

# Clase 2 - Redes Neuronales completas

## Que es aprender?

### ¿Qué es aprender? ¿Cómo describimos el aprendizaje?

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ Necesitamos definir, de alguna manera, **qué quiere decir aprender**
  - ⇒ Realizamos una **observación**. Inferimos alguna conclusión. La **contrastamos**.
  - ⇒ La próxima vez que nos topemos con una observación similar, queremos la inferencia adecuada
  
- ▶ Sabemos que para definir algo de manera unívoca, necesitamos **usar matemática**
  - ⇒ Una **observación** es un dato que denotamos como **x**
  - ⇒ La **inferencia** la llamamos información y la denotamos con **y**
  - ⇒ El **proceso de inferencia** tiene que ser una función  $f : x \rightarrow y$
  - ⇒ El **contraste** con la realidad lo tenemos que **medir**, con otra función **J**

### Similitudes

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ La próxima vez que nos topemos con una observación similar, queremos la inferencia adecuada
  - ⇒ Determinar qué quiere decir que **dos observaciones sean similares**
  
- ▶ Adoptamos una **concepción probabilística**
  - ⇒ Dos muestras son similares si vienen de la **misma distribución** de probabilidades
$$X \sim f_x(x)$$
    - ⇒ Las observaciones son **realizaciones** de la variable aleatoria
$$X \rightsquigarrow \{x_1, x_2, \dots\}$$
  
- ▶ Si las observaciones futuras **no vienen de la misma distribución**
  - ⇒ No podemos esperar que nuestro aprendizaje funcione
  - ⇒ No existe un único algoritmo de aprendizaje que funcione para todo

Ojo con el ultimo item eh, si las observaciones futuras no vienen de la misma distribucion no podemos esperar que funcionen ok nuestros modelos!

- Tenemos un **conjunto de datos** que provienen de una misma distribución  $\mathbf{X}$
- Tenemos un **algoritmo de aprendizaje**  $f$  que proporciona una inferencia  $y$
- Tenemos una medida de qué tan errónea es la inferencia  $J$
- Con todo esto podemos definir la noción del **riesgo**  $R$  del **algoritmo**  $f$

$$R\{f\} = \mathbb{E}_{\mathbf{X} \sim f_x}[J(f(\mathbf{X}))]$$

- ⇒ Obtenemos la inferencia  $\mathbf{Y} = f(\mathbf{X})$  resultante de aplicar el algoritmo de aprendizaje  $f$
- ⇒ Calculamos cuán errónea es esa inferencia  $J(f(\mathbf{X}))$
- ⇒ Obtenemos el error medio sobre todos los datos posibles a través de su distribución

Como hacemos para conocer la distribución de la VA  $X$ ? → necesitamos usar los datos. Para esto vamos a usar los datos de entrenamiento.

- El **riesgo** es una medida para saber **qué tan bueno es nuestro algoritmo** de aprendizaje
- Podemos, entonces, buscar el algoritmo de aprendizaje que **minimiza el riesgo**

$$\min_f R\{f\} = \min_f \mathbb{E}_{\mathbf{X} \sim f_x}[J(f(\mathbf{X}))]$$

Vamos a tratar de minimizar el riesgo. Cual de todos mis algoritmos es el que minimiza el riesgo? Obtener ese algoritmo va a significar aprender.  
Aprendemos cuando logramos minimizar el riesgo.

El problema es que en la práctica, no hay mucho que podamos hacer con esta ecuación

$$\min_f R\{f\} = \min_f \mathbb{E}_{\mathbf{X} \sim f_x}[J(f(\mathbf{X}))]$$

El otro problema que tenemos es, como resolvemos ese problema de

minimización?

Necesitamos una manera computacional de calcular la esperanza y de minimizar.

## Ley de los Grandes Números

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ No conocemos la distribución  $f_x$  de los datos
- ▶ Pero tenemos acceso a un gran conjunto de datos

$$\mathbf{X} \sim \{\mathbf{x}_1, \dots, \mathbf{x}_M\}$$

⇒ Las muestras  $\{\mathbf{x}_1, \dots, \mathbf{x}_M\}$  son independientes, idénticamente distribuidas (i.i.d.)

- ▶ Si las muestras  $\{\mathbf{x}_1, \dots, \mathbf{x}_M\}$  son i.i.d.
- ▶ Y si la cantidad de muestras  $M$  es lo suficientemente grande

$$\mathbb{E}_{\mathbf{X} \sim f_x} [g(\mathbf{X})] \approx \frac{1}{M} \sum_{m=1}^M g(\mathbf{x}_m)$$

⇒ Este resultado se conoce como la Ley de los Grandes Números

⇒ El promedio se parece a la media si la cantidad de muestras es grande y son i.i.d.

Vamos a querer estimar la esperanza con el promedio de los datos observados, la Ley de los Grandes Números nos dicen que si tengo suficientes datos, se van a parecer.

Por lo tanto, vamos a minimizar el riesgo empírico:

## Minimización del Riesgo Empírico

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ Tenemos acceso a una gran cantidad de datos  $\{\mathbf{x}_1, \dots, \mathbf{x}_M\}$
- ▶ Asumimos que son i.i.d. y usamos la Ley de los Grandes Números
- ▶ Proponemos, entonces, resolver el problema de minimización del riesgo empírico

### Minimización del Riesgo Empírico

$$\min_f \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m))$$

- ▶ La función  $f$  que encontraremos va a ser un algoritmo de aprendizaje
- ▶ Pero también queremos que funcione en datos no observados ⇒ Generalización
- ⇒ Esto será cierto si los datos no observados tienen la misma distribución

Sorteamos el primer problema que era calcular la esperanza, ya no necesitamos eso, sino que podemos usar el promedio → el riesgo empírico.

En este contexto entonces, aprender pasa a ser minimizar el riesgo empírico. Esta justificación matemática estadística nos permite decir que si nos sentamos a escribir el código es probable que generalice bien debido a que el

promedio es un muy buen estimador de la esperanza

Definicion de funcional: Un funcional le asigna una función a un numero!

## Optimización Funcional

Redes Neuronales Completas  
fgama@fi.uba.ar

- El problema de **minimización del riesgo empírico** es el siguiente

$$\min_f \hat{R}(f) = \min_f \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m))$$

- En este contexto, el riesgo empírico es una función  $\hat{R}(f)$  de una función  $f$   
⇒ En matemática, esto se conoce como un funcional (asigna una función a un número)

- Encontrar la *función* que minimiza otra función es un **problema difícil en extremo**  
⇒ ¿Cómo diseñar un algoritmo que actúe sobre funcionales? (Álgebra y Análisis Matemático)  
⇒ Optimización funcional ⇒ Cálculo de variaciones, métodos de kernels, etc.
- Es mucho más fácil encontrar **vectores** (o matrices, o tensores) que minimizan una determinada función  
⇒ Transformar un problema de optimización funcional en uno de **optimización paramétrica**

Nos movemos a un problema de optimización paramétrica, es mas fácil. Buscamos los parámetros que logran minimizar la función. De esta forma podemos "explorar" el espacio de números en vez de las funciones. Recordemos que estamos buscando la *función* que minimiza el numero.

El subconjunto de funciones que vamos a querer minimizar va a depender del problema. En nuestro caso van a ser redes neuronales, es una buena pregunta preguntarnos porque no usar en cambio regresiones o clasificadores? Veremos que hay un par de teoremas que justifican porque usar redes es una buena idea.

Lo que vamos a hacer es elegir una flia paramétrica, es decir una función  $f$  que depende de una serie de parámetros  $\Theta$ :

- Elegir una **familia paramétrica**  $\Rightarrow$  La función  $f$  depende de ciertos parámetros  $\theta$

$$f(\mathbf{x}) = f(\mathbf{x}; \theta) \quad \text{vector}$$

$\Rightarrow$  Ahora el riesgo empírico es una función de un vector  $\hat{R}(f) = \hat{R}(\theta) \Rightarrow$  Una función

$\Rightarrow$  Y pensar en un algoritmo que opere sobre vectores es mucho más factible

- Ciertamente, ahora no se minimiza sobre *todas* las funciones posibles

$\Rightarrow$  Se minimiza sobre todos los **parámetros** que caracterizan a la función  $f$

$$\min_{\theta \in \mathbb{R}^D} \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m; \theta))$$

$\Rightarrow$  Pero ahora es un problema que se puede resolver una forma mucho más fácil

Paso así a aprender vectores en vez de funciones a través de una parametrización.

Y que parametrización vamos a elegir? Este es el problema fundamental en Deep Learning! -> NN, RNN, CNN, etc. Cada una de estas parametrizaciones explotan una u otra de las propiedades de mis datos.

