados**, donde se cuenta con un conjunto de entrenamiento:

$$\mathcal{S} = \{(x, y) : x \in \mathbb{R}^d, y \sim q\}$$

De tal forma que problema de aprendizaje se plantea como:

$$\arg\min_{\theta \in \Theta} R(\theta) = \arg\min_{\theta \in \Theta} \sum_x \sum_y y \ln \phi(x; \theta) \tag{20}$$

¿Cómo optimizar los parámetros?

Como muchos problemas de optimización, podemos encontrar el mínimo a partir del gradiente de la función. Buscamos:

$$\nabla_\theta R(\theta) = 0$$

Encontrar estos valores dependerá de las propiedades que tenga la función R.

Por ejemplo, el teorema de valores extremos nos dice que una función continua alcanza su mínimo (y máximo) en un intervalo cerrado.

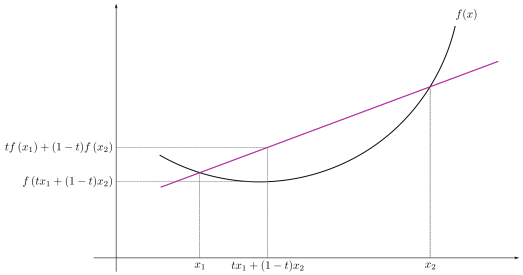
Sin embargo, este no es el caso para muchas de las funciones de riesgo.

Convexidad

Una **función convexa** es aquella que cumple que para toda $t \in [0, 1]$:

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

Una función de este tipo siempre tendrá un mínimo.



Pero, de nuevo, las funciones de riesgo no siempre serán convexas.

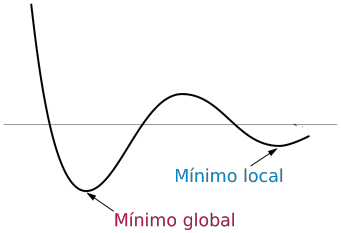
Mínimos globales y locales

El **mínimo global** es la $\hat{\theta}$ tal que $\forall\theta \in \Theta$:

$$R(\hat{\theta}) < R(\theta)$$

A esta solución se le llamará **mínimo global**.

Si la desigualdad se cumple sólo para una vecindad ($\{\theta : ||\theta - \hat{\theta}|| \leq r\}$), se le denomina **mínimo local**.



Gradiente descendiente

La derivada de la función $R(\theta)$ nos da información sobre el comportamiento de la función:

- Si $\nabla_\theta R(\theta)$ es positiva, la función asciende. Se debe disminuir el valor de θ:

$$\Delta\theta = -\nabla_\theta R(\theta) < 0$$

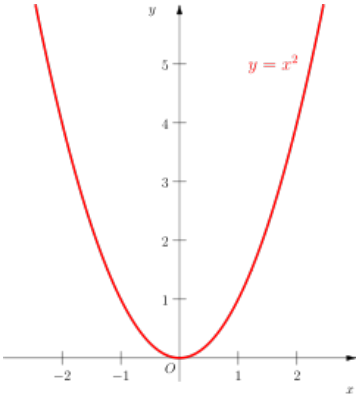
- Si $\nabla_\theta R(\theta)$ es negativa, la función desciende. Se debe aumentar el valor de θ:

$$\Delta\theta = -\nabla_\theta R(\theta) > 0$$

La regla de actualización puede verse como:

$$\theta \leftarrow \theta - \nabla_\theta R(\theta)$$

Probelma de gradiente descendiente



- Podemos definir, una función $f(x) = x^2$
- Sabemos que $f'(x) = 2x$, por tanto, para minimizar la función:

$$x \leftarrow x - 2x$$

- Si inicializamos con $x = 1$, entonces: $x \leftarrow 1 - 2 \cdot 1 = -1$
- Aplicando de nuevo el método de GD, tendremos $x = 1$. El algoritmo nunca convergerá al mínimo.

Rango de aprenndizaje

El problema anterior puede solucionarse modificando la regla de actualización $x \leftarrow x - \frac{1}{2} f'(x)$ (convergerá en el primer paso).
En general se toma un **rango de aprendizaje** $\eta \in \mathbb{R}$ para controlar la forma en que el GD desciende sobre la función.
El método de **gradiente descendiente** (GD) se define por la regla:

$$\theta \leftarrow \theta - \eta \nabla_\theta R(\theta)$$

- Si η es muy grande, el algoritmo podría nunca converger.
- Si η es muy pequeño puede demorarse mucho en converger.

