Redes Neuronales (Electivo)

Licenciatura en Física

Profesor: Graeme Candlish Semestre II 2023

1 Regresión Lineal

1.1 Introducción

Supongamos que tenemos datos que corresponden a dos variables para cada punto de datos (por ejemplo, masa y color de estrellas) y tenemos otra variable y que suponemos depende de estas (por ejemplo, temperatura). Los datos son $x_k^{(i)}$, donde k indica el vector de *features* y i indica la *instancia* (punto de datos).

Suponiendo un modelo lineal, podemos escribir

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \tag{1}$$

donde los parámetros del modelo son θ_k (también se llaman **pesos**). Si definimos $x_0 = 1$ podemos escribir

$$h(x) = \sum_{k=0}^{d} \theta_k x_k = \theta^T x \tag{2}$$

donde ahora representamos θ y x como vectores.

La pregunta central de *machine learning* es: ¿cómo elegimos los parámetros θ? Una respuesta es buscar θ tal que $h(x)$ aproxima muy bien a los valores de y. Entonces, hay que definir una función que mide, para cada θ_k, la "distancia" entre los $h(x^{(i)})$ y los $y^{(i)}$. Definimos la **función de coste**:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} [h_\theta(x^{(i)}) - y^{(i)}]^2 \tag{3}$$

Este es la función de **mínimos cuadrados**.

1.2 Algoritmo de LMS

Queremos elegir θ para minimizar $J(\theta)$. Usamos un algoritmo de búsqueda que comienza con una estimación inicial para θ, y que actualiza θ para reducir $J(\theta)$ hasta tenemos convergencia al mínimo.

Descenso por gradiente:

$$\theta_k := \theta_k - \alpha \frac{\partial}{\partial \theta_k} J(\theta). \tag{4}$$

El (hiper)parámetro α se llama la taza de aprendizaje. Necesitamos la derivada de la función $J(\theta)$. Como ejemplo, consideremos un solo punto de datos (x, y):

$$\begin{aligned}\frac{\partial}{\partial\theta_k}J(\theta) &= \frac{\partial}{\partial\theta_k}\frac{1}{2}(h_\theta(x)-y)^2 \\ &= 2\cdot\frac{1}{2}(h_\theta(x)-y)\cdot\frac{\partial}{\partial\theta_k}(h_\theta(x)-y) \\ &= (h_\theta(x)-y)\cdot\frac{\partial}{\partial\theta_k}\left(\sum_{j=0}^{d}\theta_j x_j - y\right) \\ &= (h_\theta(x)-y)x_k\end{aligned} \tag{5}$$

Para un sólo dato (de entrenamiento) este nos da

$$\theta_k \coloneqq \theta_k + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_k^{(i)}. \tag{6}$$

Esta regla se llama la regla de actualización LMS (least mean squares). Podemos ver que la actualización es proporcional al error. Tiene sentido: mayor error, mayor corrección.

1.2.1 Descenso de gradiente por lotes (batch)

Si tenemos más que un punto de datos:

$$\theta_k \coloneqq \theta_k + \alpha\sum_{i=1}^{n}(y^{(i)} - h_\theta(x^{(i)}))x_k^{(i)}. \tag{7}$$

En notación de vectores:

$$\theta \coloneqq \theta + \alpha\sum_{i=1}^{n}(y^{(i)} - h_\theta(x^{(i)}))x^{(i)}. \tag{8}$$

Este corresponde a **batch gradient descent**: en cada paso del algoritmo, usamos todos los datos (de entrenamiento).

Un problema del algoritmo es el riesgo de encontrar mínimos **locales**. En el caso de regresión lineal, hay solamente un mínimo global.

1.2.2 Descenso de gradiente estocástico

Podemos actualizar los pesos considerando los puntos de datos uno por uno:

$$\begin{aligned}&\text{for i=1 to n} \\ &\theta \coloneqq \theta + \alpha(y^{(i)} - h_\theta(x^{(i)}))x^{(i)}\end{aligned} \tag{9}$$

El algoritmo oscilará alrededor del mínimo, pero puede avanzar más rápidamente que descenso de gradiente por lote. Si hay muchos datos, la versión estocástica probablemente será la única opción.

