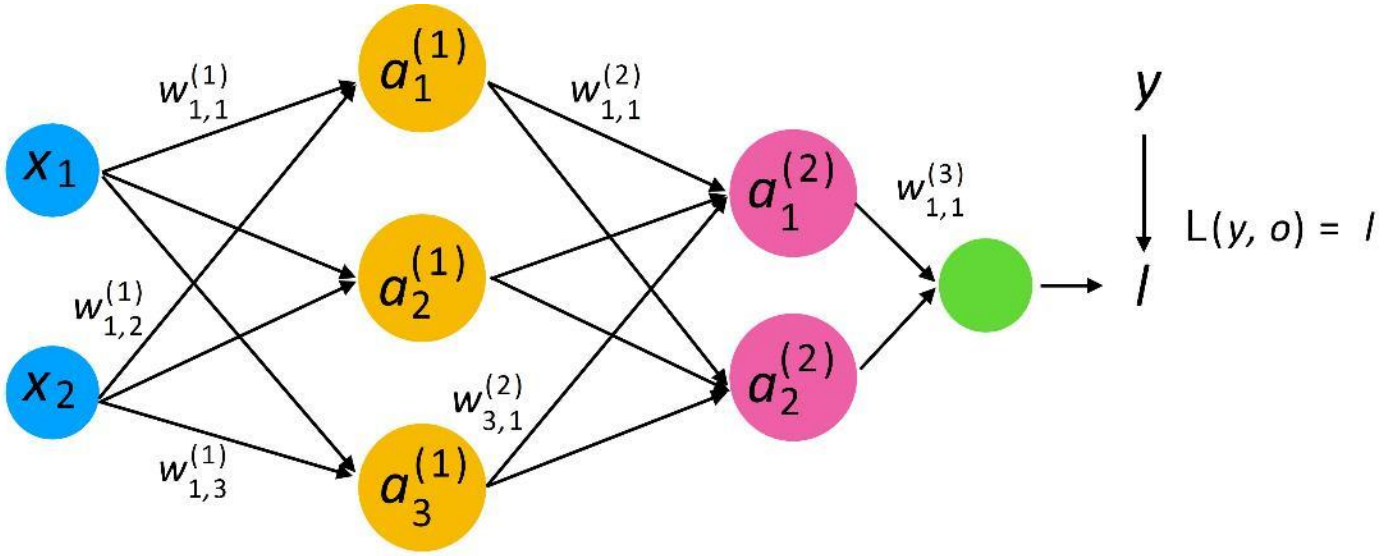
Deep Learning

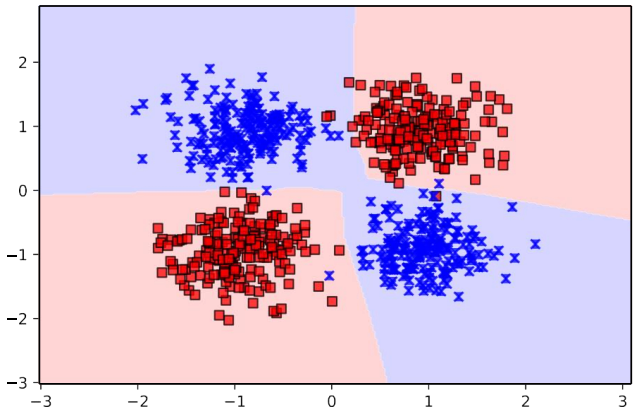
Operaciones Básicas Algebra Lineal

Ph.D Juan David Martínez Vargas
PhD. Raúl Andrés Castañeda Quintero
Escuela de Ciencias Aplicadas e Ingeniería

Operaciones Básicas Algebra Lineal

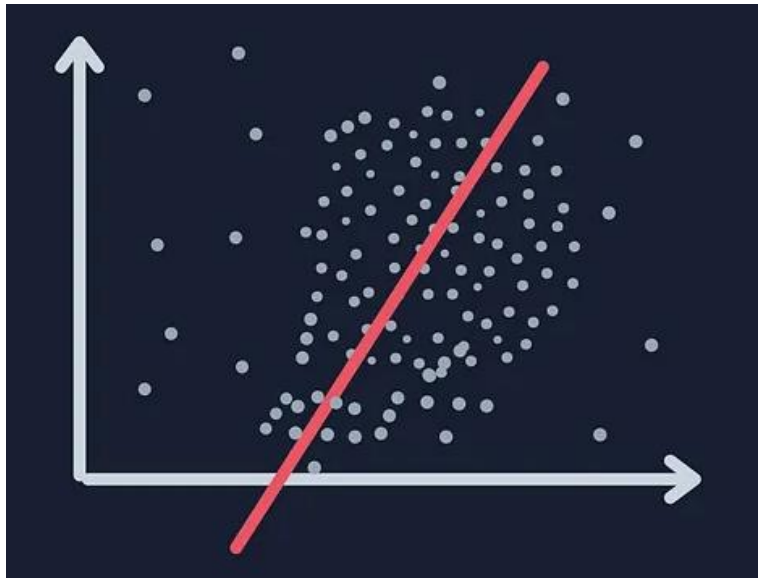


Para que podamos resolver el problema XOR, entre otras cosas…



MLP de una capa oculta con función de activación no lineal (ReLU)

Operaciones Básicas Algebra Lineal



Link

La regresión lineal muestra la relación lineal entre la variable independiente (eje X) y la variable dependiente (eje Y).

Supongamos que quiero predecir un valor→ (y)
el precio de una casa, la temperatura del día de mañana, la nota final de un estudiante.

Para hacer esa predicción, **no partimos de la nada**: tenemos mediciones, variables, características. A estas mediciones las llamamos x

Representa información de entrada

Si una variable es más importante, le asignamos un número mayor. A esos números los llamamos **pesos** (w)

El modelo más simple que existe es una **combinación lineal**

$$\hat{y} = X^T w + b$$

Operaciones Básicas Algebra Lineal

Escalar: Un **escalar** es un **número real** que representa una **magnitud sin dirección**.

$$b$$

En **Deep Learning**, los escalares se utilizan para:

- Ajustar pesos
- Medir errores (función de pérdida)
- Controlar el aprendizaje (learning rate)
- Representar salidas numéricas (regresión)

Vector: Un **vector** es un objeto matemático que representa una **colección ordenada de números**, los cuales pueden interpretarse como **componentes de magnitud y dirección** en un espacio.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

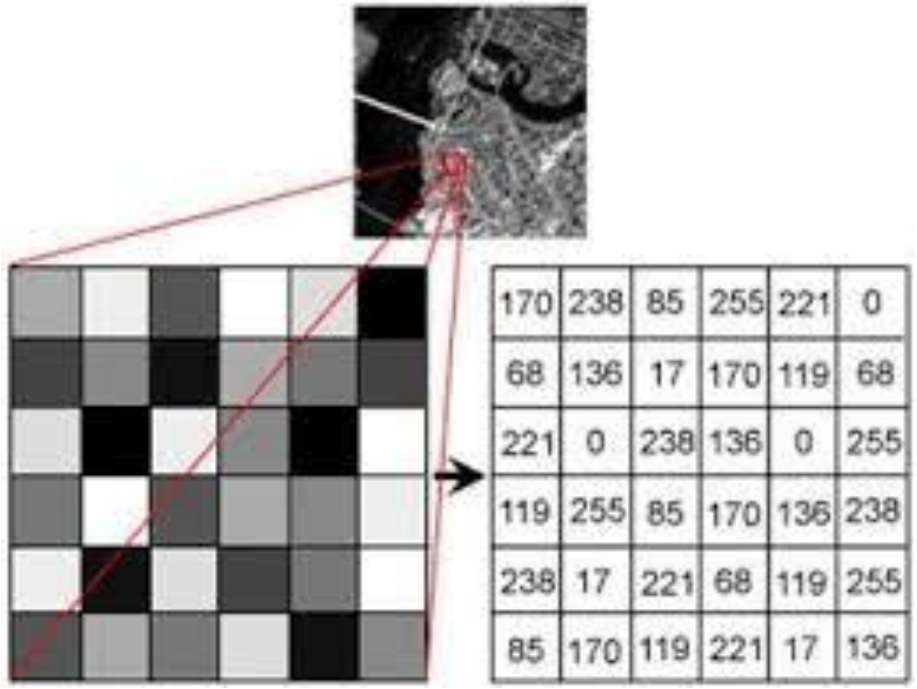
En **Deep Learning**, un vector representa **información estructurada**.