Algorimto para GD

Algorithm Gradiente descendiente

procedure $\mathrm{GD}(R, T, \eta)$
  Inputs: $R : \Theta \to \mathbb{R}$ riesgo; $T > 0$ número de iteraciones; η rango de aprendizaje.
  Inicializa $\theta^{(0)} \in \Theta$ aleatoriamente
  for t **from** 1 **to** T **do**
    Calcular: $R(\theta^{(t)})$ y $\nabla_{\theta^{(t)}} R(\theta^{(t)})$
    if $R(\theta^{(t)}) = 0$ **then** stop
    Actualizar: $\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla_{\theta^{(t)}} R(\theta^{(t)})$
  end for
  return $\hat{\theta} \leftarrow \theta^{(T)}$
end procedure

GD en el perceptrón

La función del perceptrón no es derivable, pero podemos cambiar esta función por la sigmoide:

$$\phi(x) = \sigma(wx)$$

Y utilizar como función de riesgo la entropía cruzada:

$$\begin{aligned} R(w) &= -\sum_x y \ln \phi(x) + (1-y)\ln(1-\phi(x)) \\ &= -\sum_x y \ln \sigma(wx) + (1-y)\ln \sigma(-wx) \end{aligned}$$

GD en el perceptrón

Buscaremos las derivadas parciales para cada ejemplo x:

$$\frac{\partial R}{\partial w_i} = \frac{\partial y \ln \phi(x) + (1-y)\ln(1-\phi(x))}{\partial w_i} \tag{21}$$

$$= \frac{\partial y \ln \sigma(wx) + (1-y)\ln \sigma(-wx)}{\partial w_i} \tag{22}$$

$$= (y - \phi(x))x_i \tag{23}$$

Así, la optimización para un perceptrón de este tipo es (en cada ejemplo):

$$w_i \leftarrow w_i - \eta(y - \phi(x))x_i$$

Implementaciones del Gradiente descendiente

Según al forma en que la actualización se haga con respecto a los ejemplos, tenemos:

- **(Batch) Gradient Descent**: En cada actualización de los pesos, se deben observar todos los ejemplos
- **Stochastic Gradient Descent**: Por cada iteración, SGD actualiza cada uno de los ejemplos (de forma aleatoria): $\theta \leftarrow \theta - \eta \nabla_\theta R(\theta; x, y)$
- **Mini-batch Gradient Descent** considera mini-batches de tamaño n. Un mini-batch es un subconjunto aleatorio de pares de entrenamiento: $Batch_n = \{(x^{(i)}, y^{(i)}) : i = 1, ..., n\} \subseteq X$

Adam

Otro método común basado en GD, pero que modifica el rango de aprendizaje, es **Adam**, determinado por:

$$\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{\hat{\nu}} + \epsilon}\hat{m}$$

Tal que ($\beta_1, \beta_2 \in [0, 1]$):

$$\hat{m} = \frac{m}{1 - \beta_1}$$
$$\hat{\nu} = \frac{\nu}{1 - \beta_2}$$

Y los valores m y ν se actualizan en cada época por:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_\theta R(\theta)$$
$$\nu \leftarrow \beta_2 \nu + (1 - \beta_2)[\nabla_\theta R(\theta)]^2$$

Optimización en redes multicapa

Si bien la optimización con GD optimiza los pesos adecuadamente, surge el problema de tener redes con muchas capas.

En estos casos, la red puede verse como una composición de funciones:

$$p(Y = y|x) = \phi(h^{(K)}(\cdots h^{(1)}(x)))$$

Así, para encontrar la derivada de la función de riesgo $R(\theta)$ se puede utilizar la regla de la cadena varias veces:

$$\nabla_\theta R(\theta) = \nabla_\phi R \nabla_{h^{(K)}} \phi \cdots \nabla_{\theta_{ij1}} h^{(1)}$$

Ya que derivamos sobre los parámetros θ, también debe observarse que estos se encuentran a través de todas las capas.

Algoritmo de backpropagation: intuición

Aplicar la regla de la cadena para obtener las derivadas sobre todos los pesos de la red resulta muy complejo.

El algoritmo de **backpropagation** es capaz de obtener la derivada propagando la derivación desde la capa de salida hasta la de entrada.

Ejemplo: Considérese una red con una sola capa cuyo forward esté dado por:

- $a^{(1)} = W^{(1)}x$
- $h = \tanh(a)$
- $a^{(2)} = W^{(2)}h$
- $\phi(x) = \sigma(a^{(2)})$

Supóngase que la función de riesgo está dada por $R(\theta) = \frac{1}{2}\sum_x \sum_y (y - \phi(x))^2$

Algoritmo de backpropagation: intuición

En primer lugar, obtendremos las derivadas parciales de la función de riesgo sobre los parámetros. En este caso, el conjunto de parámetros es:

$$\theta = \{W^{(1)}, W^{(2)}\}$$

Podemos denotar $\theta_{i,j,k} = W_{ij}^{(k)}$; si aplicamos **SGD** derivaremos sólo sobre un ejemplo:

$$\begin{aligned}\frac{\partial R}{\partial W_{ij}^{(2)}} &= \frac{\partial}{\partial W_{ij}^{(2)}} \frac{1}{2} \sum_y (y - \phi(x))^2 \\ &= \sum_y \frac{\partial}{\partial W_{ij}^{(2)}} \frac{1}{2} (y - \phi(x))^2\end{aligned}$$

Lo cual nos dice que tenemos que derivar sobre cada una de las neuronas de salida de la red.