1.3 Las ecuaciones normales

Ahora vamos a minimizar J directamente, sin usar un algoritmo iterativo. Hay que introducir notación.

1.3.1 Derivadas matriciales

Para un función $f : \mathbb{R}^{n\times d} \to \mathbb{R}$ que mapea matrices de $n \times d$ elementos a números reales, definimos la derivada de f con respecto a A por

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nd}} \end{bmatrix} \tag{10}$$

Para dar un ejemplo, consideremos

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{11}$$

y suponemos una función $f(A) = \frac{3}{2}A_11 + 5A_{12}^2 + A_{21}A_{22}$. El gradiente sería

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10A_{12} \\ A_{22} & A_{21} \end{bmatrix} \tag{12}$$

1.3.2 Revisitando mínimos cuadrados

Dado un conjunto de datos (de entrenamiento), consideramos la matriz X (matriz de diseño) donde cada fila es un vector de *features*. Así que la matriz tiene $n \times (d+1)$ elementos (el $+1$ es por incluir el intercepto).

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(n)})^T \end{bmatrix} \tag{13}$$

Definimos el vector de blanco (valores que queremos predecir):

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \tag{14}$$

Podemos escribir la ecuación que define el modelo lineal como

$$\begin{aligned} X\theta - y &= \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \vdots \\ (x^{(n)})^T\theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \\ &= \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(n)}) - y^{(n)} \end{bmatrix} \end{aligned} \tag{15}$$

Ya que $z^T z = \sum_i z_i^2$ podemos escribir

$$\begin{aligned} \frac{1}{2}(X\theta - y)^T(X\theta - y) &= \frac{1}{2}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= J(\theta) \end{aligned} \tag{16}$$

Para minimizar J, calculamos el gradiente con respecto a θ:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - y)^T(X\theta - y) \\
&= \frac{1}{2}\nabla_\theta((X\theta)^T X\theta - (X\theta)^T y - y^T(X\theta) + y^T y) \\
&= \frac{1}{2}\nabla_\theta(\theta^T(X^T X)\theta - y^T(X\theta) - y^T(X\theta)) \\
&= \frac{1}{2}\nabla_\theta(\theta^T(X^T X)\theta - 2(X^T y)^T\theta) \\
&= \frac{1}{2}(2X^T X\theta - 2X^T y) \\
&= X^T X\theta - X^T y
\end{aligned} \tag{17}$$

Exigiendo que el gradiente sea cero, tenemos

$$X^T X\theta = X^T y \tag{18}$$

Podemos obtener el valor de θ que minimiza $J(\theta)$:

$$\theta = (X^T X)^{-1} X^T y \tag{19}$$

donde suponemos que la matriz $X^T X$ es invertible. Si $n < d$ (i.e. hay menos datos que *features*) o los *features* no son linealmente independientes, la matriz no tiene una inversa. Pero en estos casos se puede trabajar con la **pseudoinversa** de Moore-Penrose.

2 Regresión logística

Para clasificación binaria queremos tener un resultado de 0 (no pertenece a la clase) o 1 (si pertenece). Podemos obtener un resultado del modelo donde $y \in [0, 1]$ por el uso de la función sigmoide:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + exp(-\theta^T x)} \tag{20}$$

[GRAPH THIS IN PYTHON] La derivada de la función sigmoide es bastante simple:

$$\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}(e^{-z}) \\
&= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\
&= g(z)(1 - g(z))
\end{aligned} \tag{21}$$

Vamos a interpretar el resultado del modelo como una probabilidad:

$$\begin{aligned}
P(y = 1|x; \theta) &= h_\theta(x) \\
P(y = 0|x; \theta) &= 1 - h_\theta(x)
\end{aligned} \tag{22}$$

Podemos escribir este en una forma compacta:

$$p(y|x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}. \tag{23}$$

Suponiendo que todas las instancias de entrenamiento estaban generadas independientemente:

$$\begin{aligned} L(\theta) &= p(\vec{y}|X;\theta) \\ &= \prod_{i=1}^{n} p(y^{(i)}|x^{(i)};\theta) \\ &= \prod_{i=1}^{n} (h_\theta(x^{(i)}))^{y^{(i)}} (1-h_\theta(x^{(i)}))^{1-y^{(i)}} \end{aligned} \tag{24}$$

Es más fácil **maximizar** el logaritmo de esa probabilidad:

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{n} y^{(i)} \log h(x^{(i)}) + (1-y^{(i)}) \log\Big(1 - h(x^{(i)})\Big) \tag{25}$$

Aplicamos **ascenso** por gradiente (ya que estamos maximizando la función) usando $\theta := \theta + \alpha \nabla_\theta \updownarrow(\theta)$.