Eligiendo una parametrización lineal, lo mas fácil q podemos plantear a nivel matemático, nuestro problema se convierte en encontrar una matriz  $W$  y un vector  $b$

### Ejemplo: Parametrización Lineal

- Ahora la pregunta pasa a ser: **¿Qué parametrización elijo?**

$\Rightarrow$  Este es el **problema fundamental** en los problemas de aprendizaje

$\Rightarrow$  Elegir una parametrización adecuada **determina el éxito** del algoritmo de aprendizaje

- La elección más elemental posible es una **parametrización lineal**

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- Ejemplo del episodio 1: Dados datos  $\{(\mathbf{x}_m, y_m)\}$  y función de costo  $J(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$

- Se adopta una **parametrización lineal**  $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$  ( $\mathbf{W} \Rightarrow \mathbf{w}, \mathbf{b} \Rightarrow 0$ )

- Resulta **fácil** encontrar cuál es la **solución** al problema de minimización del riesgo empírico

$$\min_{\mathbf{w} \in \mathbb{R}^N} \frac{1}{M} \sum_{m=1}^M (y_m - \mathbf{w}^\top \mathbf{x}_m)^2 \quad \Rightarrow \quad \mathbf{w}^* = \left[ \sum_{m=1}^M \mathbf{x}_m \mathbf{x}_m^\top \right]^{-1} \left[ \sum_{m=1}^M y_m \mathbf{x}_m \right]$$

- En vez de resolver el problema sobre todas las posibles funciones  $f$

$$\min_{\mathbf{w} \in \mathbb{R}^N} \frac{1}{M} \sum_{m=1}^M (y_m - \mathbf{w}^\top \mathbf{x}_m) \Rightarrow \mathbf{w}^* = \left[ \sum_{m=1}^M \mathbf{x}_m \mathbf{x}_m^\top \right]^{-1} \left[ \sum_{m=1}^M y_m \mathbf{x}_m \right]$$

⇒ Lo resolvemos sobre todas las posibles funciones lineales  $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$

- ¿Es esta una buena parametrización para el problema en cuestión?
  - ⇒ Lo va a ser si es cierto que la relación entre  $\mathbf{x}$  e  $\mathbf{y}$  es lineal  $\mathbf{Y} = \mathbf{w}^\top \mathbf{X}$
- Las parametrizaciones lineales tienen varias ventajas
  - ⇒ Trazabilidad matemática ⇒ Otorga garantías de funcionamiento
  - ⇒ Ofrece expresiones ‘cerradas’ (que no siempre son buenas para la computación)



Sin embargo, queremos ir un poco mas allá, las funciones lineales tienen sus limitaciones

- Muchas veces, asumimos que un modelo lineal es suficiente ⇒ Pero a veces realmente no alcanza
- Consideremos el ejemplo de querer aprender una XOR ⇒ Una función binaria  $f : \{0, 1\}^2 \rightarrow \{0, 1\}$

$(x_1, x_2)$	$f(x_1, x_2)$
(0, 0)	0
(0, 1)	1
(1, 0)	1
(1, 1)	0

- Asumimos que tenemos acceso a las cuatro muestras  $\{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0)\}$
- Proponemos una parametrización lineal para la función  $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$  con  $\mathbf{w} \in \mathbb{R}^2$  y  $b \in \mathbb{R}$
- Adoptamos un costo cuadrático  $J(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$

- Resolvemos el problema de minimización del riesgo empírico para la familia paramétrica

$$\min_{\mathbf{w} \in \mathbb{R}^2, b \in \mathbb{R}} \frac{1}{4} \sum_{m=1}^4 (y_m - \mathbf{w}^T \mathbf{x}_m - b)^2$$

- La solución viene dada por  $\mathbf{w} = \mathbf{0}$  y  $b = 1/2$  de tal forma que  $f(\mathbf{x}) = 0,5$

$(x_1, x_2)$	$f(x_1, x_2)$
(0, 0)	0,5
(0, 1)	0,5
(1, 0)	0,5
(1, 1)	0,5

- Hasta en ejemplos fáciles, mirar sólo algoritmos lineales puede ser un problema
  - ⇒ Resigno la trazabilidad matemática (y en muchos casos las garantías)
  - ⇒ Mejoramos en el desempeño empírico de los algoritmos

Si bien el modelo lineal tiene una garantía matemática, como vimos la clase pasada puedo calcular el W de forma matemática, no tiene la potencia de calcular cosas básicas como por ejemplo una XOR. Ese es el tradeoff que elige hacer el ML, decide ir a funciones mas complejas que no necesariamente tienen garantías matemáticas pero que SI pueden aprender funciones mas complejas.

Así llegamos al **Perceptrón**:

Como vimos en la clase práctica pasada.

- ¿Cuál es la forma más sencilla de conseguir un algoritmo no-lineal?
  - ⇒ Le agrego una función no lineal a la salida del algoritmo lineal

### Perceptrón

$$f(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- ⇒  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  es una función no lineal aplicada a cada elemento del vector
- ⇒ La elección de  $\sigma$  es muy importante y es área activa de investigación
- Por razones neurobiológicas esta ecuación se conoce como un **perceptrón**

Si bien el perceptrón es una función NO lineal, al construirlo a partir de una función lineal, nos da cierta familiaridad.

Volviendo ahora a calcular la XOR con nuestro modelo perceptrón:

### Perceptrón Aprende XOR

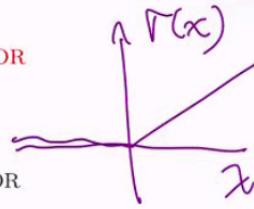
Redes Neuronales Completas  
fgama@fi.uba.ar

- El agregado de una función no lineal puntual alcanza para aprender la función XOR

$$f(\mathbf{x}; \Theta) = \mathbf{w}_2^T \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + b_2 \quad , \quad \Theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{w}_2, b_2\}$$

$\Rightarrow \sigma(x) = \max\{0, x\}$  es una ReLU (*rectified linear unit*),  $\mathbf{w}_2 \in \mathbb{R}^2$ ,  $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$\Rightarrow$  Un perceptrón seguido de una función lineal es suficiente para aprender la XOR



Buscamos obtener los  $\mathbf{W}$  y  $\mathbf{b}$  que mima a nuestra función, de la misma forma que en la clase 1, minimizando con la derivada igualada a cero, etc etc

- Los parámetros para que  $f(\mathbf{x}; \Theta)$  sea igual a la XOR son

$$\mathbf{W}_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b_2 = 0$$

### Perceptrón Aprende XOR

Redes Neuronales Completas  
fgama@fi.uba.ar

- El agregado de una función no lineal puntual alcanza para aprender la función XOR

$$f(\mathbf{x}; \Theta) = \mathbf{w}_2^T \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + b_2 \quad , \quad \Theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{w}_2, b_2\}$$

$\Rightarrow \sigma(x) = \max\{0, x\}$  es una ReLU (*rectified linear unit*),  $\mathbf{w}_2 \in \mathbb{R}^2$ ,  $\mathbf{W} \in \mathbb{R}^{2 \times 2}$

$\Rightarrow$  Un perceptrón seguido de una función lineal es suficiente para aprender la XOR

- Los parámetros para que  $f(\mathbf{x}; \Theta)$  sea igual a la XOR son

$$\mathbf{W}_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b_2 = 0$$

- Comprobamos que efectivamente lo aprende

$$\begin{aligned} [1 & -2] \sigma \left( \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right) &= [1 & -2] \sigma \left( \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right) \\ &= [1 & -2] \begin{bmatrix} \max\{0, 0\} & \max\{0, 1\} & \max\{0, 1\} & \max\{0, 2\} \\ \max\{0, -1\} & \max\{0, 0\} & \max\{0, 0\} & \max\{0, 1\} \end{bmatrix} = [1 & -2] \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

$\Rightarrow$  Y la última multiplicación de matrices arroja  $[0 \ 1 \ 1 \ 0]$  que es lo que queríamos

# La no linealidad (la función de activación) se aplica siempre de forma elemento a elemento.

Ojo, todavía no dijimos *como* el perceptron aprende los pesos que minimizan el modelo. Lo vamos a ver al final de la clase.

Ver como que con agregarle algo tan simple como una ReLu, ya podemos aprender la XOR.

- Un perceptrón seguido de una función lineal es capaz de aprender una XOR
- Una red neuronal se define como una cascada de perceptrones

## Perceptrón Multicapa

$$f(\mathbf{x}; \Theta) = \mathbf{x}_L \quad \text{con} \quad \mathbf{x}_\ell = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell), \quad \ell = 1, \dots, L \quad \text{y} \quad \mathbf{x}_0 = \mathbf{x}$$

Las vamos a definir como una cascada de perceptrones. Es también una parametrización

- Un perceptrón seguido de una función lineal es capaz de aprender una XOR
- Una red neuronal se define como una cascada de perceptrones

## Perceptrón Multicapa

$$f(\mathbf{x}; \Theta) = \mathbf{x}_L \quad \text{con} \quad \mathbf{x}_\ell = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell), \quad \ell = 1, \dots, L \quad \text{y} \quad \mathbf{x}_0 = \mathbf{x}$$

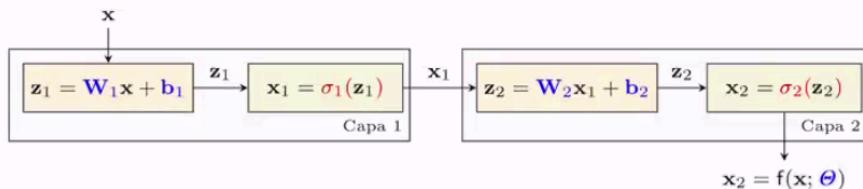
- El perceptrón multicapa es una **parametrización**  $\Rightarrow$  Aprendemos  $\Theta = \{(\mathbf{W}_\ell, \mathbf{b}_\ell)\}_{\ell=1, \dots, L}$

- Un perceptrón seguido de una función lineal es capaz de aprender una XOR
- Una red neuronal se define como una cascada de perceptrones

## Perceptrón Multicapa

$$f(\mathbf{x}; \Theta) = \mathbf{x}_L \quad \text{con} \quad \mathbf{x}_\ell = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell), \quad \ell = 1, \dots, L \quad \text{y} \quad \mathbf{x}_0 = \mathbf{x}$$

- El perceptrón multicapa es una **parametrización**  $\Rightarrow$  Aprendemos  $\Theta = \{(\mathbf{W}_\ell, \mathbf{b}_\ell)\}_{\ell=1, \dots, L}$



Si bien la función de activación capa a capa suele ser la misma en las aplicaciones actuales, nada impide que cambie capa a capa si nosotros

queremos.