Algoritmo de backpropagation: intuición

Necesitamos obervar las derivadas para cada una de las matrices de parámetros. Para la **capa de salida**, tenemos:

$$\begin{aligned}\frac{\partial R}{\partial W_{ij}^{(2)}} &= \frac{\partial R}{\partial \phi_i}\frac{\partial \phi_i}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial W_{ij}^{(2)}} \\ &= (\phi_i(x) - y_i)\sigma(a^{(2)})(1-\sigma(a^{(2)}))h_j\end{aligned}$$

Pues tenemos que: $\frac{\partial R}{\partial \phi} = \frac{\partial}{\partial \phi}\frac{1}{2}(y_i - \phi_i(x))^2 = -(y_i - \phi_i(x))$

Mientras que $\frac{\partial \phi_i}{\partial a_i^{(2)}}$ es la derivada del sigmoide; finalmente:

$$\frac{\partial a_i^{(2)}}{\partial W_{ij}^{(2)}} = \frac{\partial}{\partial W_{ij}^{(2)}} W_{i\cdot}^{(2)} h = \sum_k \frac{\partial}{\partial W_{ij}^{(2)}} W_{ik}^{(2)} h_k = h_j$$

Algoritmo de backpropagation: intuición

Para la **capa oculta** tenemos que obtener la derivada parcial:

$$\frac{\partial R}{\partial W_{ij}^{(1)}} = [\frac{\partial R}{\partial \phi}\frac{\partial \phi}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial h_i}][\frac{\partial h_i}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial W_{ij}^{(1)}}]$$

La primera parte corresponde a la derivación:

$$\begin{aligned}\frac{\partial R}{\partial \phi}\frac{\partial \phi}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial h_i} &= \sum_q \frac{\partial R}{\partial \phi_q}\frac{\partial \phi_q}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial h_i} \\ &= \sum_q \frac{\partial R}{\partial \phi_q}\frac{\partial \phi_q}{\partial a^{(2)}} W_{iq}^{(2)} \\ &= \sum_q W_{iq}^{(2)}(\phi_q(x) - y_q)\sigma_q(a^{(2)})(1 - \sigma_q(a^{(2)}))\end{aligned}$$

Algoritmo de backpropagation: intuición

Para la segunda aprte de la derivación tenemos que derivar la activación $\frac{\partial h_i}{\partial a^{(2)}}$ (tangente hiperbólica) y la pre-activación:

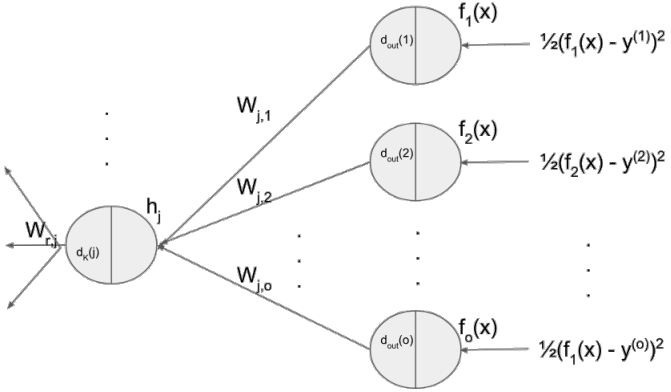
$$\frac{\partial a^{(2)}}{\partial W_{ij}^{(1)}} = \sum_k \frac{\partial}{\partial W_{ij}^{(1)}} W_{ik}^{(1)} x_k = x_j$$

Por tanto, para la **capa oculta** tenemos que obtener la derivada parcial:

$$\begin{aligned}\frac{\partial R}{\partial W_{ij}^{(1)}} &= [\frac{\partial R}{\partial \phi}\frac{\partial \phi}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial h_i}][\frac{\partial h_i}{\partial a^{(1)}}\frac{\partial a^{(1)}}{\partial W_{ij}^{(1)}}] \\ &= [\sum_q W_{iq}^{(2)}(\phi_q(x) - y_q)\sigma(a_q^{(2)})(1 - \sigma(a_q^{(2)}))][(1 - \tanh^2)x_j]\end{aligned}$$

Algoritmo de backpropagation: intuición

La idea del método de retro-propagación es propagar el error hacia tras de la red, en sentido opuesto:



Algoritmo de backpropagation

Input: Una función de riesgo $R(\theta) \equiv R \circ \phi \circ h^{(K)} \circ h^{(K-1)} \circ \cdots \circ h^{(1)}$ donde ϕ es una red neuronal con pesos $\theta = \{W^{(K+1)}, W^{(K)}, ..., W^{(1)}\}$.