- Entrada de una red neuronal
- Pesos de una Neurona
- Salida de una capa
- Embeddings

Operaciones Básicas Algebra Lineal

Matriz: Una **matriz** es un arreglo bidimensional de números organizado en **filas y columnas**. Representa relaciones entre múltiples variables y permite describir transformaciones lineales y sistemas de datos estructurados.

$$w = \begin{bmatrix} w_{11} & w_{12} \dots w_{1n} \\ w_{21} & w_{22} \dots w_{2n} \\ \vdots & \vdots \ddots \vdots \\ w_{m1} & w_{m2} \cdots w_{mn} \end{bmatrix}$$



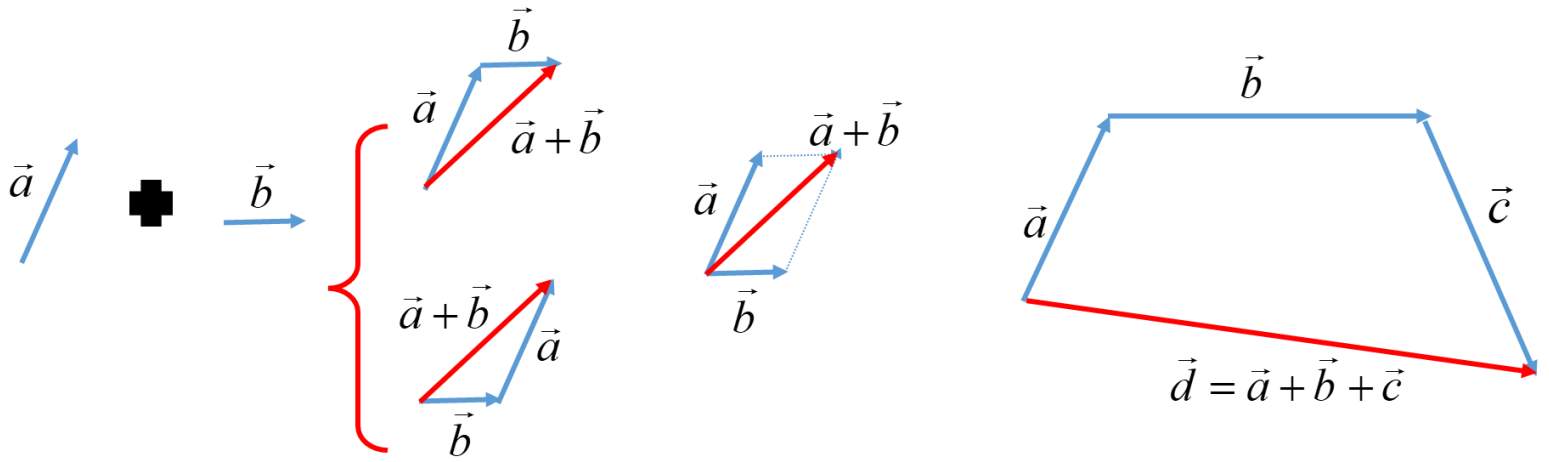
En Deep Learning, las matrices se utilizan para:

- Representar **pesos de una capa neuronal**
- Modelar **transformaciones lineales** entre capas
- Implementar **convoluciones** y operaciones lineales
- Calcular **atención** y relaciones entre múltiples vectores

Operaciones Básicas Algebra Lineal

Suma de Vectores es una operación que combina dos vectores de **igual dimensión** sumando sus componentes correspondientes.

En DL se utiliza para: Añadir el **sesgo (bias)** en una neurona, combinar **activaciones, i**mplementar **conexiones residuales**



$$\vec{a} = a_1 + a_2 + a_3 + \ldots\ldots + a_n \qquad \vec{p} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_n \end{pmatrix}$$

$$\vec{b} = (b_1, b_2, b_3, \ldots\ldots, b_n)$$

Vectores en $\mathbb{R}^2$

$$\vec{a} = a_x\hat{i} + a_y\hat{j}$$

$$\vec{b} = b_x\hat{i} + b_y\hat{j}$$

$$\vec{R} = \vec{a} + \vec{b}$$

$$\vec{R} = \underbrace{(a_x + b_x)}_{R_x}\hat{i} + \underbrace{(a_y + b_y)}_{R_y}\hat{j}$$

$$\vec{R} = R_x\hat{i} + R_y\hat{j}$$

$$\left|\vec{R}\right| = \left(R_x^2 + R_y^2\right)^{1/2}$$

$$\theta = \tan^{-1}\left(\frac{R_y}{R_x}\right)$$

Operaciones Básicas Algebra Lineal

Propiedades de los vectores

- ✓ Conmutativa $\vec{u}+\vec{v}=\vec{v}+\vec{u}$
- ✓ Asociativa suma $(\vec{u}+\vec{v})+\vec{w}=\vec{u}+(\vec{v}+\vec{w})$
- ✓ Distributiva $c(\vec{u}+\vec{v})=c\vec{u}+c\vec{v}$
- ✓ Asociativa multiplicación $a(b\vec{u})=(ab)\vec{u}$

En DL se utiliza para:

Ajustar la contribución de las características: Escala la importancia de cada componente del vector de entrada.
Pesos y activaciones: Los pesos de una neurona escalan las entradas antes de combinarlas.
Learning rate: El gradiente (vector) se escala por un escalar para controlar el tamaño del paso:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L$$

Normalización y estabilidad numérica: Controla magnitudes para evitar explosión o desaparición de gradientes.

Escalar por un vector

La **multiplicación de un escalar por un vector** consiste en multiplicar cada componente del vector por un mismo número real

- ✓ El resultado es un vector.

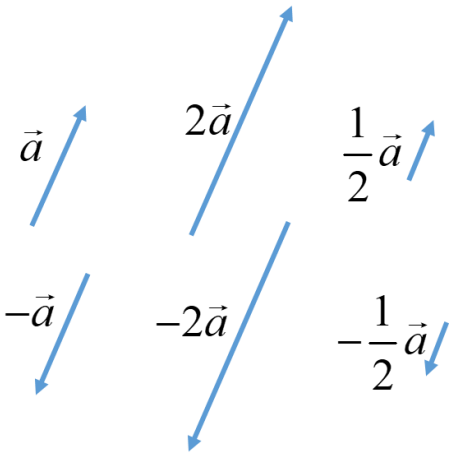
Sí $0<c<1$ la magnitud disminuye.

Sí $c>1$ la magnitud aumenta.

Sí $-1<c<0$ la dirección del vector cambia 180° y su magnitud disminuye.

Si $c<-1$ la dirección del vector cambia 180° y su magnitud aumenta.

el $c=0$ vector resultado es nulo.



UNIVERSIDAD EAFIT

Operaciones Básicas Algebra Lineal

Producto Punto

El resultado de realizar un producto punto es una cantidad escalar
Se considera la proyección de un vector sobre otro

Notación: $\vec{a} \cdot \vec{b}$

Matemáticamente se puede operar de dos formas diferentes:

1) $\vec{a} \cdot \vec{b} = |a||b|cos\theta$

2) $\vec{x} = (x_1, x_2, x_3, \dots, x_n) \quad \vec{w} = (w_1, w, w_3, \dots, w_n)$

$$\vec{x} \cdot \vec{w} = \sum_{i=1}^{n} w_i x_i = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

En DL se utiliza para: el producto punto se usa para combinar entradas con pesos y medir similitud entre representaciones, siendo la operación central de neuronas, atención y convoluciones.