**Perceptrón Multicapa**

Redes Neuronales Completas  
fgama@fi.uba.ar

Perceptrón Multicapa

$$f(\mathbf{x}; \Theta) = \mathbf{x}_L \quad \text{con} \quad \mathbf{x}_\ell = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_{\ell-1} + \mathbf{b}_\ell), \quad \ell = 1, \dots, L \quad \text{y} \quad \mathbf{x}_0 = \mathbf{x}$$

- ▶  $L$  es la cantidad de **capas** del perceptrón  $\Rightarrow$  Se elige por diseño  $\Rightarrow$  **Profundidad**
- ▶  $\mathbf{x}_\ell \in \mathbb{R}^{N_\ell}$  es la salida de la capa  $\ell$ , los parámetros son  $\mathbf{W}_\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$  y  $\mathbf{b}_\ell \in \mathbb{R}^{N_\ell}$
- ▶  $N_\ell$  es la dimensión del vector de salida (**hidden units**)  $\Rightarrow$  Se elige por diseño  $\Rightarrow$  **Ancho**
- ▶  $\sigma_\ell$  es una no-linealidad puntual conocida como **función de activación**

.UBAfiuba FACULTAD DE INGENIERIA

20/63

- ▶ El perceptrón multicapa (*multilayer perceptron*; MLP por sus siglas en inglés) recibe muchos nombres
  - $\Rightarrow$  Redes neuronales *feedforward* porque la información sólo fluye hacia adelante
  - $\Rightarrow$  Redes neuronales completas (*fully connected*) porque las matrices  $\mathbf{W}_\ell$  pueden tomar cualquier valor
- ▶ El término red neuronal surge por los primeros modelos matemáticos de la actividad neuronal

Notar como la elección de la función de activación resultó ser clave, suele ser pericia del diseñador, no es fácil elegirlas. Fernando nos va a dar un par de reglitas como para guiarnos pero es mucho prueba y error

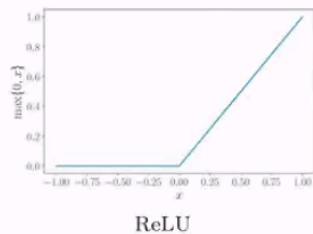
- La inclusión de una función de activación no lineal  $\sigma_\ell$  es clave  $\Rightarrow$  ¿Cómo elegir  $\sigma_\ell$ ?
  - $\Rightarrow$  Hay muchos tipos de activations disponibles  $\Rightarrow$  Es una elección del diseñador
  - $\Rightarrow$  Suelo ser difícil determinar cuál es la apropiada
  - $\Rightarrow$  Conocer cuáles son las opciones y sus características principales ayuda
  - $\Rightarrow$  Saber de antemano cuál es la mejor de todas es prácticamente imposible
  - $\Rightarrow$  Se eligen por prueba y error observando el desempeño del algoritmo en las muestras de prueba
- Vamos a presentar varias de las opciones y discutir sus características principales

## > Funciones de activación

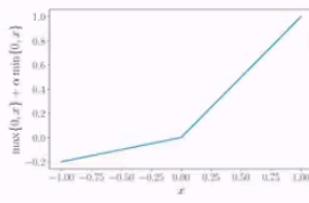
### - ReLU

Suele ser el principal por default debido a que por su comportamiento similar linear es muy fácil de optimizar. Existen algunas varianzas como las leaky relus (donde ademas tenemos que elegir el valor de alfa) y las valor absoluto  
 Notar que cuando vamos a programar NN vamos a tener que ir iterando y probando muchas cosas, vamos a necesitar manejarlos en un entorno donde sea rápido ir probando cosas porque si no no terminamos mas. Por eso es clave pytorch por ej, nos resuelve todo el resto para que podamos probar piola

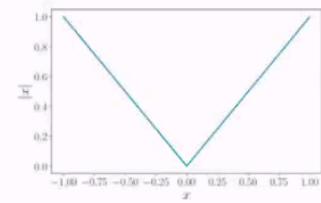
- $\text{ReLU}(x) = \max\{0, x\}$ 
  - $\Rightarrow$  Elección por default  $\Rightarrow$  Fáciles de optimizar cuando el comportamiento es casi lineal
  - $\Rightarrow$  La unidad se considera *activa* cuando está en los positivos
  - $\Rightarrow$  Al inicializar  $b_\ell$  es bueno que sean positivos para asegurarse que todas las unidades están activas
- Limitaciones: Las ReLU no pueden aprender cuando están inactivas
- Extensiones:  $\text{LeakyReLU}(x) = \max\{0, x\} + \alpha \min\{0, x\}$  para algún valor  $\alpha$  a elección, valor absoluto  $|x|$



ReLU



LeakyReLU



Valor absoluto

### - Sigmoid o Tan hyperbolic

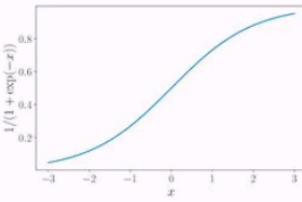
Tienen un valor de saturación, a diferencia de las relus que no limitan el valor que podemos observar en la salida y se pueden ir al infinito, estas acotan el valor en la salida.

El libro dice que no se usan mas porque tienen problemas al aprendizaje, sin embargo Fernando dice que para regresión el suele usar TanHyp y para clasificación usa ReLus. Sin embargo notar que SI es mas difícil de aprender, son mas costosas de calcular ademas a nivel computacional, la tanhyp pero que si tenemos los suficientes datos no deberíamos tener problema.

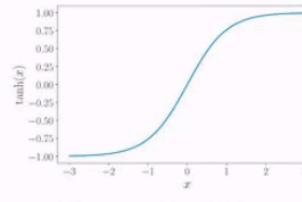
**Sigmoideas y Tangente Hiperbólico**

Redes Neuronales Completas  
fgama@fi.uba.ar

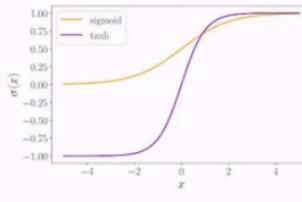
- Sigmoidea  $\sigma(x) = 1/(1 + \exp(-x))$ , tangente hiperbólico  $\sigma(x) = \tanh(x)$
- ⇒ Saturan la entrada ⇒ Es importante cuando sabemos que queremos valores limitados
- ⇒ Cuando saturan, no aprenden ⇒ Son particularmente útiles si los valores están cerca de cero
- ⇒ Se dice que han caído en desuso ⇒ Su utilidad depende de la función de costo J



Sigmoidea



Tangente hiperbólico



Ambas

**.UBAfiuba** FACULTAD DE INGENIERÍA

24 / 63

## - Hyperparametros

Son las decisiones que toma el diseñador, son diferentes de los parámetros de la función. La realidad es que para elegir los mejores valores de los hyperparametros es mucho prueba y error.

- ▶ Un perceptrón multicapa tiene varios **valores que dependen del diseñador**
  - ⇒ La elección de la **función de activación**  $\sigma_\ell$
  - ⇒ La cantidad de **unidades en cada capa**  $N_\ell$
  - ⇒ La **cantidad de capas**  $L$
- ▶ Estos valores se conocen como **hiperparámetros** (los parámetros son los que se aprenden)
- ▶ Los mejores valores de los hiperparámetros **se encuentran mediante prueba y error**
  - ⇒ Existen métodos para automatizar la búsqueda de hiperparámetros (pero es costoso)

## -> Porque Redes Neuronales?

- ▶ El objetivo de una red neuronal es **aproximar una función  $f(\mathbf{x}; \Theta) \approx f(\mathbf{x})$**
- ▶ Sabemos que un algoritmo lineal sirve para describir modelos lineales
  - ⇒ Son fáciles de aprender y tienen trazabilidad matemática para dar garantías
  - ⇒ Muchas veces los modelos lineales son suficientes ⇒ Pero muchas veces, no
- ▶ Por lo que, en general, queremos algoritmos de aprendizaje para modelos más complejos
- ▶ Si un algoritmo lineal aprende modelos lineales, **¿qué clase de modelos aprende una red neuronal?**

### Teorema Universal de Aproximación (Cybenko, 1989; Hornik, 1991)

Sea  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  una función continua, no-constante, acotada y monótonamente creciente.  
 Sea  $f \in C([0, 1]^N)$  una función continua  $f : [0, 1]^N \rightarrow \mathbb{R}$ .

Dadas una función  $f \in C([0, 1]^N)$  y un  $\varepsilon > 0$ , entonces existen: un entero  $N_1$  y matrices  $\mathbf{W}_1 \in \mathbb{R}^{N_1 \times N}$ , y vectores  $\mathbf{b}_1, \mathbf{w}_2 \in \mathbb{R}^{N_1}$ , tal que la red neuronal

$$f(\mathbf{x}; \Theta) = \mathbf{w}_2^T \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_2)$$

con  $\Theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{w}_2\}$  satisface

$$|f(\mathbf{x}) - f(\mathbf{x}; \Theta)| \leq \varepsilon$$

para todo  $\mathbf{x} \in \mathbb{R}^N$ .

*Si yo tengo una función continua en un cubo de tamaño n, lo que yo puedo saber es que una nn de dos capas puede aproximar el valor de la función hasta el nivel epsilon*

"Tan arbitrario como quiera". Obvio que hay un costo que pagar:

## Implicancias y Extensiones del Teorema

### Teorema Universal de Aproximación (Cybenko, 1989; Hornik, 1991)

$$|f(\mathbf{x}) - \mathbf{w}_2^T \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_2)| \leq \varepsilon$$

- Una red neuronal con una sola capa es suficiente para aproximar  $f(\mathbf{x})$   
 ⇒ Siempre y cuando sea lo suficientemente ancha ⇒  $N_1$  tiene que ser grande
- Podemos aprender funciones no lineales con una red neuronal de una capa  
 ⇒ No es necesario diseñar algoritmos complejos específicos para cada tarea

Este epsilon depende del ancho de mi red neuronal. No hay ningún resultado que nos diga que TAN ancho tiene que ser, por lo que podríamos irnos hasta un ancho infinito a nivel teórico.