Inicialización: Para la función en la capa de salida calcular las variables:

$$d_{out}(j) = \frac{\partial R}{\partial \phi_j(a)} \frac{\partial \phi_j(a)}{\partial a}$$

Y obtener la derivada parcial: $\frac{\partial R}{\partial W_{ij}^{(K+1)}} = d_{out}(j) h_i^{(K)}$

Inducción: Para toda capa oculta k con $k = K, K-1, ..., 1$ calcular las variables:

$$d_k(j) = \frac{\partial h_j^{(k)}}{\partial a^{(k)}} \sum_q W_{j,q}^{(k+1)} d_{k+1}(q)$$

Y obtener la derivada parcial como: $\frac{\partial R}{\partial W_{i,j}^{(k)}} = d_k(j) h_i^{(k-1)}$, donde $h^{(0)} = x$.

Output: Derivada de la función de riesgo a partir de las derivadas parciales.

Algoritmo de backpropagation: intuición

En resumen, la **capa de salida** depende de tres derivadas parciales:

$$\frac{\partial R}{\partial W_{ij}^{(2)}} = \frac{\partial \mathbf{R}}{\partial \phi_{\mathbf{i}}} \frac{\partial \phi_{\mathbf{i}}}{\partial \mathbf{a^{(2)}}} \frac{\partial a^{(2)}}{\partial W_{ij}^{(2)}}$$

Mientras que en la **capa oculta** podemos escribirla como:

$$\frac{\partial R}{\partial W_{ij}^{(1)}} = [\sum_{q} \frac{\partial \mathbf{R}}{\partial \phi_{\mathbf{q}}} \frac{\partial \phi_{\mathbf{q}}}{\partial \mathbf{a^{(2)}}} W_{iq}^{(2)}][\frac{\partial h_i}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial W_{ij}^{(1)}}]$$

En ambos derivadas, podemos encontrar un factor común, por lo que podemos escribir este factor como una variable:

$$d_{out}(i) = \frac{\partial R}{\partial \phi_i} \frac{\partial \phi_i}{\partial a} \tag{24}$$

Algoritmo de backpropagation: capas ocultas

Cuando agregamos más capas ocultas, podremos observar un patrón. Por ejemplo, pensemos que derivamos una red con **tres capas ocultas** ($\theta = \{W^{(1)}, W^{(2)}, W^{(3)}\}$):

$$\frac{\partial R}{\partial W_{ij}^{(2)}} = \frac{\partial \mathbf{R}}{\partial \phi} \frac{\partial \phi}{\partial \mathbf{a^{(3)}}} \frac{\partial \mathbf{a^{(3)}}}{\partial \mathbf{h_i^{(2)}}} \frac{\partial \mathbf{h_i^{(2)}}}{\partial \mathbf{a^{(2)}}} \frac{\partial a^{(2)}}{\partial W_{ij}^{(2)}}$$

Podemos ver que en la capa inferior pasa algo similar a la capa de salida:

$$\begin{aligned} \frac{\partial R}{\partial W_{ij}^{(1)}} &= \frac{\partial R}{\partial \phi} \frac{\partial \phi}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial h_i^{(1)}} \frac{\partial h_i^{(1)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial W_{ij}^{(1)}} \\ &= [\sum_q \frac{\partial \mathbf{R}}{\partial \phi} \frac{\partial \phi}{\partial \mathbf{a^{(3)}}} \frac{\partial \mathbf{a^{(3)}}}{\partial \mathbf{h_q^{(2)}}} \frac{\partial \mathbf{h_q^{(2)}}}{\partial \mathbf{a^{(2)}}} W_{iq}^{(2)}] \frac{\partial h_i^{(1)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial W_{ij}^{(1)}} \end{aligned}$$

El patrón que se repite se almacenará como una variable $d_3(i)$.