Propiedades básicas del producto punto

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$$

$$\vec{a} \cdot (\vec{b} + \vec{c}) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$$

$$p(\vec{b} \cdot \vec{c}) = (p\vec{a}) \cdot \vec{b}$$

$$\vec{a} \cdot 0 = 0$$

$$\vec{a} \cdot \vec{a} = |\vec{a}|^2$$

Operaciones Básicas Algebra Lineal

Ejemplo: Considere los vectores $\vec{a} = -5\hat{i} + 4\hat{j}$ y $\vec{b} = -3\hat{i} - 8\hat{j}$

1) Encuentre el producto punto utilizando ambos métodos y compare los resultados.

$$\vec{a}\bullet\vec{b} = a_x b_x + a_y b_y$$

$$\vec{a}\bullet\vec{b} = (-5)(-3) + (4)(-8)$$

$$\vec{a}\bullet\vec{b} = (-5)(-3) + (4)(-8)$$

$$\vec{a}\bullet\vec{b} = 15 - 32$$

$$\boxed{\vec{a}\bullet\vec{b} = -17}$$

$$\vec{a}\bullet\vec{b} = |\vec{a}||\vec{b}|\cos\theta$$

$$|\vec{a}| = \left(a_x^2 + a_y^2\right)^{1/2} = \left(5^2 + 4^2\right)^{1/2} = 6,40$$

$$|\vec{b}| = \left(b_x^2 + b_y^2\right)^{1/2} = \left(3^2 + 8^2\right)^{1/2} = 8,54$$

$$\theta = \alpha + \beta$$

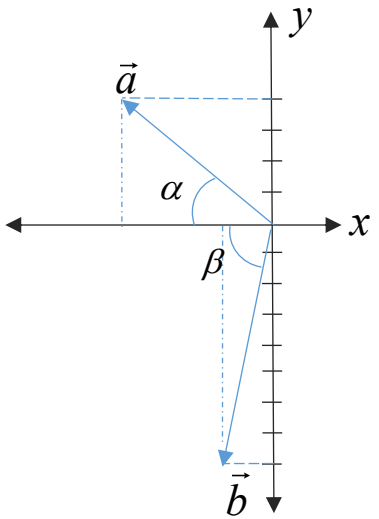
$$\alpha = \tan^{-1}\left(\frac{4}{-5}\right) = 38,65^\circ$$

$$\beta = \tan^{-1}\left(\frac{8}{3}\right) = 69,44^\circ$$

$$\theta = 38,65 + 69,44 = 108,09^\circ$$

$$\vec{a}\bullet\vec{b} = (6,40)(8,54)\cos(108,09)$$

$$\boxed{\vec{a}\bullet\vec{b} = -16,97 \approx -17}$$



$$\theta = \cos^{-1}\left(\frac{a\bullet b}{|\vec{a}||\vec{b}|}\right) = \cos^{-1}\left(\frac{-17}{|6,40||8,54|}\right)$$

$$\theta = 108,12^\circ$$

Operaciones Básicas Algebra Lineal

Multiplicación de matrices.

A @ B → multiplicación matricial

$$A @ B = C; \quad C \in \mathbb{R}^{m \times p}$$

$A \in \mathbb{R}^{m \times n}$ A es una matriz de números reales con m filas y n columnas.

$B \in \mathbb{R}^{n \times p}$ B es una matriz de números reales con n filas y p columnas.

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

A * B → multiplicación punto a punto

$$(A * B)_{ij} = A_{ij} * B_{ij} = C; \quad C \in \mathbb{R}^{m \times n}$$

$A \in \mathbb{R}^{m \times n}$ A es una matriz de números reales con m filas y n columnas.

$B \in \mathbb{R}^{m \times n}$ B es una matriz de números reales con m filas y n columnas.

Operaciones Básicas Algebra Lineal

$$A \,@\, B = C; \quad C \in \mathbb{R}^{m \times p}$$

$$(AB)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

- Tomamos la fila i de A
- Tomamos la columna j de B
- Multiplicamos elemento a elemento
- Sumamos todo

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \Rightarrow C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ - & - \end{bmatrix}$$

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \Rightarrow C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

$$\Rightarrow C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A$$

Operaciones Básicas Algebra Lineal

En DL no trabajamos con un solo número, ni con un solo vector o una sola matriz, sino con **datos multidimensionales**. Para manejar todo eso de forma unificada usamos **tensores**

Tensores: Un **tensor** es una **generalización de los escalares, vectores y matrices** a un número arbitrario de dimensiones

Objeto	Tensor de orden	Ejemplo
Escalar	Tensor de orden 0	$x \in \mathbb{R}$
Vector	Tensor de orden 1	$\mathbf{x} \in \mathbb{R}^n$
Matriz	Tensor de orden 2	$\mathbf{W} \in \mathbb{R}^{m \times n}$
Tensor	Orden ≥ 3	$\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times \cdots}$

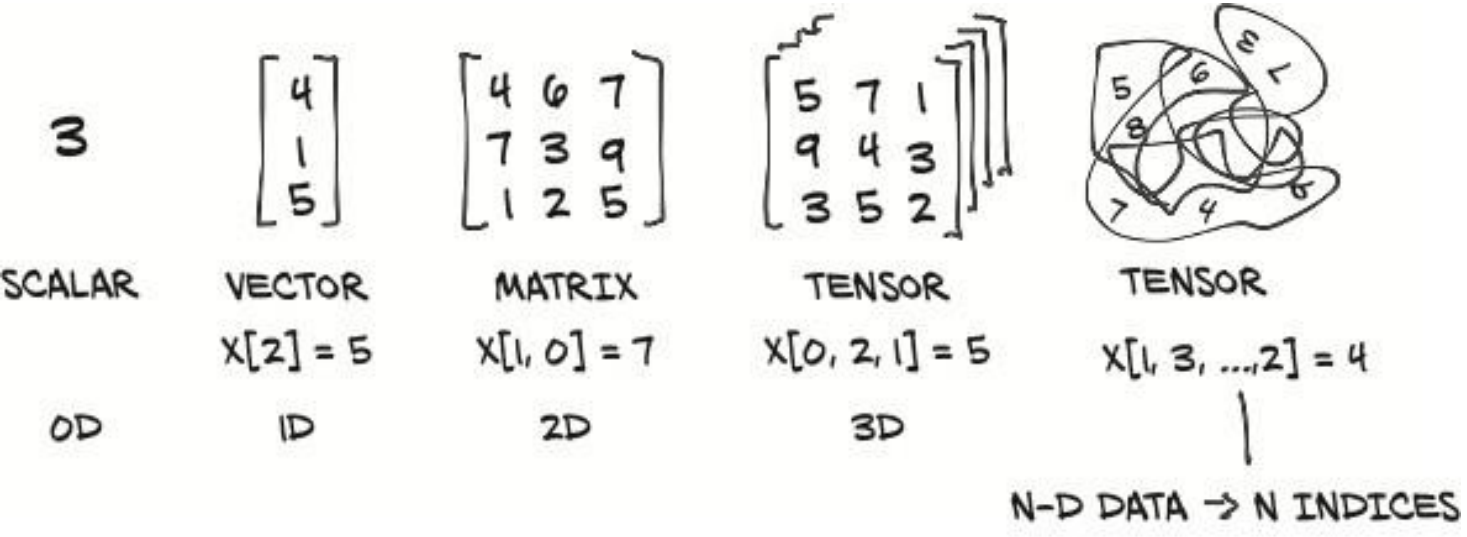


Figure 3.2 Tensors are the building blocks for representing data in PyTorch.

UNIVERSIDAD EAFIT

Tensores

En el contexto de TensorFlow, NumPy, PyTorch, etc.,tensores = arreglos multidimensionales

La dimensionalidad coincide con el número de índices de .shape

Importa la librería PyTorch

Crear un tensor

PyTorch imprime el contenido del tensor

Dimensiones del tensor

```
In [1]: import torch

In [2]: t = torch.tensor([[1, 2, 3], [4, 5, 6]])

In [3]: t
Out[3]:
tensor([[1, 2, 3],
        [4, 5, 6]])

In [4]: t.shape
Out[4]: torch.Size([2, 3])

In [5]: t.ndim
Out[5]: 2
```

Arreglos multidimensionales como tensores

numpy.array / numpy.ndarray = (representación estructural de datos de un tensor)

pytorch.tensor / pytorch.Tensor = (representación estructural de datos de un tensor)

```
In [1]: import numpy as np

In [2]: a = np.array([1., 2., 3.])

In [3]: print(a.dtype)
float64

In [4]: print(a.shape)
(3,)
```

```
In [5]: import torch

In [6]: b = torch.tensor([1., 2., 3.])

In [7]: print(b.dtype)
torch.float32

In [8]: print(b.shape)
torch.Size([3])
```

La sintaxis de NumPy y PyTorch es muy similar

```
In [9]: a = np.array([1., 2., 3.])

In [10]: print(a.dot(a))
14.0


In [12]: print(b.matmul(b))
tensor(14.)

In [13]: b
Out[13]: tensor([1., 2., 3.])

In [14]: b.numpy()
Out[14]: array([1., 2., 3.], dtype=float32)
```

Concepto	NumPy	PyTorch
Vector	np.array	torch.tensor
Producto punto	a.dot(a)	b.matmul(b)
Resultado	escalar	tensor escalar
Conversión	—	b.numpy()

```python
import numpy as np

# array([0, 1, 2])
# x.ndim -> 1
# x.shape() -> (3,)
x = np.array([0, 1, 2])

# np.int64(2)
x[2]
```

```python
import torch

# tensor([0, 1, 2])
# x.ndim -> 1
# x.shape() -> torch.Size([3])
x = torch.arange(3)

# tensor(2)
x[2]
```

¿Cuál es la diferencia?