*Podemos traducir coloquialmente el teorema en: "las redes neuronales son repiolas y pueden aprender cualquier cosa"*

- ▶ El teorema es válido para funciones definidas en cualquier subconjunto compacto de  $\mathbb{R}^N$
- ▶ El teorema también es válido para funciones que mapean entre espacios discretos
- ▶ Las derivadas de la red neuronal también aproximan las derivadas de la función
- ▶ La formulación original del teorema requiere que  $\sigma$  sea tanh o similar
  - ⇒ Se pueden probar resultados similares para  $\sigma(x) = \text{ReLU}(x)$  y sus extensiones

- ▶ El teorema dice que podemos aprender cualquier modelo con una red neuronal lo suficientemente ancha
  - ⇒ No nos aclara qué tan ancha tiene que ser (puede ser necesario una cant. exponencial)
- ▶ Tampoco nos garantiza que el entrenamiento va a ser suficiente para aprender el modelo
- ▶ Incluso si la red neuronal puede representar la función, el aprendizaje puede fallar
  - ⇒ El algoritmo de optimización puede no encontrar los valores correctos
  - ⇒ Si las muestras de entrenamiento no son representativas, podemos incurrir en *overfitting*
- ▶ Nada es Gratis ⇒ No hay un procedimiento universal de aprendizaje que permita generalizar

Como no se que tan ancho tiene que ser, es un resultado teórico y no tan práctico. Como que no puedo ir tan ancho, se ha ido mas al lado de profundidad en vez de ancho. Como que sumando todos los parámetros a lo ancho estoy aumentando ese epsilon teórico. Ponele.

## > Como encuentro los parámetros de esa función? Viene el problema de optimizar la NN

### Minimización del Riesgo Empírico

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ Aprendizaje ⇒ Encontrar el algoritmo  $f : \mathbb{R}^N \rightarrow \mathbb{R}^{N'}$  que minimice el riesgo  $R(f)$ 

$$\min_f \mathbb{E}_{\mathbf{X} \sim f_x} [J(f(\mathbf{X}))]$$
- ▶ No conocemos  $\mathbf{X} \sim f_x$  la distribución de los datos
  - ⇒ Tenemos acceso a una muestra  $\mathbf{X} \sim \{\mathbf{x}_1, \dots, \mathbf{x}_M\}$  ⇒ Minimización del Riesgo Empírico
 
$$\min_f \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m))$$
- ▶ Minimizar sobre funcionales es sumamente difícil ⇒ Parametrización
 
$$\min_{\theta} \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m; \theta))$$
- ▶ ¿Cómo resolvemos el problema de minimización?

- ▶ ¿Cómo resolvemos el problema de minimización?

- ▶ En el caso de aprender la XOR lo hicimos “a ojo”, probando con criterio
- ▶ En el caso de regresión lineal con costo cuadrático encontramos el punto crítico
  - ⇒ Tomamos la derivada de la función, la igualamos a cero, y resolvimos

- ▶ Y en un caso general, ¿cómo hacemos?

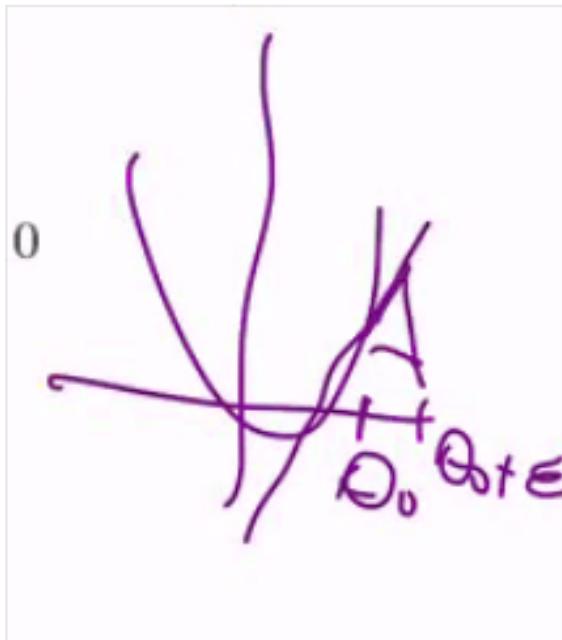
¿Se puede automatizar el problema de optimización?

## – El algoritmo del Descenso del Gradiente

Quiero resolver de forma numérica e iterativa. Arranco con unos valores random de los parámetros y me voy a ir acercando.

- ▶ Usemos lo que sabemos del análisis matemático para diseñar un método automático
- ▶ Queremos minimizar la función  $\hat{R}(\theta) = \frac{1}{M} \sum_{m=1}^M J(f(x_m; \theta))$  numéricamente y automáticamente
  - ⇒ Supongamos que lo queremos hacer de manera iterativa  $\theta_{k+1} \leftarrow \text{algoritmo}(\theta_k)$
  - ⇒ Empezamos con un punto  $\theta_0 \Rightarrow$  ¿A dónde nos movemos luego?
- ▶ Un poco de inspiración: En una dimensión  $\hat{R}(\theta + \varepsilon) \approx \hat{R}(\theta) + \varepsilon \hat{R}'(\theta)$  para  $\varepsilon \approx 0, \varepsilon > 0$ 
  - ⇒ Si yo muevo el valor de  $\theta$  a  $\theta + \varepsilon$ , la función se va a mover en  $\varepsilon \hat{R}'(\theta)$
  - ⇒ Es evidente que si  $\hat{R}'(\theta) < 0$  la función va a ser más chica en  $\theta + \varepsilon$
  - ⇒ Pero si  $\hat{R}'(\theta) > 0$  la función va a ser más grande en  $\theta + \varepsilon$
- ▶ ¿Cómo podemos elegir  $\varepsilon$ , entonces? Si  $\varepsilon = -\eta \hat{R}'(\theta)$  obtenemos que  $\hat{R}(\theta + \varepsilon) \approx \hat{R}(\theta) - \eta (\hat{R}'(\theta))^2$ 
  - ⇒ Es decir, la función siempre es más chica ( $\eta$  es tal que  $\eta \hat{R}'(\theta) = \varepsilon \ll 0$ )

$\theta_{k+1} \leftarrow \theta_k - \eta \hat{R}'(\theta_k)$



cuando me muevo un epsilon me estoy moviendo por el plano tangente. Si es positiva la derivada me voy para arriba y si es negativa para abajo. El siguiente paso es independizarme de esto, si me muevo en dirección **opuesta** a la derivada, siempre estoy achicando mi función.

Que pasa cuando me voy a mayores dimensiones, tengo que empezar a mirar las derivadas direccionales:

### Optimización de Gradiente Descendente

Redes Neuronales Completas  
fgama@fi.uba.ar

- La derivada contiene información de los lugares donde la función (de)crece

- Para  $\hat{R} : \mathbb{R}^M \rightarrow \mathbb{R}$  una función de múltiples dimensiones  $\Rightarrow$  Gradiente  $\nabla_{\theta}\hat{R}$

$$\nabla_{\theta}\hat{R} = \left[ \frac{\partial \hat{R}}{\partial [\theta]_1} \quad \frac{\partial \hat{R}}{\partial [\theta]_2} \quad \dots \quad \frac{\partial \hat{R}}{\partial [\theta]_M} \right] \in \mathbb{R}^M$$

- Derivada direccional  $\Rightarrow$  Pendiente de la función  $\hat{R}$  en la dirección  $\check{u}$

$$\frac{\partial}{\partial \alpha} \{ \hat{R}(\theta + \alpha \check{u}) \} \Big|_{\alpha=0} = \check{u}^T \nabla_{\theta} \hat{R}$$

$\Rightarrow$  La contribución a la pendiente en la dirección  $\check{u}$  es  $\langle \check{u}, \nabla_{\theta} \hat{R} \rangle$

- ¿Cuál es la dirección donde la función decrece lo más posible?

$\Rightarrow \langle \check{u}, \nabla_{\theta} \hat{R} \rangle$  es un producto interno  $\Rightarrow$  El mínimo es cuando  $\check{u}$  va en la dirección de  $-\nabla_{\theta} \hat{R}$

#### Gradiente Descendente

$$\theta_{k+1} \leftarrow \theta_k - \eta \nabla_{\theta} \hat{R}(\theta_k)$$

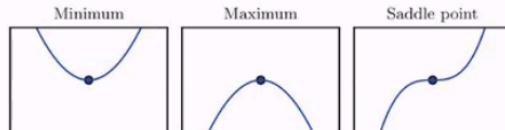
La derivada, lo q nos vamos a mover en la dirección del parámetro k, va a ser proporcional a la componente k del valor del gradiente. Nos movemos en sentido negativo.