Algoritmo de backpropagation

Para describir un algoritmo general, primero obtendremos la derivada sobre la capa de salida. Sea $a = W^{(K+1)} h^{(K)}(x) + b^{(K+1)}$. Debemos obtener las siguientes variables:

$$d_{out}(j) = \frac{\partial R}{\partial \phi_j(a)} \frac{\partial \phi_j(a)}{\partial a}$$

Para obtener la derivada, nótese que:

$$\begin{aligned} \frac{\partial R}{\partial W_{i,j}^{(K+1)}} &= \frac{\partial R}{\partial \phi_j(a)} \frac{\partial \phi_j(a)}{\partial a} h_i^{(K)} \\ &= d_{out}(j) h_i^{(K)} \end{aligned}$$

Derivación del error

Posteriormente, obtendremos la derivada sobre las capas ocultas. Definimos las variables:

$$
\begin{aligned}
d_k(j) &= \frac{\partial R}{\partial h^{(k+1)}} \frac{\partial h^{(k+1)}}{\partial h_j^{(k)}} \frac{\partial h_j^{(k)}}{\partial a^{((k))}} \\
&= \frac{\partial h_j^{(k)}}{\partial a^{(k)}} \sum_q W_{j,q}^{(k+1)} d_{k+1}(q)
\end{aligned}
$$

En este caso, la derivada viene dada por ($0 \leq k \leq K$, $h^{(0)} = x$):

$$
\begin{aligned}
\frac{\partial R}{\partial W_{i,j}^{(k)}} &= [\frac{\partial h_j^{(k)}}{\partial a^{(k)}} \sum_q W_{j,q}^{(k+1)} d_{k+1}(q)] h_i^{(k-1)} \\
&= d_k(j) h_i^{(k-1)}
\end{aligned}
$$

Algoritmo de backpropagation

Algorithm Backpropagation

procedure BACKPROPAGATION(R)
  Inputs: $R(\theta) \equiv R \circ \phi \circ h^{(K)} \circ h^{(K-1)} \circ \cdots \circ h^{(1)}$, ϕ es una red neuronal con pesos $\theta = \{W^{(K+1)}, W^{(K)}, ..., W^{(1)}\}$.
  $d_{out}(j) = \frac{\partial R}{\partial \phi_j(a)} \frac{\partial \phi_j(a)}{\partial a}$, $j = 1, ..., o$ — Inicialización
  $\frac{\partial R}{\partial W_{ij}^{(K+1)}} = d_{out}(j) h_i^{(K)}$, $j = 1, ..., o$
  for k **from** K **to** 1 **do**
    $d_k(j) = \frac{\partial h_j^{(k)}}{\partial a^{(k)}} \sum_q W_{j,q}^{(k+1)} d_{k+1}(q)$, $j = 1, ..., m_k$ — Inducción
    $\frac{\partial R}{\partial W_{i,j}^{(k)}} = d_k(j) h_i^{(k-1)}$, $j = 1, ..., m_k$ y $h^{(0)} = x$
  end for
  return $\frac{\partial R}{\partial W_{i,j}^{(k)}}$, para todo k, i, j
end procedure

Algoritmo de aprendizaje en redes FeedForward

1: **procedure** FitFeedForward(Red, R, T)
2: **Inicializa**: $\theta = \{W^1, ..., W^{(K+1)}\}$ aleatoriamente.
3: **for** t **from** 1 **to** T **do**
4: **for** x, y **in** Batch(X, Y) **do**
5: **Forward**:
6: $f(x) = \phi\left(W^{(K+1)} h^{(K)}(x) + b^{(K+1)}\right)$
7: **Backward**:
8: Loss $\leftarrow R(\theta; x, y)$
9: $\frac{\partial R}{\partial W_{i,j}^{(k)}} \leftarrow$ Backpropagation$(R(\theta; x, y))$
10: $W_{i,j}^{(k)} \leftarrow W_{i,j}^{(k)} - \eta \cdot \frac{\partial R}{\partial W_{i,j}^{(k)}}$
11: **end for**
12: **end for**
13: **return** Red con pesos actualizados θ
14: **end procedure**

Ejemplo

Considérese una red neuronal con una sola capa oculta y con dos salidas. Supóngase que los bias son 0. Para la capa oculta, considérese que:

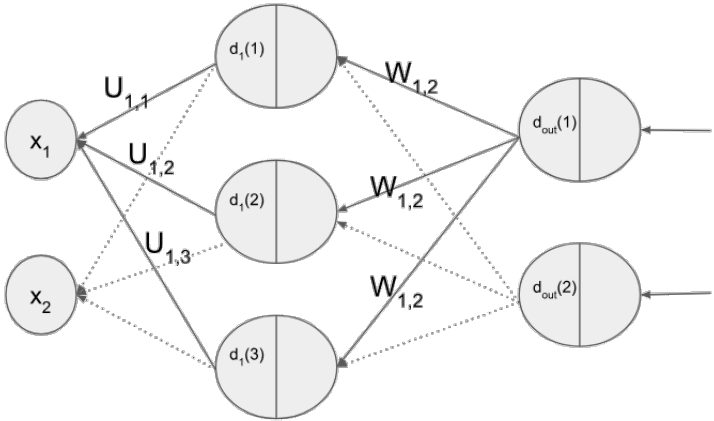
$$h(x) = Softplus(Ux)$$

Para la capa de salida, supóngase que:

$$f(x) = \sigma(Wh(x))$$

Elijamos como función de riesgo al error cuadrático medio.