Nota: Tradicionalmente, PyTorch usaba "matmul", pero hoy en día "dot" también funciona.

```
In [12]: print(b.matmul(b))
tensor(14.)
```

```
In [15]: print(b.dot(b))
tensor(14.)

In [16]: print(b @ b)
tensor(14.)
```

Tipos de datos para memorizar

NumPy data	Tensor data type	
numpy.uint8	torch.ByteTensor	
numpy.int16	torch.ShortTensor	
numpy.int32	torch.IntTensor	
numpy.int	torch.LongTensor	
numpy.int64	torch.LongTensor	default int in NumPy & PyTorch
numpy.float16	torch.HalfTensor	
numpy.float32	torch.FloatTensor	default float in PyTorch
numpy.float	torch.DoubleTensor	
numpy.float64	torch.DoubleTensor	default float in NumPy

- Ej., int32 significa entero de 32 bits.
- Los flotantes de 32 bits son menos precisos que los de 64 bits, pero para redes neuronales, no importa mucho.
- Para GPUs regulares, usualmente queremos flotantes de 32 bits (vs 64 bits) para mayor rapidez.

Especifica el tipo al momento de la construcción

```
In [21]: c = torch.tensor([1., 2., 3.], dtype=torch.float)

In [22]: c.dtype
Out[22]: torch.float32

In [23]: c = torch.tensor([1., 2., 3.], dtype=torch.double)

In [24]: c.dtype
Out[24]: torch.float64

In [25]: c = torch.tensor([1., 2., 3.], dtype=torch.float64)

In [26]: c.dtype
Out[26]: torch.float64
```

Entonces, ¿por qué no simplemente usar NumPy?

- PyTorch tiene soporte para GPU:
  - Podemos cargar el conjunto de datos y los parámetros del modelo en la memoria de la GPU.
  - En la GPU, luego tenemos mejor paralelismo para computar (muchas) multiplicaciones de matrices.

- PyTorch tiene diferenciación automática (más adelante).

- También, PyTorch implementa muchas funciones convenientes para el aprendizaje profundo (más adelante).

PyTorch nos permite entrenar modelos grandes de forma eficiente usando GPU, calcular derivadas automáticamente y construir redes profundas con herramientas ya implementadas.

Ejemplos

Queremos estimar el precio de una casa usando solo dos cosas

Tamaño de la casa (en m²)

Número de habitaciones

$$x = \begin{bmatrix} 80 \\ 3 \end{bmatrix}$$

Las características (features)

x es un vector de características para cada entrada → es una propiedad medible de la casa.

Los pesos (weights)

w es un vector de pesos → Cuánto importa el tamaño / cuánto importa el número de habitaciones.

$$w = \begin{bmatrix} 2000 \\ 10000 \end{bmatrix}$$

Ejemplos

Queremos estimar el precio de una casa usando solo dos cosas

$$\begin{bmatrix}80\\3\end{bmatrix}\begin{bmatrix}2000\\10000\end{bmatrix}=\begin{bmatrix}160000\\30000\end{bmatrix}$$

$$xw$$

$$\begin{bmatrix}80\\3\end{bmatrix}\begin{bmatrix}2000\\10000\end{bmatrix}=(80 * 2000) + (3*10000)=190000$$

$$x^{T}w \Rightarrow \begin{bmatrix}80 & 3\end{bmatrix}\begin{bmatrix}2000\\10000\end{bmatrix}=(80 * 2000) + (3*10000)=190000$$

El bias: ¿por qué existe?

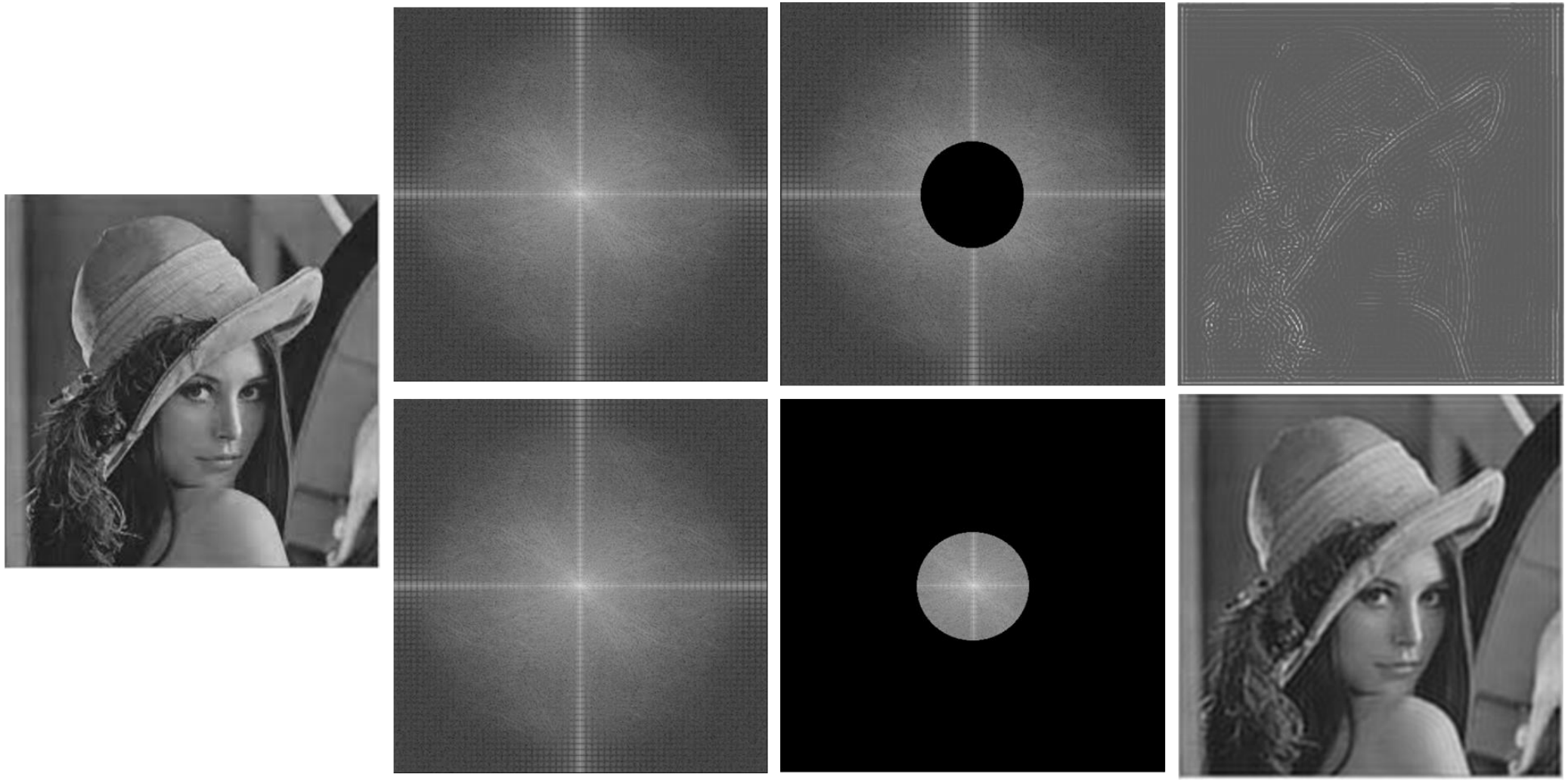
¿Qué pasa si la casa tuviera tamaño cero y cero habitaciones?

puede representar: costo mínimo del terreno, impuestos base, ubicación, servicios.
Cosas que no dependen directamente de las características que estamos usando.

El modelo toma las características de la casa, las pondera según su importancia, suma esas contribuciones y luego agrega un valor base llamado bias.

Ejemplos

¿Cuando es una multiplicación punto a punto?