"El siguiente valor del parámetro iterativo va a tener que ser en la dirección opuesta que da el gradiente"

### Gradiente Descendente

$$\theta_{k+1} \leftarrow \theta_k - \eta \nabla_{\theta} \hat{R}(\theta_k)$$

- ▶ El valor de  $\eta$  se conoce como **tasa de aprendizaje** (*learning rate*)  $\Rightarrow$  Determina el **tamaño del salto**  
 $\Rightarrow$  Es un parámetro **fundamental** para que la optimización de gradiente descendente sea exitosa
- ▶ ¿Qué pasa cuando  $\nabla_{\theta} \hat{R}(\theta_k) = 0$ ? El algoritmo no cambia  $\theta_{k+1} \leftarrow \theta_k$   
 $\Rightarrow$  Los puntos donde  $\nabla_{\theta} \hat{R}(\theta) = 0$  se conocen como **puntos críticos**  
 $\Rightarrow$  Pueden ser mínimos, máximos o puntos de ensilladura



El valor de eta es el learning rate y determinar que tanto me muevo. Es un hyperparametro absolutamente fundamental para los algoritmos de ML. Si un algoritmo de ML nos da cualquiera cosa, siempre empecemos por tocar el Learning rate.

Sobre **mínimos locales**: Que pasa si el gradiente es cero? Deje de moverme. Podria ocurrir esto en un mínimo o en un mínimo local o bien un punto de ensilladura. La realidad es que nosotros vemos la progresión de los valores, entonces vemos si los valores estan subiendo o bajando. En un para de slides mas vamos a ver porque en la practica no nos interesa tener un mínimo local, si bien en principio si podria ser un gran problema.

> Lo que vamos a tener que hacer nosotros sera entonces calcular el **gradiente del riesgo**:

### Gradiente Descendente para un Perceptrón Multicapa

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ Es fundamental **calcular el gradiente del riesgo**, aunque sea numéricamente

$$\hat{R}(\theta) = \frac{1}{M} \sum_{m=1}^M J(f(x_m; \theta))$$

- ▶ Para nosotros,  $f(x_m; \theta)$  tiene una forma muy particular  $\Rightarrow$  **Perceptrón multicapa**  
 $\Rightarrow$  ¿Cómo queda la derivada cuando  $f(x_m; \theta) = x_L$  para  $x_\ell = \sigma(W_\ell x_{\ell-1} + b_\ell)$ ?
- ▶ En el caso del perceptrón multicapa,  $\Theta = \{(W_1, b_1), \dots, (W_L, b_L)\}$
- ▶ ¿Cómo calculamos la derivada?  $\Rightarrow$  Usando la **regla de la cadena**

multicapa.

Lo que vamos a tener que hacer para encontrar el gradiente de  $R$  va a ser usar la **regla de la cadena**,

### Gradiente Descendente para un Perceptrón

Redes Neuronales Completas  
fgama@fi.uba.ar

- Empecemos fácil  $\Rightarrow$  Una sola capa  $\Rightarrow$  Calculemos el gradiente para un perceptrón

$$f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1) = \sigma(\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1)$$

- Recordemos la **regla de la cadena**:  $(f(g(x)))' = f'(y)g'(x)$  con  $y = g(x)$

- Lo función  $\hat{R}(\mathbf{W}_1, \mathbf{b}_1)$  es  $\hat{R} : \mathbb{R}^{N_1 \times N} \times \mathbb{R}^{N_1} \rightarrow \mathbb{R}$ , lo que implica que las derivadas que queremos

$$\frac{\partial \hat{R}}{\partial \mathbf{W}_1} \in \mathbb{R}^{N_1 \times N} : \left[ \frac{\partial \hat{R}}{\partial \mathbf{W}_1} \right]_{ij} = \frac{\partial \hat{R}}{\partial [\mathbf{W}_1]_{ij}} \quad \text{y} \quad \frac{\partial \hat{R}}{\partial \mathbf{b}_1} \in \mathbb{R}^{N_1} : \left[ \frac{\partial \hat{R}}{\partial \mathbf{b}_1} \right]_i = \frac{\partial \hat{R}}{\partial [\mathbf{b}_1]_i}$$

*derivada con respecto a  
una matriz y un vector*

### Gradiente Descendente para un Perceptrón

Redes Neuronales Completas  
fgama@fi.uba.ar

- Empecemos a calcular  $\partial \hat{R} / \partial \mathbf{W}_1$

$$\frac{\partial \hat{R}}{\partial \mathbf{W}_1} = \frac{\partial}{\partial \mathbf{W}_1} \left\{ \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1)) \right\} = \frac{1}{M} \sum_{m=1}^M \frac{\partial}{\partial \mathbf{W}_1} \{ J(f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1)) \}$$

- El paso siguiente es calcular  $\partial J / \partial \mathbf{W}_1 \Rightarrow$  Para eso definimos  $\mathbf{x}_{1m} = f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1) \in \mathbb{R}^{N_1}$

$$\frac{\partial J}{\partial \mathbf{W}_1} = \frac{\partial}{\partial \mathbf{W}_1} \{ J(f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1)) \} = \frac{\partial J}{\partial \mathbf{x}_{1m}} \frac{\partial}{\partial \mathbf{W}_1} \{ f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1) \}$$

$\Rightarrow$  La función  $J(y)$  como función de su argumento va de  $\mathbb{R}^{N_1} \rightarrow \mathbb{R}$ , con lo que  $\partial J / \partial \mathbf{x}_{1m} \in \mathbb{R}^{N_1}$  implica

$$\frac{\partial J}{\partial \mathbf{x}_{1m}} = \frac{\partial J}{\partial \mathbf{y}} \Big|_{\mathbf{y}=\mathbf{x}_{1m}=f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1)} \Rightarrow \left[ \frac{\partial J}{\partial \mathbf{x}_{1m}} \right]_i = \frac{\partial J}{\partial [y]_i} \Big|_{\mathbf{y}=\mathbf{x}_{1m}=f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1)}$$

$\Rightarrow$  Para simplificar la notación, denotamos  $\mathbf{g}_{J'(\mathbf{x}_{1m})} = \partial J / \partial \mathbf{x}_{1m}$  como un vector en  $\mathbb{R}^{N_1}$

- Ahora viene la parte de  $\partial f / \partial \mathbf{W}_1$  donde tenemos que usar que  $f(\mathbf{x}_m; \mathbf{W}_1, \mathbf{b}_1) = \sigma(\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1)$

$$\frac{\partial f}{\partial \mathbf{W}_1} = \frac{\partial}{\partial \mathbf{W}_1} \{\sigma(\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1)\} = \frac{\partial \sigma}{\partial \mathbf{z}_{1m}} \frac{\partial}{\partial \mathbf{W}_1} \{\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1\} \text{ donde } \mathbf{z}_{1m} = \mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1$$

$\Rightarrow$  La función  $\sigma(\mathbf{z})$  como función de su argumento va de  $\mathbb{R}^{N_1} \rightarrow \mathbb{R}^{N_1}$ , con lo que  $\partial \sigma / \partial \mathbf{z}_{1m} \in \mathbb{R}^{N_1 \times N_1}$

$$\frac{\partial \sigma}{\partial \mathbf{z}_{1m}} = \frac{\partial \sigma}{\partial \mathbf{y}} \Big|_{\mathbf{y}=\mathbf{z}_{1m}=\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1} \Rightarrow \left[ \frac{\partial \sigma}{\partial \mathbf{z}_{1m}} \right]_{ij} = \frac{\partial \sigma_i}{\partial [\mathbf{z}_{1m}]_j} = \sigma'([\mathbf{z}_{1m}]_j)$$

$\Rightarrow$  Pero  $\sigma$  es una función puntual, con lo que cada fila de la matriz  $\partial \sigma / \partial \mathbf{z}_{1m}$  es igual

$\Rightarrow$  Para simplificar la notación, denotamos  $\mathbf{G}_{\sigma'(\mathbf{z}_{1m})} = \partial \sigma / \partial \mathbf{z}_{1m}$  una matriz en  $\mathbb{R}^{N_1 \times N_1}$

En un preceptor primero tenemos la derivada con respecto a la función de la activación, que sera la derivada de cada uno de los componentes de la salida. Y finalmente le podemos calcular la derivada de lo que esta adentro

- El último paso:  $\partial \{\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1\} / \partial \mathbf{W}_1 \Rightarrow$  Como función de  $\mathbf{W}_1$ , va de  $\mathbb{R}^{N_1 \times N} \rightarrow \mathbb{R}^{N_1}$

$\Rightarrow$  La derivada es un tensor de tamaño  $N_1 \times N_1 \times N$  (pero es una función lineal)

$$\left[ \frac{\partial}{\partial \mathbf{W}_1} \{\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1\} \right]_{ijk} = \frac{\partial [\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1]_i}{\partial [\mathbf{W}]_{jk}}$$

$\Rightarrow$  La derivada del elemento  $i$  de  $\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1$  respecto del elemento  $(j, k)$  de la matriz  $\mathbf{W}_1$

- Calculamos el elemento  $i$  de  $\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1$  como

$$[\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1]_i = \sum_{k'=1}^N [\mathbf{W}_1]_{ik'} [\mathbf{x}_m]_{k'} + [\mathbf{b}]_i$$

$\Rightarrow$  Primera observación:  $\partial [\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1]_i / \partial [\mathbf{W}]_{jk} = 0$  si  $i \neq j \Rightarrow$  La fila  $j$  de  $\mathbf{W}_1$  no participa

$$\frac{\partial [\mathbf{W}_1 \mathbf{x}_m + \mathbf{b}_1]_i}{\partial [\mathbf{W}]_{ik}} = \frac{\partial}{\partial [\mathbf{W}]_{ik}} \left\{ \sum_{k'=1}^N [\mathbf{W}_1]_{ik'} [\mathbf{x}_m]_{k'} + [\mathbf{b}]_i \right\} = \sum_{k'=1}^N \frac{\partial [\mathbf{W}_1]_{ik'} [\mathbf{x}_m]_{k'}}{\partial [\mathbf{W}]_{ik}}$$

$\Rightarrow$  Esto es no-nulo únicamente cuando  $k = k'$  con lo cual la derivada es  $[\mathbf{x}_m]_k$