Aquí usamos un solo dato de entrenamiento (x, y):

$$\begin{aligned} \frac{\partial}{\partial \theta_j}\ell(\theta) &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)\frac{\partial}{\partial \theta_j} g(\theta^T x) \\ &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right) g(\theta^T x)(1-g(\theta^T x))\frac{\partial}{\partial \theta_j}\theta^T x \\ &= (y(1-g(\theta^T x)) - (1-y)g(\theta^T x))x_j \\ &= (y - h_\theta(x))x_j \end{aligned} \tag{26}$$

donde hemos usado $g'(z) = g(z)(1-g(z))$. Así que la regla de actualización de los pesos es:

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)} \tag{27}$$

Este es muy parecida a lo que vimos para la regla de actualización para regresión lineal.

3 Redes neuronales

3.1 Red de dos capas, completamente conectadas

Una red con m unidades escondidas, y un vector de entrada de d elementos, $x \in \mathbb{R}^d$.

$$z_j = w_j^{[1]T} x + b_j^{[1]} \quad \text{donde} \quad w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \tag{28}$$

donde $j = 1, \ldots, m$. Las salidas de la primera capa son

$$a_j = \sigma(z_j) \quad \text{donde} \quad a = [a_1, \ldots, a_m]^T \in \mathbb{R}^m \tag{29}$$

La salida de la red entera es

$$h_\theta(x) = w^{[2]T} a + b^{[2]} \quad \text{donde} \quad w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R}. \tag{30}$$

3.2 Notación vectorial

Ahora escribimos la red de dos capas en forma vectorial. Primero, definimos una matriz de pesos $W^{[1]} \in \mathbb{R}^{m \times d}$:

$$W^{[1]} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ \vdots \\ w_m^{[1]T} \end{bmatrix} \in \mathbb{R}^{m \times d} \tag{31}$$

Así que podemos escribir

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ \vdots \\ w_m^{[1]T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} b_1^{[1]1} \\ b_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix} \tag{32}$$

donde $z \in \mathbb{R}^{m \times 1}$, $W^{[1]} \in \mathbb{R}^{m \times d}$, $x \in \mathbb{R}^{d \times 1}$ y $b^{[1]} \in \mathbb{R}^{m \times 1}$. En la forma más compacta:

$$z = W^{[1]}x + b^{[1]} \tag{33}$$

La aplicación de la función de activación se escribe así:

$$a = \sigma(z) \tag{34}$$

Definimos $W^{[2]} = w^{[2]T} \in \mathbb{R}^{1 \times m}$. Entonces, el modelo completo es

$$h_\theta(x) = W^{[2]}(\sigma(W^{[1]}x + b^{[1]})) + b^{[2]} \tag{35}$$

Los parámetros θ son los $W^{[1]}$, $W^{[2]}$ (matrices de pesos) y los $b^{[1]}$, $b^{[2]}$ (vectores de sesgos [*biases*]).

3.3 Redes de multi-capa

Supongamos que hay r capas en la red. La salida de cada capa está dada por

$$\begin{aligned} a^{[1]} &= \sigma(W^{[1]}x + b^{[1]}) \\ a^{[2]} &= \sigma(W^{[2]}a^{[1]} + b^{[2]}) \\ &\cdots \\ a^{[r-1]} &= \sigma(W^{[r-1]}a^{[r-2]} + b^{[r-1]} \\ h_\theta(x) &= W^{[r]}a^{[r-1]} + b^{[r]}. \end{aligned} \tag{36}$$

Si $a^{[k]}$ tiene dimensión m_k, la matriz $W^{[k]}$ es $m_k \times m_{k-1}$, y el *bias* $b^{[k]} \in \mathbb{R}^{m_k}$. Además, $W^{[1]} \in \mathbb{R}^{m_1 \times d}$ y $W^{[r]} \in \mathbb{R}^{1 \times m_{r-1}}$.