Vectores

Tensores

¿Cómo llamamos a esto nuevamente en el contexto de las redes neuronales?

$$\mathbf{w}^\top \mathbf{x} + b = z$$

where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$ $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$

X w + *b* = z donde

(por eso w no es un "vector"
sino una matriz de m × 1)

Dos oportunidades de paralelismo:

- calcular el producto punto en paralelo
- calcular múltiples productos punto a la vez

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

$$\mathbf{z} = \begin{bmatrix} \mathbf{w}^\top \mathbf{x}^{[1]} + b \\ \mathbf{w}^\top \mathbf{x}^{[2]} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{[n]} + b \end{bmatrix} = \begin{bmatrix} z^{[1]} \\ z^{[2]} \\ \vdots \\ z^{[n]} \end{bmatrix}$$

Calculando la salida a partir de múltiples ejemplos de entrenamiento a la vez

$$\mathbf{X}\,\mathbf{w} + b = \mathbf{z}$$

(por eso w no es un "vector"
sino una matriz de m × 1)

Pero NumPy y PyTorch no son muy exigentes con eso:

```
In [1]: import torch

In [2]: X = torch.arange(6).view(2, 3)

In [3]: X
Out[3]:
tensor([[0, 1, 2],
        [3, 4, 5]])
In [4]: w = torch.tensor([1, 2, 3])

In [5]: X.matmul(w)
Out[5]: tensor([ 8, 26])

In [6]: w = w.view(-1, 1)

In [7]: X.matmul(w)
Out[7]:
tensor([[ 8],
        [26]])
```

igual que reshape
(razones históricas)

Calculando la salida a partir de múltiples ejemplos de entrenamiento a la vez

Dos oportunidades de paralelismo:

- calcular el producto punto en paralelo
- calcular múltiples productos punto a la vez

$$\mathbf{X}\,\mathbf{w} + b = \mathbf{z}$$

donde

$$\mathrm{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}, \quad \mathrm{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

(por eso w no es un "vector"
sino una matriz de m × 1)

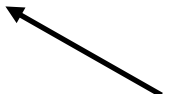
$$\mathbf{z} = \begin{bmatrix} \mathbf{w}^\top \mathbf{x}^{[1]} + b \\ \mathbf{w}^\top \mathbf{x}^{[2]} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{[n]} + b \end{bmatrix} = \begin{bmatrix} z^{[1]} \\ z^{[2]} \\ \vdots \\ z^{[n]} \end{bmatrix}$$

¿Puedes detectar el error en esta diapositiva?

Calculando la salida a partir de múltiples ejemplos de entrenamiento a la vez

$$\mathbf{X}\,\mathbf{w} + b = \mathbf{z}$$

¿Puedes detectar el error en esta diapositiva?



Esto debería ser

$$\mathbf{X}\,\mathbf{w} + \mathbf{1}_m b = \mathbf{z}$$

pero ¡nosotros los investigadores en aprendizaje profundo somos perezosos! :)

Broadcasting

- En PyTorch, funciona perfectamente.
- Esta característica (general) se llama "broadcasting"

```
In [4]: torch.tensor([1, 2, 3]) + 1
Out[4]: tensor([2, 3, 4])
```

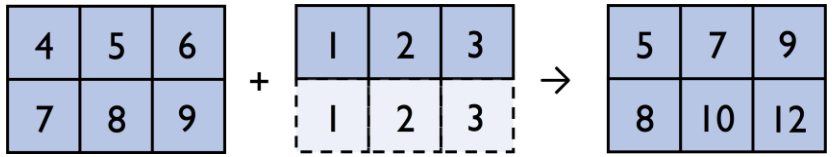
```
In [5]: t = torch.tensor([[4, 5, 6], [7, 8, 9]])

In [6]: t
Out[6]:
tensor([[4, 5, 6],
        [7, 8, 9]])

In [7]: t + torch.tensor([1, 2, 3])
Out[7]:
tensor([[ 5,  7,  9],
        [ 8, 10, 12]])
```

Broadcasting

- En PyTorch, funciona perfectamente.
- Esta característica (general) se llama "broadcasting"

```
In [4]: torch.tensor([1, 2, 3]) + 1
Out[4]: tensor([2, 3, 4])
```

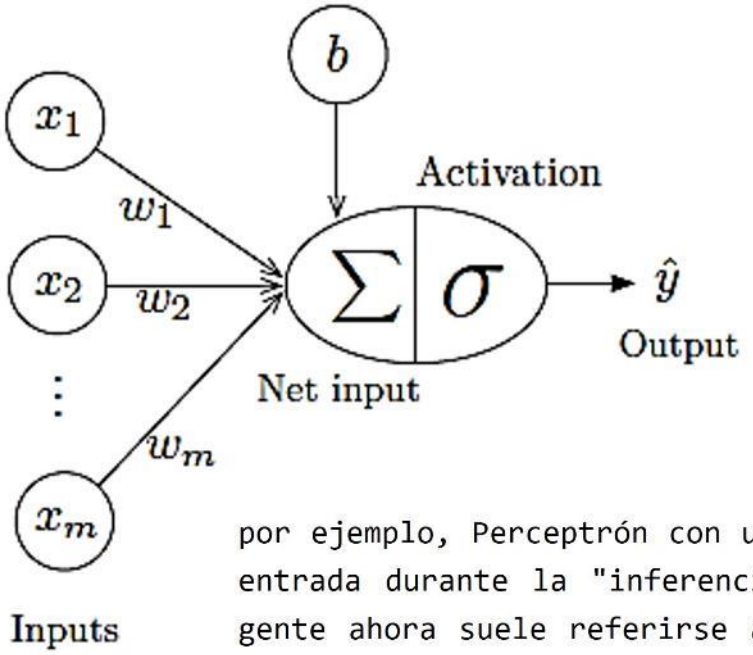

```
In [5]: t = torch.tensor([[4, 5, 6], [7, 8, 9]])

In [6]: t
Out[6]:
tensor([[4, 5, 6],
        [7, 8, 9]])

In [7]: t + torch.tensor([1, 2, 3])
Out[7]:
tensor([[ 5,  7,  9],
        [ 8, 10, 12]])
```



Se agregan dimensiones implícitas, los elementos se duplican implícitamente.

Conexiones que ya hemos visto...

$$\mathbf{x}^\top \mathbf{w} + b = z$$

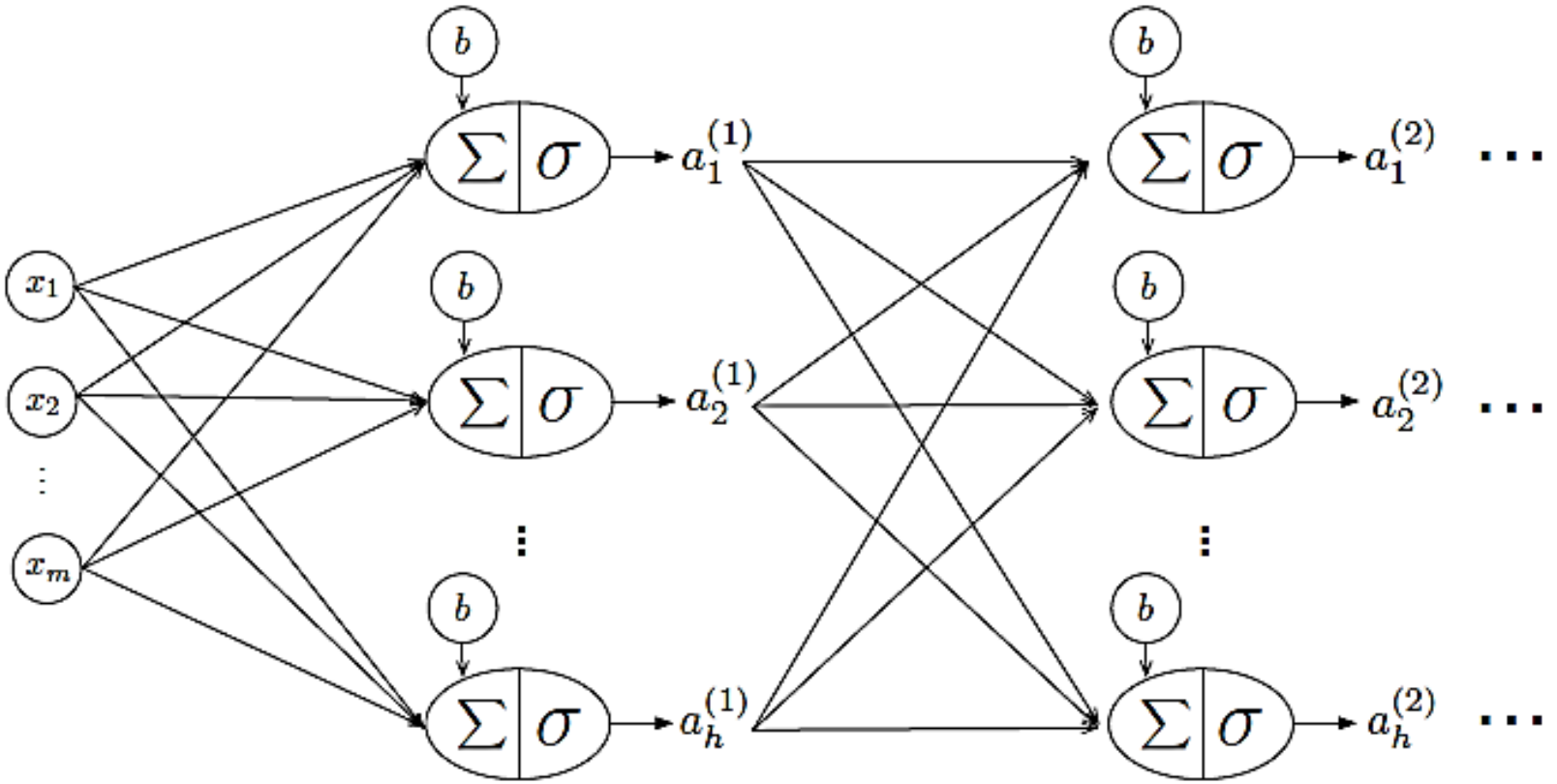


por ejemplo, Perceptrón con un ejemplo de entrenamiento como entrada durante la "inferencia"(en aprendizaje profundo, la gente ahora suele referirse a predecir la variable objetivo como "inferencia")