Ejemplo



Ejemplo

Necesitamos derivar el sigmoide. Tenemos que:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Como asumimos que la red neuronal tenía sólo una salida:

$$d_{out}(j) = (f_j(x) - y(j))f_j(x)(1 - f_j(x))$$

De donde:

$$W_{i,j} \leftarrow W_{i,j} - \eta d_{out}(j)h_i(x)$$

Ejemplo

Para las capas ocultas, ya que tenemos una función lineal sobre x, tenemos que:

$$d_1(j) = \sigma(a) \sum_q W_{j,q} d_{out}(j)$$

Y la actualización de los pesos se da como:

$$U_{i,j} \leftarrow U_{i,j} - \eta d_1(j) x_i$$

Modelo del lenguaje neuronal (FeedForward)

Modelo del lenguaje neuronal

El modelo del lenguaje neuronal que propone Bengio (2003), se basa en una arquitrctura de FeedForward que cuenta con los siguientes componentes:

- **Entrada:** Una palabra (o historia) que alimenta la red. Se suele utilizar representación indezal.
- **Capa de embedding:** Representaciones distribuidas de las entradas.
- **Capa oculta:** Con activación de tangente hiperbólica. Asegura que tengamos un aproximador universal.
- **Capa de salida:** Estima las probabilidades de las palabras del vocabulario dada la entrada.

Representaciones de las entradas

El vector de entrada de la red es una representación **indexal**, llamada one-hot representation. Dado una palabra w_i con un índice i, su representación en un vector one-hot se define como:

$$x(i)_k := \begin{cases} 1 & \text{si } k = i \\ 0 & \text{si } k \neq i \end{cases} \tag{25}$$

Así, para la palabra w_3 con índice 3 se tiene el vector:

$$x(i) = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Capa de embedding

Una vez obtenido la representación indexical de las palabras, estas se representan en un vector distribuido (con entradas reales) a partir de una capa de embedding. Esta capa se define como:

$$C(i) := Cx(i) \tag{26}$$

Tal que $C \in \mathbb{R}^{d \times N}$, donde $|\Sigma| = N$ y d es un hiperparámetro que define la dimensión de estas representaciones.

Capa de embedding

La capa de embedding es **lineal**, pues de esta forma se facilita el acceso a los vectores distribuidos que representan las palabras. El índice i apunta hacia la i-ésima columna de la matriz C

$$Cx(2) = \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} c_{1,2} \\ c_{2,2} \end{pmatrix}$$

Una forma de optimizar el acceso a los embeddings es: no construir el one-hot vector, sólo tomar la columna de C que corresponde al índice:

$$C(i) = \text{columna } i \text{ de } C$$

Segunda capa oculta

Para poder estimar las probabilidades adecuadamente, se propone una segunda capa oculta que tenga una función de activación $\tanh$:

$$h(i) := \tanh(WC(i) + b) \tag{27}$$

Donde $W \in \mathbb{R}^{m \times d}$ y $b \in \mathbb{R}^m$, con m el número de unidades ocultas.

Esta segunda capa oculta asegura que la red sea un **aproximador universal**, pues la capa de embedding no cuenta con una activación (es lineal).

Capa de salida

Finalmente, dada la entrada $x(i)$, la capa de salida tiene una activación $a(i)$ (i índica cuál es la palabra de entrada):

$$a(i) := Uh(i) + c \tag{28}$$

Tal que $U \in \mathbb{R}^{N \times m}$ y $c \in \mathbb{R}^N$. Por tanto $a(i)$ es un vector, cuyas entradas $a_k(i)$ indexan a la palabra subsiguiente w_k. Así, la activación es:

$$p(w_j|w_i) := \frac{e^{a_j(i)}}{\sum_{k=1}^{K} e^{a_k(i)}} \tag{29}$$

$a_j(i)$ son las neuronas de salida de la red dada la entrada w_i.

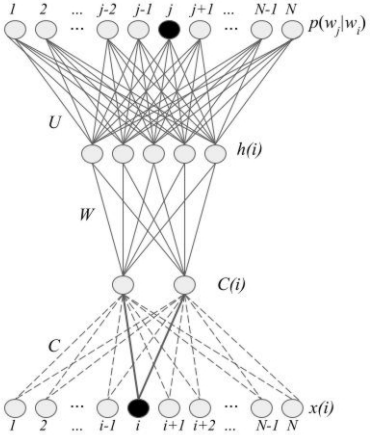
Arquitectura: Forward

Entrada: One-hot $x(i)$ representando w_i.

Capa de embedding: $C(i) = Cx(i)$, $C \in \mathbb{R}^{d \times N}$, d hiperparámetro.