- En resumen, tenemos que el tensor de derivadas  $\partial\{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\}/\partial\mathbf{W}_1 \in \mathbb{R}^{N_1 \times N_1 \times N}$  satisface

$$\left[ \frac{\partial}{\partial\mathbf{W}_1} \{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\} \right]_{ijk} = \begin{cases} [\mathbf{x}_m]_k & \text{si } i=j \\ 0 & \text{en caso contrario} \end{cases}$$

- Si pensamos los tensores como un conjunto de  $N_1$  matrices de tamaño  $N_1 \times N$  tenemos algo así

$$\frac{\partial}{\partial\mathbf{W}_1} \{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\} = \left\{ \begin{bmatrix} [\mathbf{x}_m]_1 & [\mathbf{x}_m]_2 & \cdots & [\mathbf{x}_m]_N \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ [\mathbf{x}_m]_1 & [\mathbf{x}_m]_2 & \cdots & [\mathbf{x}_m]_N \end{bmatrix}, \right\}$$

⇒ Se repite el dato en las filas ⇒ Se puede escribir de manera compacta como

$$\frac{\partial}{\partial\mathbf{W}_1} \{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\} = \left\{ \begin{bmatrix} \mathbf{x}_m^T \\ \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix}, \begin{bmatrix} \mathbf{0}^T \\ \mathbf{x}_m^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix}, \dots, \begin{bmatrix} \mathbf{0}^T \\ \vdots \\ \mathbf{x}_m^T \end{bmatrix} \right\}$$

Una vez q me metí adentro del perceptron multicapa, la derivada es siempre la misma una y otra vez.

- Ahora juntamos todo para obtener la derivada

$$\frac{\partial \hat{R}}{\partial \mathbf{W}_1} = \frac{1}{M} \sum_{m=1}^M \underbrace{\mathbf{g}_{j'}^T(\mathbf{x}_{1m})}_{1 \times N_1} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{1m})}^T}_{N_1 \times N_1} \underbrace{\frac{\partial}{\partial \mathbf{W}_1} \{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\}}_{N_1 \times N_1 \times N}$$

- Recordamos multiplicación con tensores: pensarlo como un conjunto de  $N_1$  matrices de tamaño  $N_1 \times N$

⇒ El resultado es  $N_1$  multiplicaciones de  $(1 \times N_1)(N_1 \times N_1)(N_1 \times N) = 1 \times N$

⇒ Da por resultado  $N_1$  vectores de tamaño  $N$  ⇒ Matriz  $N_1 \times N$  ⇒ Lo que estábamos buscando

- Para obtener  $\partial \hat{R} / \partial \mathbf{b}_1$  sólo el último término cambia (función  $\mathbb{R}^{N_1} \rightarrow \mathbb{R}^{N_1}$ )

$$\left[ \frac{\partial}{\partial \mathbf{b}_1} \{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\} \right]_{ij} = \frac{\partial}{\partial [\mathbf{b}_j]} \{[\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1]_i\} = \frac{\partial}{\partial [\mathbf{b}_j]} \left\{ \sum_{k=1}^N [\mathbf{W}_1]_{ik} [\mathbf{x}_m]_k + [\mathbf{b}]_i \right\}$$

⇒ Esto es igual a 1 si  $i = j$  y 0 en otro caso ⇒  $\partial\{\mathbf{W}_1\mathbf{x}_m + \mathbf{b}_1\}/\partial\mathbf{b}_1 = \mathbf{I}_{N_1 \times N_1}$

El haber elegido la parametrización me permite a mí conocer las derivadas y resolver esto de forma automática. Puedo "codear" (ya está hecho) la regla de la cadena y me sirve pa resolver para siempre.

Para calcular el gradiente nos queda una multiplicación del costo en cada capa. Donde los  $w$  son los parámetros que estamos tratando de encontrar, y los  $z$  son las salidas de cada una de mis capas. Aca es donde entra el algoritmo de back propagation, una forma eficiente de poder hacer estas cuentas con la mínima cantidad de pasadas posibles.

- ▶ Juntando todo, ahora tenemos que

$$\frac{\partial \hat{R}}{\partial \mathbf{W}_\ell} = \frac{1}{M} \sum_{m=1}^M \underbrace{\mathbf{g}_{J'(\mathbf{x}_{Lm})}^\top}_{\text{costo}} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}^\top \mathbf{W}_L}_{\text{capa } L} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{(L-1)m})}^\top \mathbf{W}_{L-1}}_{\text{capa } L-1} \cdots \mathbf{G}_{\sigma'(\mathbf{z}_{\ell m})}^\top \frac{\partial}{\partial \mathbf{W}_\ell} \{ \mathbf{W}_\ell \mathbf{x}_{(\ell-1)m} + \mathbf{b}_\ell \}$$

⇒ Acá tenemos que  $\partial \{ \mathbf{W}_\ell \mathbf{x}_{(\ell-1)m} + \mathbf{b}_\ell \} / \partial \mathbf{W}_\ell$  el tensor  $N_\ell \times N_\ell \times N_{\ell-1}$

⇒ Si queremos  $\partial \{ \mathbf{W}_\ell \mathbf{x}_{(\ell-1)m} + \mathbf{b}_\ell \} / \partial \mathbf{W}_\ell$  simplemente reemplazamos el último término por  $\mathbf{I}_{N_\ell \times N_\ell}$

## Implementación Numérica: Backpropagation

Redes Neuronales Completas  
fgama@fi.uba.ar

- ▶ Pero para calcular el gradiente necesitamos

$$\frac{\partial \hat{R}}{\partial \mathbf{W}_{\ell k}} = \frac{1}{M} \sum_{m=1}^M \underbrace{\mathbf{g}_{J'(\mathbf{x}_{Lm})}^\top}_{\text{costo}} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}^\top \mathbf{W}_{Lk}}_{\text{capa } L} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{(L-1)m})}^\top \mathbf{W}_{(L-1)k}}_{\text{capa } L-1} \cdots \mathbf{G}_{\sigma'(\mathbf{z}_{\ell m})}^\top \frac{\partial}{\partial \mathbf{W}_{\ell k}} \{ \mathbf{W}_{\ell k} \mathbf{x}_{(\ell-1)m} + \mathbf{b}_{\ell k} \}$$

- ▶ Se observa que cada capa necesita el valor de la capa anterior

⇒ En la pasada hacia delante, también podemos calcular lo que necesitamos para el gradiente

$$\begin{aligned} \mathbf{x}_m &\rightarrow \left( \mathbf{z}_{1m}, \mathbf{x}_{1m}, \mathbf{G}_{\sigma'(\mathbf{z}_{1m})}, \frac{\partial}{\partial \mathbf{W}_{1k}} \{ \mathbf{W}_{1k} \mathbf{x}_m + \mathbf{b}_{1k} \} \right) \rightarrow \left( \mathbf{z}_{2m}, \mathbf{x}_{2m}, \mathbf{G}_{\sigma'(\mathbf{z}_{2m})}, \frac{\partial}{\partial \mathbf{W}_{2k}} \{ \mathbf{W}_{2k} \mathbf{x}_{1m} + \mathbf{b}_{2k} \} \right) \\ &\rightarrow \cdots \rightarrow \left( \mathbf{z}_{Lm}, \mathbf{x}_{Lm}, \mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}, \frac{\partial}{\partial \mathbf{W}_{Lk}} \{ \mathbf{W}_{Lk} \mathbf{x}_{(L-1)m} + \mathbf{b}_{Lk} \} \right) \rightarrow \left( \mathbf{J}(\mathbf{x}_{Lm}), \mathbf{g}_{J'(\mathbf{x}_{Lm})} \right) \end{aligned}$$

- ▶ Pero para calcular el gradiente necesitamos

$$\frac{\partial \hat{R}}{\partial \mathbf{W}_{\ell k}} = \frac{1}{M} \sum_{m=1}^M \underbrace{\mathbf{g}_{J'(\mathbf{x}_{Lm})}^\top}_{\text{costo}} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}^\top}_{\text{capa } L} \mathbf{W}_{Lk} \underbrace{\mathbf{G}_{\sigma'(\mathbf{z}_{(L-1)m})}^\top}_{\text{capa } L-1} \mathbf{W}_{(L-1)k} \cdots \mathbf{G}_{\sigma'(\mathbf{z}_{\ell m})}^\top \frac{\partial}{\partial \mathbf{W}_{\ell k}} \{ \mathbf{W}_{\ell k} \mathbf{x}_{(\ell-1)m} + \mathbf{b}_{\ell k} \}$$

La idea del bp nob es solo calcular las salidas en cu de las capas sino también las derivadas. A medida que voy avanzando hacia adelante voy guardando ambas cosas. Una vez que llegue hasta l final, puedo ir calculando hacia atrás el gradiente de cada una de las capas.