El número total de neuronas es $m_1 + \cdots + m_r$, y el número total de parámetros es $(d+1)m_1 + (m_1+1)m_2 + \cdots + (m_{r-1}+1)m_r$.

3.4 Otras funciones de activación

- ReLU(z): $\max(wx + b, 0)$

- $\tanh(z)$
- Leaky ReLU: $\max\{z, \gamma z\}$, $\gamma \in (0, 1)$.
- GELU: $\frac{z}{2}\left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right]$

3.5 Fuente de la no-linealdad de las redes neuronales

Supongamos que $\sigma(z) = z$, etonces tenemos

$$\begin{aligned} h_\theta(x) &= W^{[2]} a^{[1]} \\ &= W^{[2]} \sigma(z^{[1]}) \\ &= W^{[2]} z^{[1]} \\ &= W^{[2]} W^{[1]} x \\ &= \tilde{W} x \end{aligned} \tag{37}$$

El modelo es lineal. Así que, es solamente con funciones de activación no lineal que podemos obtener un modelo no lineal.

3.6 Capas de normalización

A veces es necesario normalizar la salida de una capa, z. Hay varias opciones. Una se llama *layer norm*:

$$\text{LN}(z) = \begin{bmatrix} \beta + \gamma \left(\frac{z_1 - \hat{\mu}}{\hat{\sigma}}\right) \\ \beta + \gamma \left(\frac{z_2 - \hat{\mu}}{\hat{\sigma}}\right) \\ \vdots \\ \beta + \gamma \left(\frac{z_m - \hat{\mu}}{\hat{\sigma}}\right) \end{bmatrix} \tag{38}$$

donde β y γ son parámetros entrenables, $\hat{\mu} = \sum z_i / m$ es el promedio, $\hat{\sigma} = \sqrt{\sum (z_i - \hat{\mu}^2)/m}$ es la desviación estandar.

Otra posibilidad se llama *normalización por lote*, donde el promedio y desviación estandar están calculados sobre cada lote.

3.7 Capas convolucionales

Una capa convolucional aplica un filtro a las entradas. Es típicamente usada en procesamiento de imagenes. La idea básica está mostrada en la Fig. 1.

El *kernel* puede ser aplicada en cada pixel, o podemos saltar pixeles (se llama *stride*).

Después del filtro, aplicamos una función de activación (a veces llamada un *feature map*) para introducir no-linealidad en el modelo.

Aplicando el *kernel* directamente a la imagen resulta en una reducción de su tamaño. Para evitar ese podemos agregar pixeles falsas alrededor de la imagen. Eso se llama *padding* (ver Fig. 2).