Capa oculta: $h(i) = \tanh(WC(i) + b)$, $W \in \mathbb{R}^{m \times d}$, $b \in \mathbb{R}^m$, m hiperparámetro.

Salida: Pre-activación: $a(i) = Uh(i) + c$, $U \in \mathbb{R}^{N \times m}$, $c \in \mathbb{R}^N$.
Activación:
$p(w_j|w_i) = Softmax\big(a_j(i)\big)$



Aprendizaje del modelo del lenguaje neuronal

Para entrenar esta red neuronal, se requieren obtener los *n*-gramas del texto. Si se toman los bigramas, se puede construir el conjunto supervisado:

$$\mathcal{S} = \{(w_i, w_j) : (w_i, w_j) \text{ es un bigrama}, w_i, w_j \in \Sigma\}$$

Concretamente, a partir de los índices se puede obtener el conjunto:

$$\mathcal{S} = \{(x(i), y_j) : (w_i, w_j) \text{ es un bigrama}, w_i, w_j \in \Sigma\}$$

La idea es dado la entrada i predecir la salida j, como si se tratará de una clase.

Función de riesgo

Definimos la función de riesgo a partir de la entropía cruzada:

$$R(\theta) = -\sum_i \sum_k y_k \ln p(w_k|w_i) \tag{30}$$

donde $y_k = 1$ si $k = j$ tal que w_j es parte del bigrama (w_i, w_j) y será 0 en cualquier otro caso.

Los **bigramas** determinan las **clases**. Este tipo de aprendizaje se conoce como **auto-supervisado** (*self-supervised*).

Backpropagation

Dada la función de riesgo, las variables de backpropagation son:

1. Para la capa de salida:
$$d_{out}(k) = p(w_k|w_i) - \delta_{j=k}$$
2. Para la capa oculta:
$$d_h(k) = \left(1 - h(i)_k^2\right) \sum_q U_{k,q} d_{out}(q)$$
3. Para la capa de embedding:
$$d_C(k) = \sum_q W_{k,q} d_h(q)$$

Algoritmo de aprendizaje modelo del lenguaje

Algorithm Algoritmo de aprendizaje modelo del lenguaje

procedure NEURAL-LM($w_1 ... w_n$, T, η)
  $\mathcal{S} \leftarrow \{(i,j) : (w_i, w_j)$es bigrama$\}$
  Inicializa: $\theta = \{C, W, U, b, c\}$ aleatoriamente.
  for t **from** 1 **to** T **do**
    for (i,j) **in** BATCH($\mathcal{S}$) **do**
      $h(i) = \tanh(WC(i) + b$
      $\mathbf{p_i} = Softmax\Big(Uh(i) + c\Big)$ Forward
      $d_{out}(k) = p(w_k|w_i) - \delta_{y=k}$; $d_h(k) = \left(1 - h(i)_k^2\right) \sum_q U_{k,q} d_{out}(q)$; $d_C(k) = \sum_q W_{k,q} d_h(q)$
      $U_{j,k} \leftarrow U_{j,k} - \eta \cdot d_{out}(k) h(i)_j$; $W_{j,k} \leftarrow W_{j,k} - \eta \cdot d_h(k) C(i)_j$; $C_{j,k} \leftarrow C_{j,k} - \eta \cdot d_C(k) x_j(i)$
    end for
  end for
  return Red con pesos actualizados θ
end procedure

Ejemplo de modelo del lenguaje neuronal

Tómese el corpus:

El niño juega
El niño salta
El perro juega
El perro salta

1) Agregar los símbolos de inicio y de fin:

BOS el niño juega EOS
BOS el niño salta EOS
BOS el perro juega EOS
BOS el perro salta EOS

2) Obtener el vocabulario e indexarlo:

$$\Sigma = \{\text{el}_1, \text{niño}_2, \text{perro}_3, \text{juega}_4, \text{salta}_5, \text{BOS}_6, \text{EOS}_7\}$$

3) Obtener los bigramas del corpus:

$$(\textit{BOS}, \textit{el}), (\textit{el}, \textit{nino}), (\textit{nino}, \textit{juega}), (\textit{juega}, \textit{EOS})$$

$$(\textit{BOS}, \textit{el}), (\textit{el}, \textit{nino}), (\textit{nino}, \textit{salta})...$$

Ejemplo de modelo del lenguaje neuronal

4) Se contruye el conjunto supervisado a partir de los bigramas (w_i, w_j):

$$\mathcal{S} = \{(x(i), y_j)\}$$

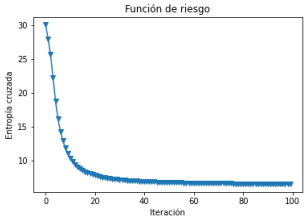
Por ejemplo, para la palabra 'perro' con el índice 3, se tiene el vector:

$$x(3)^T = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Y $y_k = 1$ si $k = 4$ para el bigrama (perro, juega) o si $k = 5$ para el bigrama (perro, salta).