- Y ahora que calculamos todo lo que necesitamos en la pasada hacia adelante

$$\begin{aligned} \mathbf{x}_m &\rightarrow \left( \mathbf{z}_{1m}, \mathbf{x}_{1m}, \mathbf{G}_{\sigma'(\mathbf{z}_{1m})}, \frac{\partial}{\partial \mathbf{W}_{1k}} \{ \mathbf{W}_{1k} \mathbf{x}_m + \mathbf{b}_{1k} \} \right) \rightarrow \left( \mathbf{z}_{2m}, \mathbf{x}_{2m}, \mathbf{G}_{\sigma'(\mathbf{z}_{2m})}, \frac{\partial}{\partial \mathbf{W}_{2k}} \{ \mathbf{W}_{2k} \mathbf{x}_{1m} + \mathbf{b}_{2k} \} \right) \\ &\rightarrow \dots \rightarrow \left( \mathbf{z}_{Lm}, \mathbf{x}_{Lm}, \mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}, \frac{\partial}{\partial \mathbf{W}_{Lk}} \{ \mathbf{W}_{Lk} \mathbf{x}_{(L-1)m} + \mathbf{b}_{Lk} \} \right) \rightarrow \left( J(\mathbf{x}_{Lm}), \mathbf{g}_{J'(\mathbf{x}_{Lm})} \right) \end{aligned}$$

⇒ Podemos volver hacia atrás, calculando las derivadas, de la última a la primera

$$\begin{aligned} \left[ \frac{\partial \hat{R}}{\partial \mathbf{W}_{Lk}} \right] &= \frac{1}{M} \sum_{m=1}^M \mathbf{g}_{J'(\mathbf{x}_{Lm})}^\top \mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}^\top \frac{\partial}{\partial \mathbf{W}_{Lk}} \{ \mathbf{W}_{Lk} \mathbf{x}_{(L-1)m} + \mathbf{b}_{Lk} \} \\ \left[ \frac{\partial \hat{R}}{\partial \mathbf{W}_{(L-1)k}} \right] &= \frac{1}{M} \sum_{m=1}^M \mathbf{g}_{J'(\mathbf{x}_{Lm})}^\top \mathbf{G}_{\sigma'(\mathbf{z}_{Lm})}^\top \mathbf{W}_{Lk} \mathbf{G}_{\sigma'(\mathbf{z}_{(L-1)m})}^\top \frac{\partial}{\partial \mathbf{W}_{(L-1)k}} \{ \mathbf{W}_{(L-1)k} \mathbf{x}_{(L-2)m} + \mathbf{b}_{(L-1)k} \} \\ &\vdots \end{aligned}$$

- El algoritmo de backpropagation sirve para calcular el gradiente y consiste en dos fases
  - ⇒ ‘Hacia adelante’: calcula la salida y guarda valores relevantes para el gradiente
  - ⇒ ‘Hacia atrás’: multiplica esos valores para calcular el gradiente en cada capa

### Algoritmo de Backpropagation

```
Dados  $\{\mathbf{W}_{\ell k}, \mathbf{b}_{\ell k}\}$  para todo  $\ell$ ; dado  $\mathbf{x}_m$ 
Para cada  $\ell = 1, \dots, L$ 
  Calcular  $\mathbf{z}_{\ell m} = \mathbf{W}_{\ell k} \mathbf{x}_{(\ell-1)m} + \mathbf{b}_{\ell k}$ 
  Calcular  $\mathbf{x}_{\ell m} = \sigma(\mathbf{z}_{\ell m})$ 
  Calcular  $\mathbf{G}_{\sigma'(\mathbf{z}_{\ell m})}$ 
  Calcular  $\partial \mathbf{z}_{\ell m} / \partial \mathbf{W}_{\ell k}$ 
  Calcular  $\partial \mathbf{z}_{\ell m} / \partial \mathbf{b}_{\ell k}$  → para la salida
Para cada  $\ell = L, \dots, 1$ 
  Calcular  $\partial \hat{R} / \partial \mathbf{W}_{\ell k}$ 
  Calcular  $\partial \hat{R} / \partial \mathbf{b}_{\ell k}$  → Para el gradiente
```

- El algoritmo de **gradiente descendente** es sencillo  $\Rightarrow \theta_{k+1} \leftarrow \theta_k - \eta \frac{\partial \hat{R}}{\partial \theta_k}$
- La clave está en calcular el gradiente del riesgo empírico  $\hat{R}(\theta)$

$$\frac{\partial \hat{R}}{\partial \theta} = \frac{\partial}{\partial \theta} \left\{ \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m; \theta)) \right\} = \frac{1}{M} \sum_{m=1}^M \frac{\partial}{\partial \theta} \{ J(f(\mathbf{x}_m; \theta)) \}$$

$\Rightarrow$  Para  $f(\mathbf{x}; \theta)$  un perceptrón multicapa  $\Rightarrow$  Usamos backpropagation para calcular el gradiente

- Ahora bien, el uso de backpropagation parece razonable, pero **abre dos interrogantes**
  - $\Rightarrow$  Teórico  $\Rightarrow$  ¿Hacer gradiente descendente sobre el riesgo empírico  $\hat{R}$  sirve para el riesgo  $R$ ?
  - $\Rightarrow$  Práctico  $\Rightarrow$  Cada pasada del backpropagation se realiza para una sola muestra

- $\text{¿Hacer gradiente descendente sobre el riesgo empírico sirve para minimizar el riesgo?} \rightarrow$ 
  - $\Rightarrow$  Recordemos que el riesgo es  $R(\theta) = \mathbb{E}_{\mathbf{X} \sim f_x}[J(f(\mathbf{X}; \theta))]$   $\Rightarrow$  Su derivada es  $\partial R / \partial \theta = \mathbb{E}_{\mathbf{X} \sim f_x}[\partial J / \partial \theta]$
  - $\Rightarrow$  No tenemos acceso a  $\mathbb{E}_{\mathbf{X} \sim f_x}[\partial J(f(\mathbf{X}; \theta)) / \partial \theta]$ , sólo tenemos acceso a  $\partial J(f(\mathbf{x}_m; \theta)) / \partial \theta$
  - $\Rightarrow$  Ley de los grandes números  $\Rightarrow M^{-1} \sum_{m=1}^M \partial J(f(\mathbf{x}_m; \theta)) / \partial \theta \approx \mathbb{E}_{\mathbf{X} \sim f_x}[\partial J(f(\mathbf{X}; \theta)) / \partial \theta]$ ,  $M \gg 1$
  - $\Rightarrow$  Gradiente Descendente Estocástico (SGD: *stochastic gradient descent*) (muestras, no esperanza)
- Cada pasada del backpropagation se realiza para una sola muestra
  - $\Rightarrow$  Necesito muchas muestras para estimar bien la esperanza
  - $\Rightarrow$  Pero cada muestra demanda una pasada “hacia adelante y hacia atrás” del algoritmo
  - $\Rightarrow$  Cuantas más muestras, más tarde en actualizar los parámetros

La palabra estocástico nace de que no conocemos el gradiente del riesgo posta sino que del riesgo empírico. La diferencia es que el GDE usa MUESTRAS para estimar el gradiente, el gradiente posta no lo podemos conocer porque no podemos conocer la esperanza.

importante: tengo que calcular el gradiente para cada una de mis muestras y para cada una de esas tengo que hacer bp. Cuantas mas muestras tenga, mas tiempo.

Acá aparece un tradeoff que lo vamos a solucionar a través de:

- ▶ Tenemos una muestra de tamaño  $M$ :  $\{\mathbf{x}_1, \dots, \mathbf{x}_M\} \Rightarrow M$  debería ser grande
  - ▶ Tomamos un subconjunto de muestras  $M' \ll M$  de manera aleatoria  $\{\mathbf{x}_{m'}\}_{m'=1}^{M'}$
  - ▶ Hacemos una pasada del backpropagation, sólo sobre las  $M'$  muestras  $\Rightarrow$  Más rápido
  - ▶ El estimador del gradiente ahora es  $M'^{-1} \sum_{m'=1}^{M'} \partial J(\mathbf{f}(\mathbf{x}_{m'}; \boldsymbol{\theta})) / \partial \boldsymbol{\theta} \Rightarrow$  Un estimador un poco peor
  - ▶ Actualizamos:  $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \eta \partial \hat{R} / \partial \boldsymbol{\theta}_k \Rightarrow$  Pero usando  $\partial \hat{R} / \partial \boldsymbol{\theta}_k$  estimado sobre las  $M'$  muestras
  - ▶ Tomamos un nuevo subconjunto de  $M'$  muestras  $\Rightarrow$  Repetimos el proceso actualizando los parámetros
- 
- ▶ Cada subconjunto de  $M'$  muestras se conoce como *batch* (*minibatch*)
  - ▶ Si queremos seguir actualizando los parámetros, podemos volver a pasar por la muestra (en otro orden)  
 $\Rightarrow$  El estimador del gradiente deja de consistir en muestras i.i.d.  $\Rightarrow$  Sesgo
  - ▶ Cada pasada por el conjunto completo de muestras se conoce como *epoch*

Si cada vez que yo quiero actualizar mis parámetros, voy a hacerlo de partes.  $M' < M$ . El costo que pago va a ser un poco peor. No me importa porque lo puedo actualizar mas rápido, me acerco igual al optimo. me acerco en una dirección no tan correcta pero me acerco igual. Este subconjunto de muestras es lo que se conoce com un batch o mini-batch.

Cada vez que hago un paso por todo el dataset, se conoce como Epoch.

Acá volvemos a la pregunta del mínimo local. No importa si no llego al mínimo, lo único que yo quiero es ir achicando la función  $J$ , no me interesa si llegue o no al mínimo siempre y cuando siga achicando la función  $J$ . En general, nunca vamos a llegar a ningún mínimo, por lo general vamos a parar cuando lleguemos a una mejora muy chica.

El Lugar “random” donde yo inicialice mis parámetros va a afectar el llegar a uno u otro mínimo local si existieran. Pero medio que no nos importa mientras el error sea chico y baje el  $J$

- ▶ **Implementación práctica** del uso de batches y epochs
  - ⇒ Para cada epoch, reordenamos los índices  $\{1, \dots, M\}$  de manera aleatoria
  - ⇒ Elegimos  $M'$  índices de manera consecutiva del conjunto reordenado ⇒ Un batch
  - ⇒ Una vez que llegamos a la última muestra, termina el epoch
  - ⇒ Realizamos un nuevo reordenamiento aleatorio de los índices y repetimos ⇒ Nueva epoch
  
- ▶ **Valores de  $M'$  más grandes mejoran el estimador, pero no de manera lineal**
- ▶ **Procesamiento en paralelo** ⇒ Cada muestra va a un procesador ⇒ Memoria escala con  $M'$
- ▶ Si los batches son muy pequeños pueden quedar procesadores sin utilizar
- ▶ Es común utilizar una cantidad de batches que sean potencias de 2 (especialmente en GPUs)
- ▶ La aleatoriedad del SGD parece tener un efecto regularizador que ayuda a la generalización

Nos estamos sumando mas hiperparametros: batch size y cantidad de epochs.