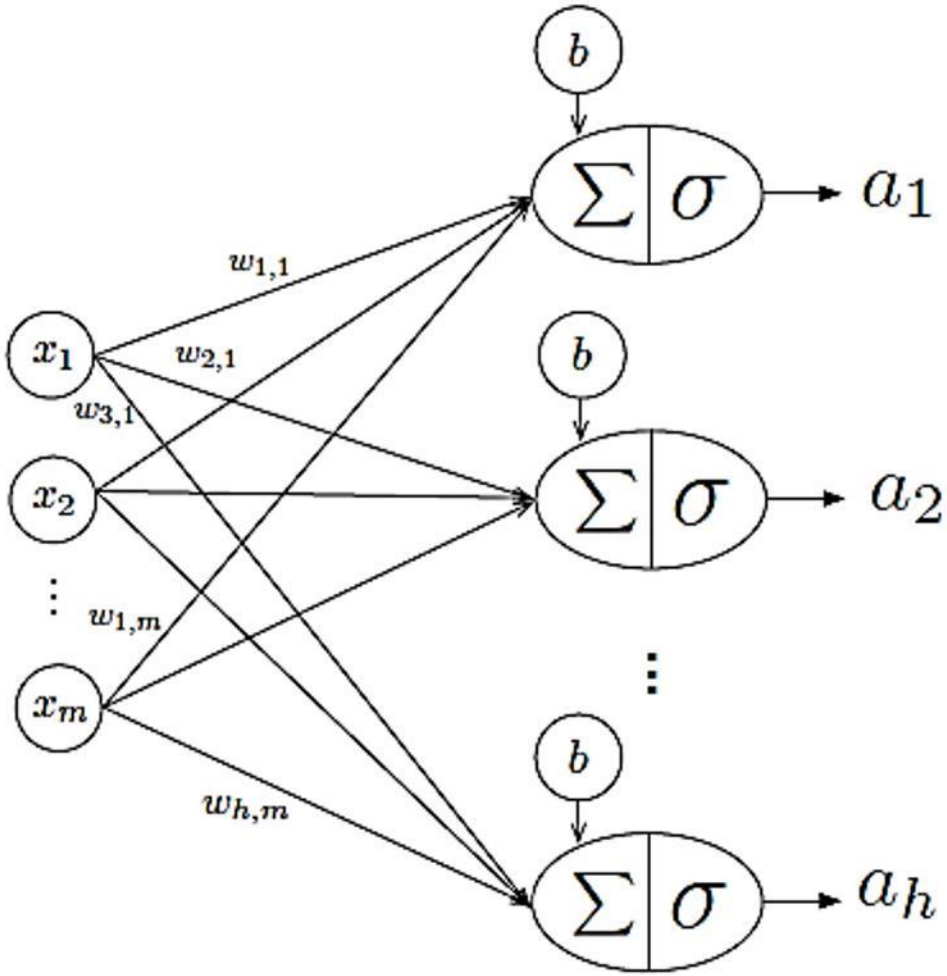
Si tenemos *n* ejemplos de entrenamiento, $\mathbf{X} \in \mathbb{R}^{n \times m}$, $\mathbf{z} \in \mathbb{R}^{n \times 1}$

X w + *b* = **z**

Conexiones que encontraremos más adelante...



Una capa totalmente conectada



$$\text{where} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \cdots & w_{h,m} \end{bmatrix}$$

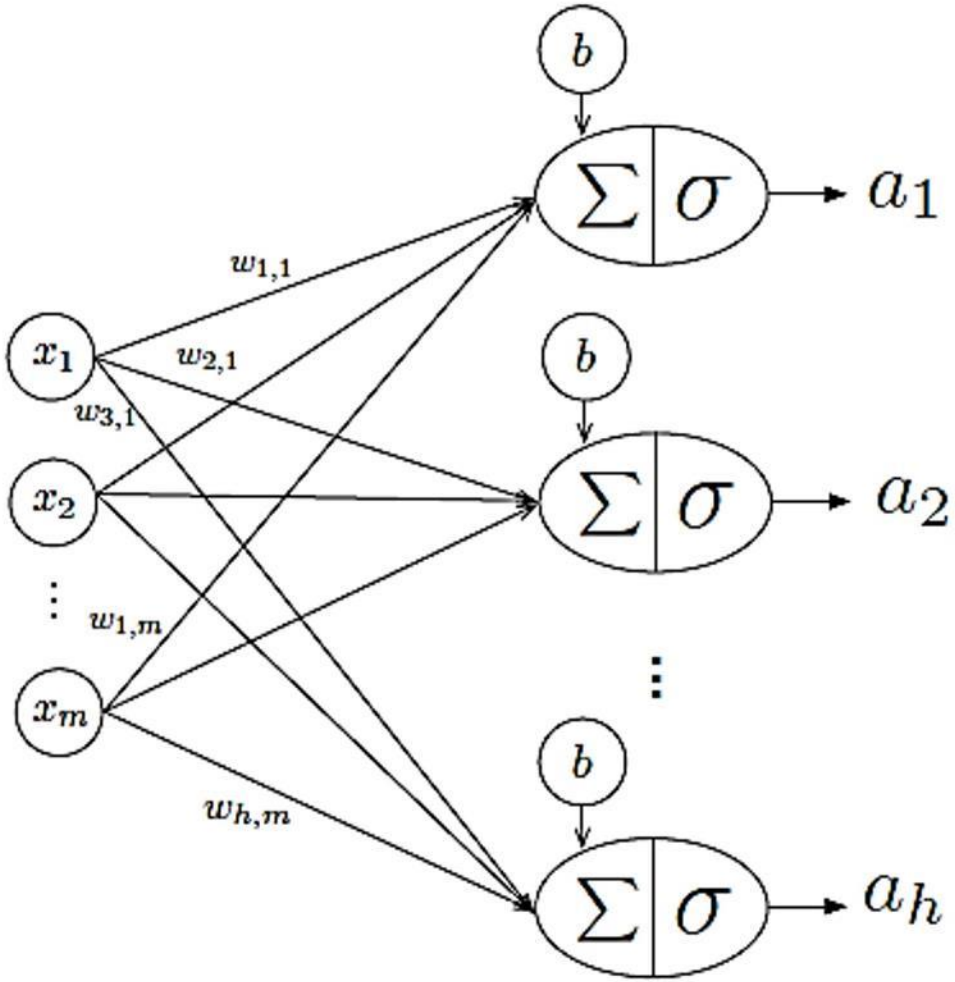
Activaciones de capa para 1 ejemplo de entrenamiento

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

$$\mathbf{a} \in \mathbb{R}^{h \times 1}$$

nota que $w_{l,w}$ se refiere al peso que conecta laentrada j-ésima con la salida i-ésima.