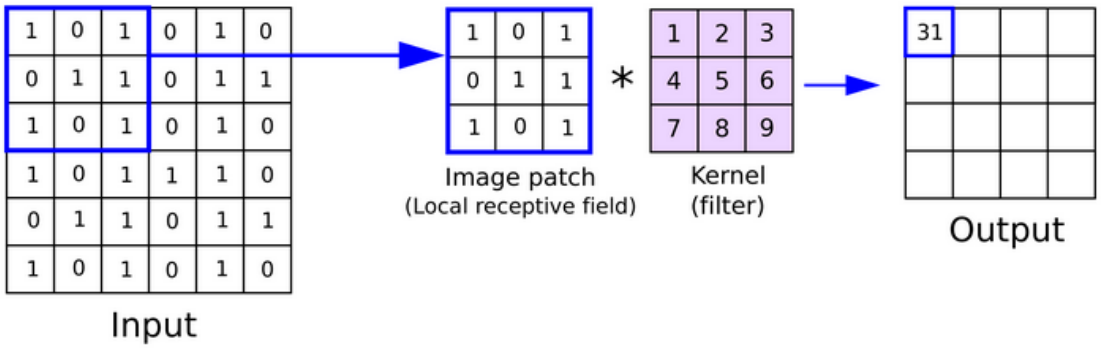

Figure 1: Idea básica de una capa convolucional

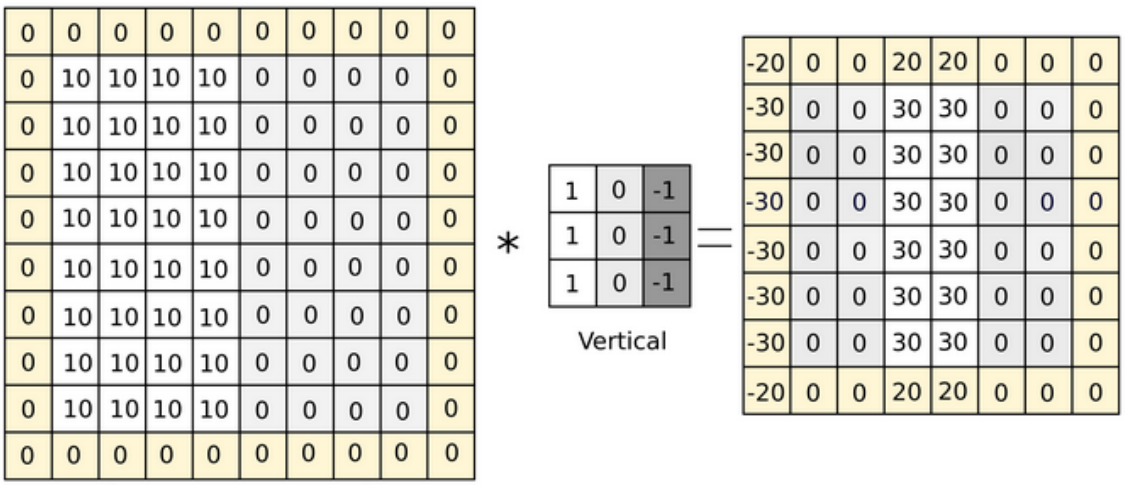

Figure 2: *Padding*

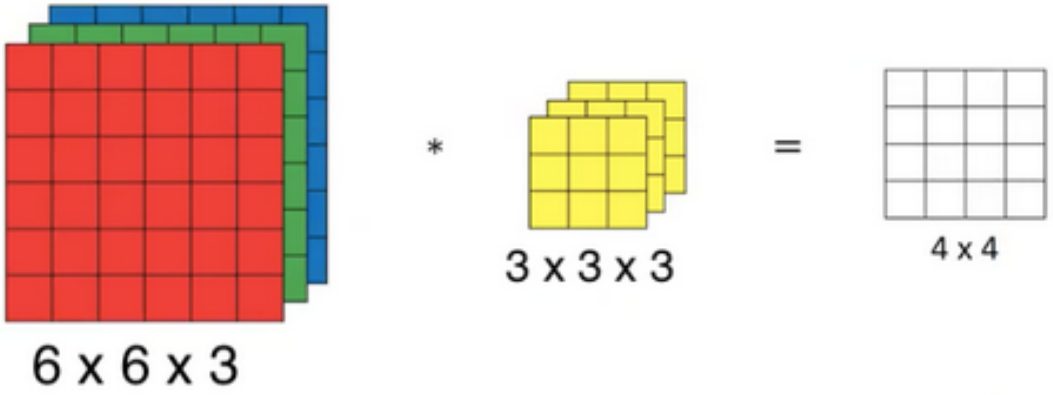

Figure 3: *3 canales de color*

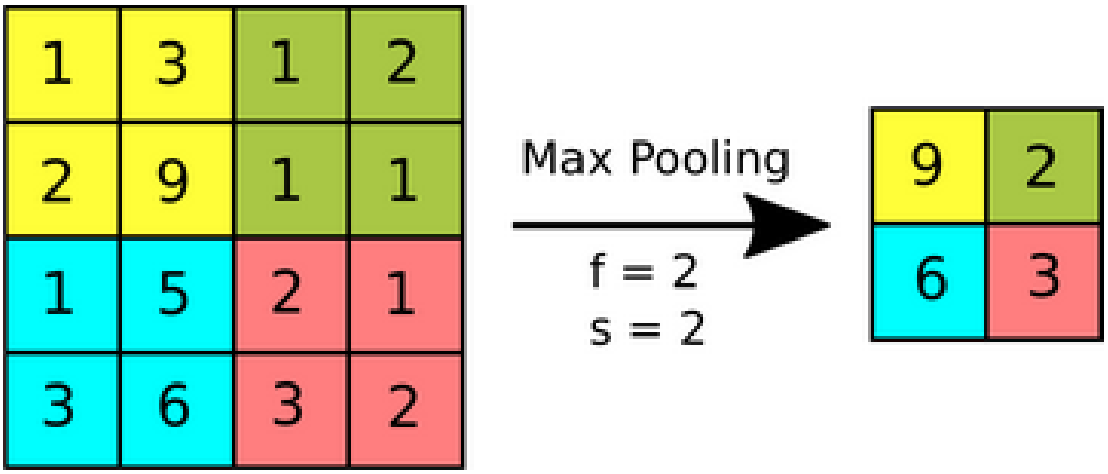

Figure 4: *Una capa de max pooling*

Las imagenes tienen 3 canales de color, así que podemos aplicar filtros de 3 dimensiones para reducir la dimensionalidad (ver Fig. 3).

También, para reducir la dimensionalidad, tenemos capas de *pooling* (ver Fig. 4).

Un ejemplo de una red convolucional conectada al final a una red totalmente conectada (*feed forward*) llegando a una función *softmax* para clasificación (ver Fig. 5).