5) Se realiza el entrenamiento. Para esto, se realizan los siguientes pasos:

- Selección de hiperparámetros: $d = 2$, $m = 3$, $\eta = 0.1$, 100 iteraciones.
- Entrenamiento de la red por gradiente descendiente estocástico.



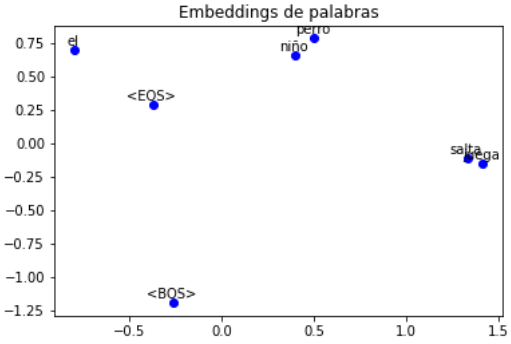
Ejemplo de resultados obtenidos con modelo del lenguaje

Se obtienen **probabilidades de transición**:
Probabilidades iniciales $p(w_k| < BOS >)$ son:

el: 0.9897852001791725
niño: 0.0016858039542186078
salta: 0.0006103737947787955
juega: 0.0004651588886763637
perro: 0.00237470507074468
Para $p(w_k|perro)$ son:

el: 0.0009741799781947631
niño: 0.0038623171456911424
salta: 0.4955997959099393
juega: 0.4851511969451326
perro: 0.00676949789500489
EOS: 0.0076430121260375105

Además el modelo permite obtener **representaciones distribuidas** de las palabras del vocabulario:



Generalización del modelo a n-gramas

Hasta ahora hemos revisado los modelos de bigramas. Pero la propuesta de Bengio (2003) se extiende a *n*-gramas.

Para trabajar con los *n*-gramas, debemos enfocarnos en la capa de embedding. Lo que se hace es un vector **promedio** de los vectores distribuidos de las palabras. Por ejemplo, para trigramas:

$$C(i, i'') = \frac{C(i) + C(i')}{2}$$

Y en general:

$$C(i_1, i_2, ..., i_n) = \frac{C(i_1) + C(i_2) + ... + C(i_n)}{n - 1}$$

Generalización a n-gramas

De esta forma, para un bigrama $w_{i_1}, w_{i_2}, ..., w_{i_n}, w_j$, se tiene el forward:

Capa de embedding: Determinada por:

$$C(i_1, i_2, ..., i_n) = \frac{C(i_1) + C(i_2) + ... + C(i_n)}{n-1}$$

donde $C \in \mathbb{R}^{d \times N}$, donde d es un hiperparámetro.

Capa oculta: Determinada por:

$$h(i_1, i_2, ..., i_n) = \tanh(WC(i_1, i_2, ..., i_n) + b)$$

Con $W \in \mathbb{R}^{m \times d}$ y $b \in \mathbb{R}^m$, donde m es un hiperparámetro.

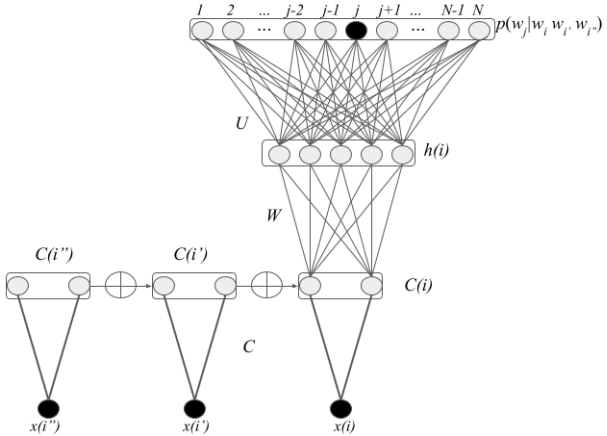
Salida: Determinada por la pre-activación:

$$a(i_1, i_2, ..., i_n) = Uh(i_1, i_2, ..., i_n) + c$$

Con $U \in \mathbb{R}^{N \times m}$ y $c \in \mathbb{R}^N$. La de salida es:

$$p(w_j | w_{i_n}, ..., w_{i_1}) = Softmax[a_j(i_1, i_2, ..., i_n)]$$

Generalización a n-gramas



Textos recomendados

Jurafski, D. y Martin, . (2000). "N-gram Language Model". *Speech and language processing: An introduction to natural language processing*.

Rincón, L. (2007). *Procesos estocásticos*. UNAM.

Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep Learning*, MIT Press. https://www.deeplearningbook.org/

Bengio, Y. (2003). "A Neural Probabilistic Language Model". *Journal of Machine Learning*, pp. 1137-155.