Una capa totalmente conectada



Activaciones de capa para n ejemplos de entrenamiento

$$\sigma\left([\mathbf{W}\mathbf{X}^\top + \mathbf{b}]^\top\right) = \mathbf{A}$$

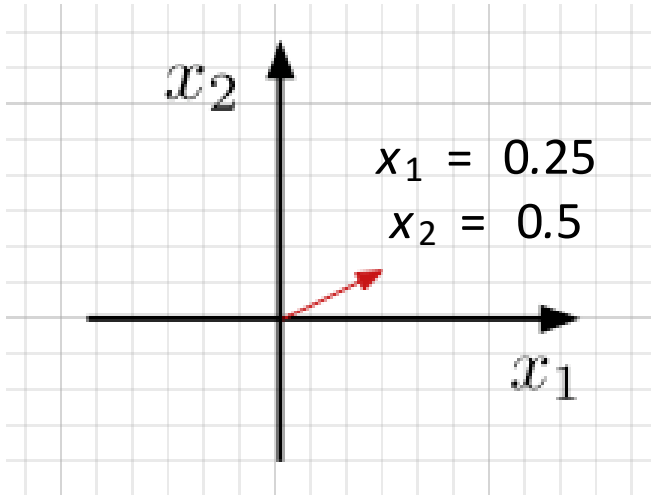
$$\mathbf{A} \in \mathbb{R}^{n \times h}$$

Los libros de texto de aprendizaje automático usualmente representan los ejemplos de entrenamiento sobre las columnas, y las características sobre las filas (en lugar de usar la "matriz de diseño") – en ese caso, podríamos omitir la transpuesta.

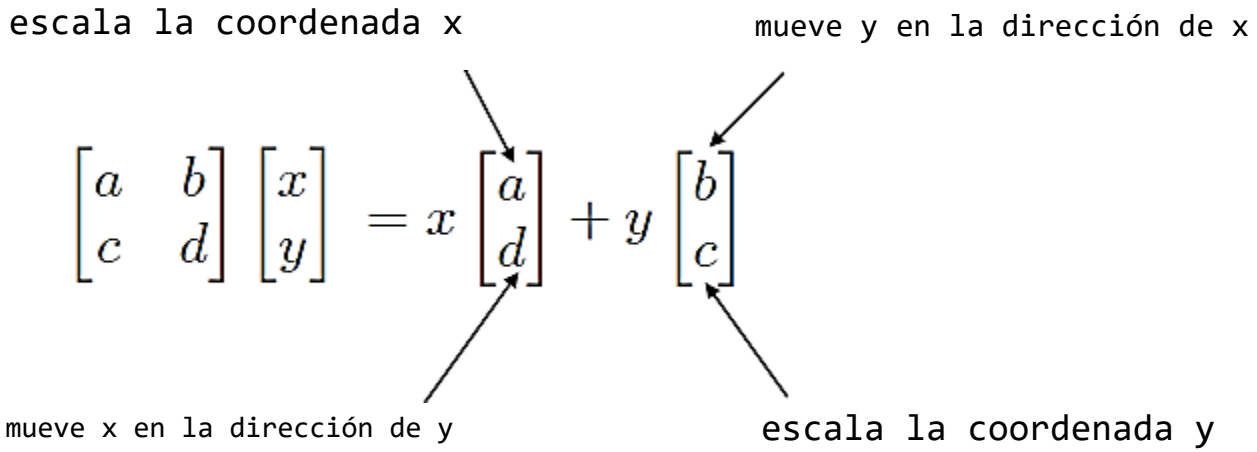
¿Pero por qué la notación Wx es intuitiva?

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Matriz de transformación



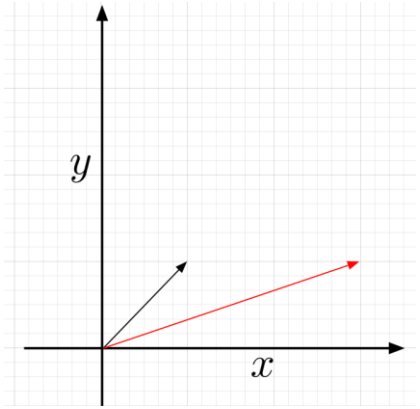
¿Pero por qué la notación Wx es intuitiva?



¿Pero por qué la notación Wx es intuitiva?

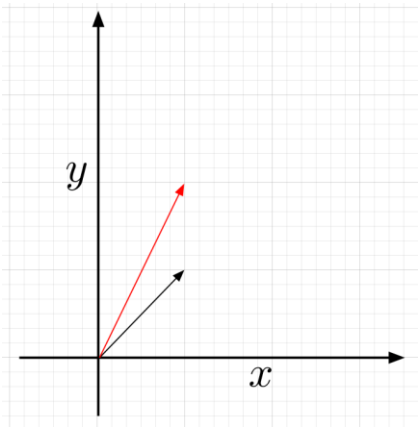
Estiramiento del eje x por un factor de 3

$$\begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ y \end{bmatrix}$$



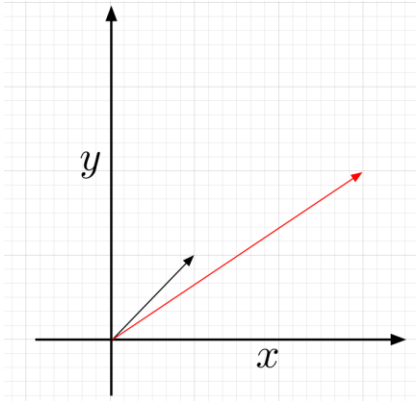
Estiramiento del eje y por un factor de 2

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ 2y \end{bmatrix}$$



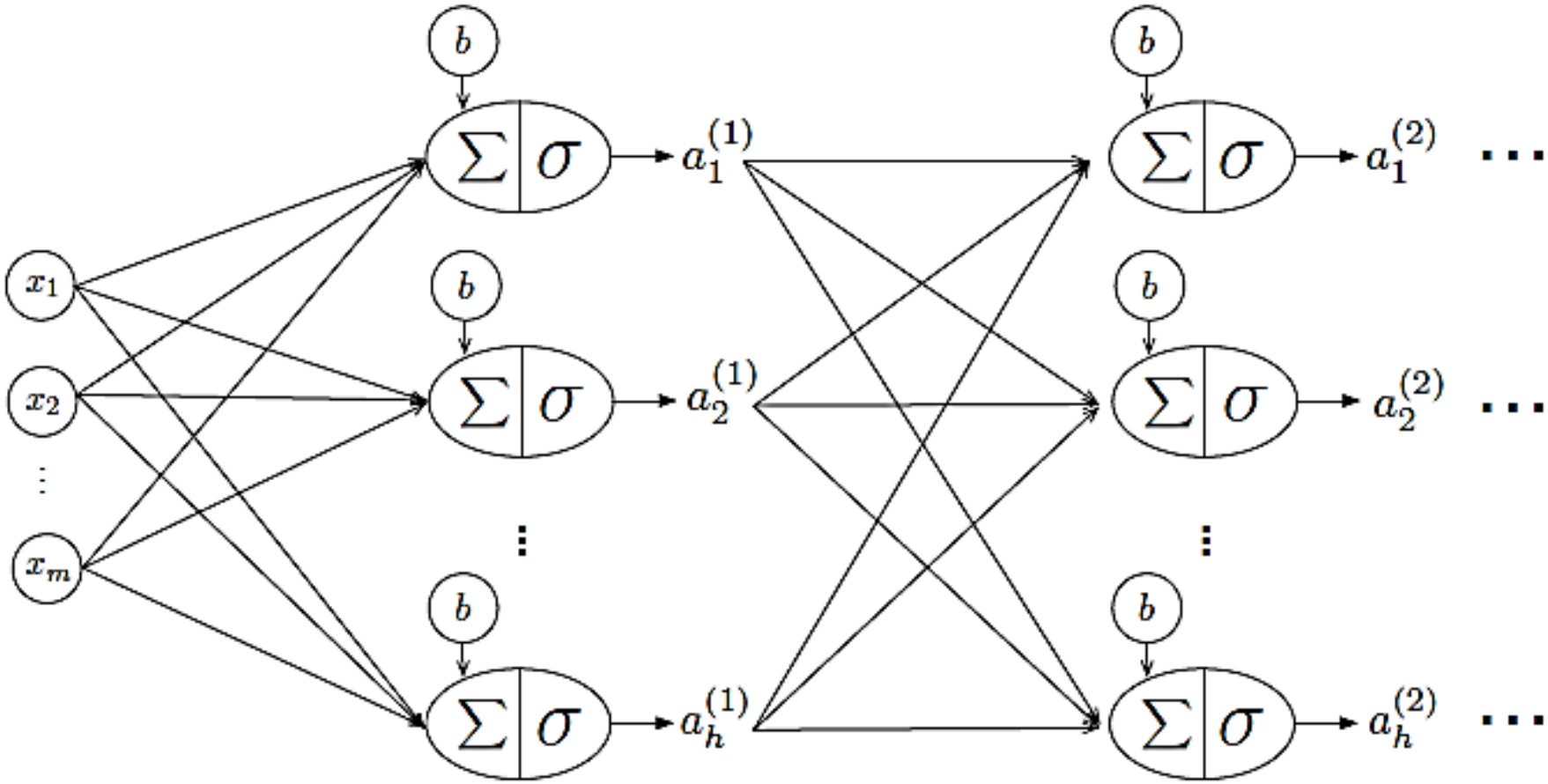
Estiramiento del eje x por un factor de 3 y del eje y por un factor de 2

$$\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}$$



Una capa totalmente conectada (lineal) en PyTorch

1. Tensores en el aprendizaje profundo
2. Tensores y PyTorch
3. Vectores, matrices y broadcasting
4. Convenciones de notación para redes neuronales
5. **Una capa totalmente conectada (lineal) en PyTorch**


b
$\sum$ σ
$a_1^{(1)}$
$a_2^{(1)}$
$a_h^{(1)}$
$a_1^{(2)}$
$a_2^{(2)}$
$a_h^{(2)}$
x_1
x_2
x_m
...