Se puede expresar todas las operaciones de una red convolucional en términos de matrices. Así que, de nuevo las operaciones de la red son operaciones matriciales.

Finalmente, en términos de la arquitectura de la red, podemos interpretar los filtros como conexiónes entre neuronas en la misma capa.

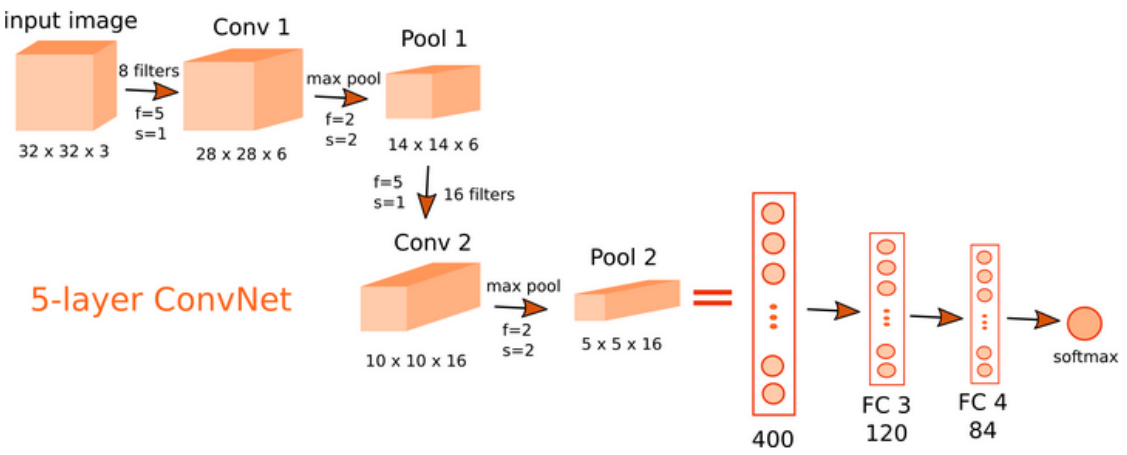

Figure 5: ConvNet

3.8 Redes recurrentes

Esta arquitectura es para el analisis de secuencias de datos. Ejemplos:

- Precios de acciones
- Traducción automática
- Generación de texto (*large language models*).

Dificultades con RNNs:

- Gradientes inestables (se puede aliviar con *dropout* o normalización.
- Una memoria muy limitada (se puede aliviar con celdas de LSTM/GRU)

3.9 Ejemplo: una sola neurona recurrente

En la Fig. 6 vemos una sola neurona recurrente, desenrollada en el tiempo.