Si yo tengo una función convexa (no es el mas común de los casos), tomando unos pequeños baches mi GDE va a converger igual

- ▶ Cuando usamos SGD estamos estimando el gradiente ⇒ **La estimación no es perfecta**
  - ⇒ El algoritmo va a seguir saltando alrededor del mínimo por cambios en cada estimación
  - ⇒ **Es necesario reducir la tasa de aprendizaje a medida que nos acercamos al mínimo**

#### Convergencia de SGD

Cuando usamos SGD, cambiamos la tasa de aprendizaje a cada iteración:

$$\theta_{k+1} \leftarrow \theta_k - \eta_k \partial \hat{R} / \partial \theta_k$$

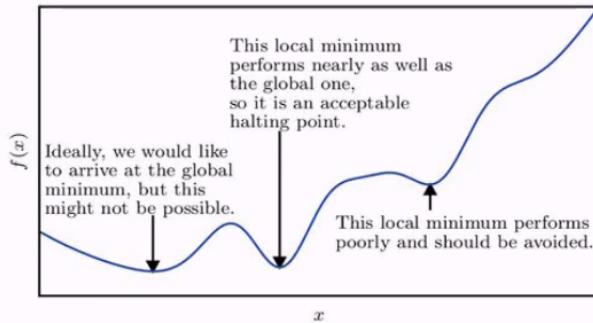
Si la tasa de aprendizaje satisface que

$$\sum_{k=1}^{\infty} \eta_k = \infty \quad \text{y} \quad \sum_{k=1}^{\infty} \eta_k^2 < \infty$$

entonces el algoritmo de SGD converge a un mínimo.

⇒ Típicamente  $\eta_k = (1 - k/\tau)\eta_0 + k\epsilon_\tau/\tau$  para algún horizonte  $\tau$

- En general, los algoritmos de optimización no ofrecen garantías de optimalidad
  - ⇒ A menos que se trate de un problema de optimización convexa
  - ⇒ En ese caso, siempre se garantiza que se llega al punto óptimo
- En casos no-convexos, como lo es la mayoría de los problemas de deep learning, hay que tener cuidado



## Mejorando SGD: Momentum

- Los problemas con mínimos locales parecen ser menos relevantes en la práctica
- Los problemas con puntos de ensilladura son un poco más críticos ⇒ Diseñar métodos para superarlos
  - ⇒ Todos los métodos construyen sobre las nociones básicas de SGD (gradiente descendente)
- Momentum: Acelerar la convergencia utilizando una media móvil de los últimos valores
  - ⇒ El algoritmo tiene tendencia a moverse en la misma dirección en la que venía

### SGD con Momentum

Dado: tasa de aprendizaje  $\eta$ , parámetro  $\alpha$ , valor inicial  $\theta_0$ , velocidad inicial  $v_0$

Para cada  $k = 0, 1, \dots$   
 Obtener un batch de muestras  $\{\mathbf{x}_1, \dots, \mathbf{x}_{M'}\}$   
 Estimar el gradiente:  $\mathbf{g}_k \leftarrow M'^{-1} \sum_{m'} \partial J(f(\mathbf{x}_{m'}; \theta_k)) / \partial \theta_k$   
 Calcular la velocidad  $\mathbf{v}_{k+1} \leftarrow \alpha \mathbf{v}_k - \eta \mathbf{g}_k$   
 Actualizar los parámetros  $\theta_{k+1} \leftarrow \theta_k + \mathbf{v}_{k+1}$

- Es evidente para cualquier usuario de deep learning que la tasa de aprendizaje es fundamental  
⇒ Y es uno de los hiperparámetros más difíciles de determinar correctamente

## AdaGrad (Duchi et al , 2011)

Dado: tasa de aprendizaje  $\eta$ , valor inicial  $\theta_0$ , constante  $\delta \approx 10^{-7}$

Inicializar  $\mathbf{r}_0 = \mathbf{0}$

Para cada  $k = 0, 1, \dots$

Obtener un batch de muestras  $\{\mathbf{x}_1, \dots, \mathbf{x}_{M'}\}$

Estimar el gradiente:  $\mathbf{g}_k \leftarrow M'^{-1} \sum_{m'} \partial J(f(\mathbf{x}_{m'}; \theta_k)) / \partial \theta_k$

Calcular el cuadrado del gradiente  $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \mathbf{g}_k^2$

Actualizar los parámetros  $\theta_{k+1} \leftarrow \theta_k - \frac{\eta}{\delta + \sqrt{\mathbf{r}_{k+1}}} \odot \mathbf{g}_k$

- Parámetros con gradiente elevado decrecen más rápido ⇒ Avanza en direcciones no muy inclinadas
- Acumular gradientes desde el principio del entrenamiento ocasiona un excesivo decrecimiento de  $\eta$

Adam es la que mas se usa, en la slide anterior que no llegue a copiar esta RMS prop. Ver pdf si interesa, pero son otros algoritmos.

## ADAM (Kingma y Ba , 2014)

Dado: tasa de aprendizaje  $\eta$ , decaimientos  $\rho_1, \rho_2$ , valor inicial  $\theta_0$ , constante  $\delta \approx 10^{-8}$

Inicializar  $\mathbf{r}_0 = \mathbf{0}$  y  $\mathbf{s}_0 = \mathbf{0}$

Para cada  $k = 0, 1, \dots$

Obtener un batch de muestras  $\{\mathbf{x}_1, \dots, \mathbf{x}_{M'}\}$

Estimar el gradiente:  $\mathbf{g}_k \leftarrow M'^{-1} \sum_{m'} \partial J(f(\mathbf{x}_{m'}; \theta_k)) / \partial \theta_k$

Estimar el primer momento  $\mathbf{s}_{k+1} \leftarrow \rho_1 \mathbf{s}_k + (1 - \rho_1) \mathbf{g}_k$

Estimar el segundo momento  $\mathbf{r}_{k+1} \leftarrow \rho_2 \mathbf{r}_k + (1 - \rho_2) \mathbf{g}_k^2$

Corregir el sesgo  $\mathbf{s}_{k+1} \leftarrow \mathbf{s}_{k+1} / (1 - \rho_1^{k+1})$  y  $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_{k+1} / (1 - \rho_2^{k+1})$

Actualizar los parámetros  $\theta_{k+1} \leftarrow \theta_k - \eta \mathbf{s}_{k+1} / (\delta + \sqrt{\mathbf{r}_{k+1}})$

- Se utiliza momentum para estimar el gradiente (ver vector  $\mathbf{s}$ )
- Se corrigen los estimadores de primer y segundo orden

## Inicialización de los parámetros:

- ▶ Algunos algoritmos no son iterativos sino que proponen una solución para el punto de interés
- ▶ Otros algoritmos son iterativos y convergen independientemente de la inicialización
- ▶ En Deep Learning, los algoritmos son iterativos y son sensibles a la elección de  $\theta_0$ 
  - ⇒ Los puntos de inicialización determinan el tipo de solución, la convergencia y la velocidad
- ▶ Las estrategias de inicialización son mayormente heurísticas (dificultad del problema)
- ▶ Objetivo primordial de la inicialización: quebrar la simetría de las distintas hidden units
- ▶ Solución típica: elegir los pesos  $W_0$  de manera aleatoria (normal o uniforme)
  - ⇒ La distribución no parece afectar mucho, pero la escala sí
  - ⇒ Valores grandes de inicialización van a quebrar mejor la simetría
  - ⇒ Si son muy grandes puede que desestabilicen numéricamente la optimización
- ▶ Elegir los vectores de offset  $b_0$  a una determinada constante (cercana a cero)

Fernando siempre usa la q viene por default

### Para resumir la clase:

- ▶ Qué quiere decir aprender ⇒ Resolver el problema de minimización del riesgo empírico
  - ⇒ Usar los datos porque no contamos con la distribución de los mismos
  - ⇒ Problema que persiste: encontrar la mejor función
- ▶ Optimizar sobre el espacio de todas las funciones posibles es prácticamente imposible
  - ⇒ Elegir una parametrización de la función de aprendizaje ⇒ Redes Neuronales
  - ⇒ Las redes neuronales son cascadas de capas ⇒ Transformación lineal + activación no lineal
  - ⇒ Las redes neuronales tienen hiperparámetros a cargo del diseñador: profundidad, ancho, activación
- ▶ Optimizar sobre parámetros se puede resolver ⇒ Pero necesitamos automatizar la optimización
  - ⇒ Algoritmo de gradiente (estocástico) descendente
  - ⇒ Algoritmo de backpropagation (calcular automáticamente la derivada ⇒ regla de la cadena)
  - ⇒ Más hiperparámetros para el diseñador: tasa de aprendizaje, tamaño de batch, cantidad de epochs

- ▶ El algoritmo de gradiente descendente requiere un riesgo derivable
  - ⇒ ¿Qué hacemos en problemas que no ofrecen riesgos derivables de manera natural?
  - ⇒ Elección de la función de costo
  
- ▶ ¿Alcanza sólo con minimizar la función de costo? ¿Se puede incorporar información externa?
  - ⇒ Regularización: Técnicas para mejorar el entrenamiento y aprendizaje