Capa totalmente conectada en PyTorch

[1]:
```
import torch
```

[2]:
```
X = torch.arange(50, dtype=torch.float).view(10, 5)
# .view() and .reshape() are equivalent
X
```

[2]:
```
tensor([[ 0.,  1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.],
        [10., 11., 12., 13., 14.],
        [15., 16., 17., 18., 19.],
        [20., 21., 22., 23., 24.],
        [25., 26., 27., 28., 29.],
        [30., 31., 32., 33., 34.],
        [35., 36., 37., 38., 39.],
        [40., 41., 42., 43., 44.],
        [45., 46., 47., 48., 49.]])
```

[3]:
```
fc_layer = torch.nn.Linear(in_features=5,
                           out_features=3)
```

[4]:
```
fc_layer.weight
```

[4]:
```
Parameter containing:
tensor([[-0.1706,  0.1684,  0.3509,  0.1649,  0.1903],
        [-0.1356,  0.0663, -0.4357,  0.2710,  0.1179],
        [-0.0736,  0.0413, -0.0186,  0.4032,  0.0992]], requires_grad=True)
```

[5]:
```
fc_layer.bias
```

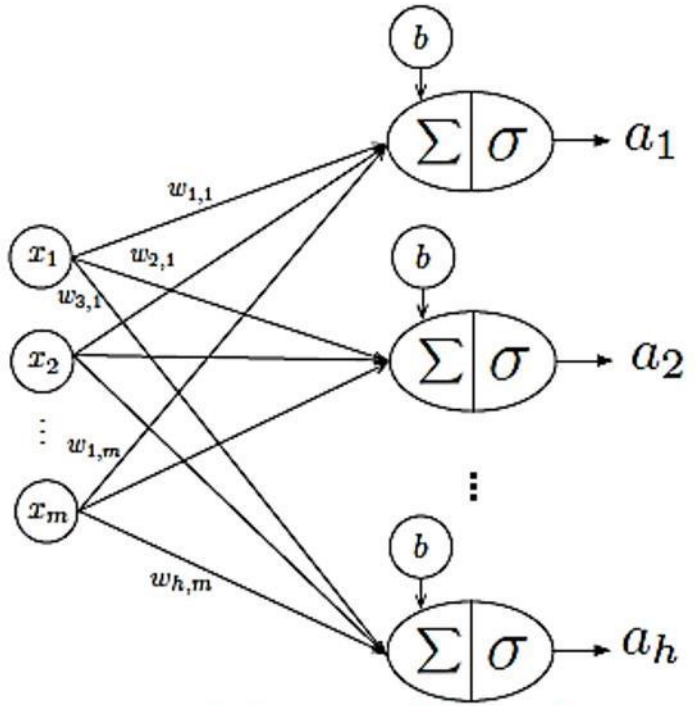
[5]:
```
Parameter containing:
tensor([-0.2552,  0.3918,  0.2693], requires_grad=True)
```

Capa totalmente conectada en PyTorch

[6]:
```
print('X dim:', X.size())
print('W dim:', fc_layer.weight.size())
print('b dim:', fc_layer.bias.size())
# .size() is equivalent to .shape
A = fc_layer(X)
print('A:', A)
print('A dim:', A.size())
```

```
X dim: torch.Size([10, 5])
W dim: torch.Size([3, 5])
b dim: torch.Size([3])
A: tensor([[ 1.2004,  2.3291,  2.0036],
        [ 4.5367,  7.7858,  5.4519],
        [ 7.8730, 13.2424,  8.9003],
        [11.2093, 18.6991, 12.3486],
        [14.5457, 24.1557, 15.7970],
        [17.8820, 29.6123, 19.2453],
        [21.2183, 35.0690, 22.6937],
        [24.5546, 40.5256, 26.1420],
        [27.8910, 45.9823, 29.5904],
        [31.2273, 51.4389, 33.0387]], grad_fn=<ThAddmmBackward>)
A dim: torch.Size([10, 3])
```

Basado en PyTorch, tenemos otra convención



Nota que *wi,j* se refiere al peso que conecta la entrada j-ésima con la salida i-ésima.

...

where $\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \cdots & w_{h,m} \end{bmatrix}$

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 \dots & x_m \end{bmatrix}$$

Activaciones de la capa para 1 ejemplo de entrenamiento

$$\sigma\left(\mathbf{x}\mathbf{W}^\top + \mathbf{b}\right) = \mathbf{a}$$

$$\mathbf{a} \in \mathbb{R}^{1 \times h}$$

Activaciones de la capa para n ejemplos de entrenamiento

$$\sigma\left(\mathbf{X}\mathbf{W}^\top + \mathbf{b}\right) = \mathbf{A}$$

$$\mathbf{W}^\top \in \mathbb{R}^{m \times h}$$

$$\mathbf{A} \in \mathbb{R}^{n \times h}$$

Puedes encontrar el código fuente aquí
https://github.com/pytorch/pytorch/blob/18edd3ab0828acaa81dc052dba8644c874dc62db/torch/nn/functional.py#L1368

Conclusión

- Piensa siempre en cómo se calculan los productos punto al escribir e implementar la multiplicación de matrices.
- La intuición teórica y la convención no siempre coinciden con la conveniencia práctica (al programar).
- Al cambiar entre la teoría y el código, estas reglas pueden ser útiles:

$$\mathbf{AB} = (\mathbf{B}^\top \mathbf{A}^\top)^\top$$

$$(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$$

Resumen: Tradicional vs PyTorch

(La matriz de transformación idealmente debería ir siempre al frente)

where $\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \cdots & w_{h,m} \end{bmatrix}$

ten en cuenta que wi,j se refiere al peso que conecta la entrada j con la salida i..

Activaciones de capa para 1 ejemplo de entrenamiento

$$\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{a} \;,\; \mathbf{a} \in \mathbb{R}^{h \times 1} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^{m \times 1}$$

$$\Leftrightarrow \sigma([\mathbf{x}^\top \mathbf{W}^\top]^\top + \mathbf{b}) = \mathbf{a} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^{m \times 1}$$

$$\Leftrightarrow \sigma([\mathbf{x}\mathbf{W}^\top] + \mathbf{b}) = \mathbf{a} \quad \text{with} \quad \mathbf{x} \in \mathbb{R}^{1 \times m} \quad \text{(PyTorch)}$$

Activaciones de capa para n ejemplos de entrenamiento

$$\sigma([\mathbf{W}\mathbf{X}^\top]^\top + \mathbf{b}) = \mathbf{A} \;,\; \mathbf{A} \in \mathbb{R}^{n \times h} \quad \text{with} \quad \mathbf{X} \in \mathbb{R}^{n \times m}$$

$$\Leftrightarrow \sigma([\mathbf{X}\mathbf{W}^\top] + \mathbf{b}) = \mathbf{A} \quad \text{with} \quad \mathbf{X} \in \mathbb{R}^{n \times m}$$

Ejercicio / Experimento de tarea no calificada

• Revisita nuestro código del perceptrón en NumPy: https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L03-perceptron/code/perceptron-numpy.ipynb

1. Sin ejecutar el código, ¿puedes decir si el perceptrón podría predecir las etiquetas de clase si alimentamos un arreglo de múltiples ejemplos de entrenamiento a la vez (es decir, mediante su método forward)?

    - ¿Sí? ¿Por qué?
    - ¿No? ¿Qué cambio necesitarías hacer?

2. Ejecuta el código para verificar tu intuición.

3. ¿Y qué hay del método train? ¿Podemos tener paralelismo a través de multiplicación matricial sin afectar la regla de aprendizaje del perceptrón?

Thank you all!