3.10 Una capa recurrente

En la Fig. 7 vemos una capa recurrente, desenrollada en el tiempo. Cada *timestep* la capa recibe el vector de entrada, y el vector de salida del *timestep* anterior.

La salida de una capa recurrente se puede calcular así:

$$\vec{y}_{(t)} = \sigma\left(W_x^T \vec{x}_{(t)} + W_y^T \vec{y}_{(t-1)} + \vec{b}\right) \tag{39}$$

donde σ es la función de activación, y las matrices W_x, W_y contienen los pesos de la capa.

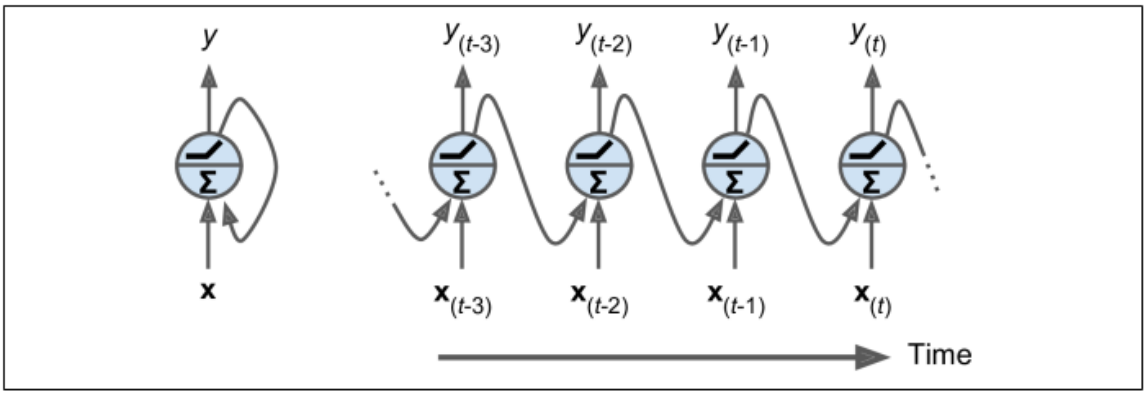

Figure 6: Una sola neurona recurrente, desenrollada en el tiempo.

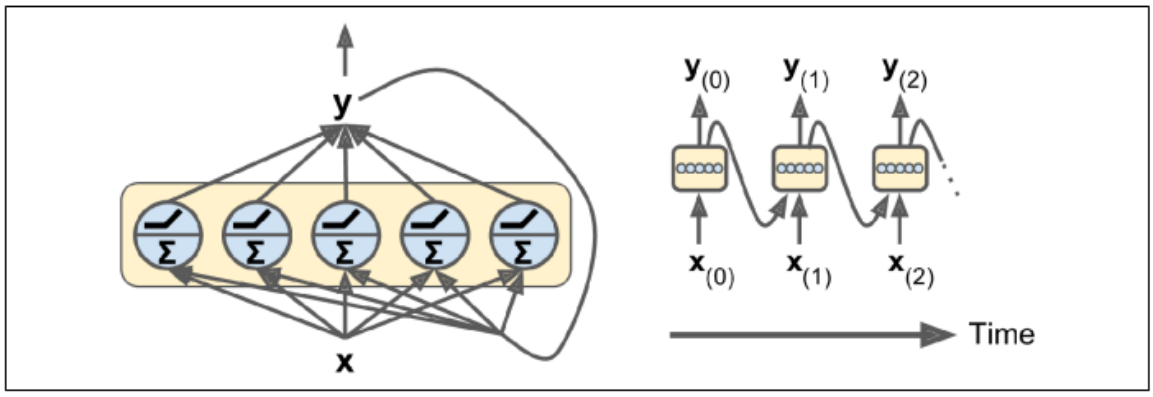

Figure 7: Una capa recurrente, desenrollada en el tiempo.

La salida de la capa para todas las instancias en un mini-lote sería

$$Y_{(t)} = \sigma\left(X_{(t)}W_x + Y_{(t-1)}W_y + \vec{b}\right) \tag{40}$$

donde

- $Y_{(t)}$ es una matriz $m \times n_{\text{neuronas}}$ que contiene las salidas de la capa en momento t para cada instancia en el mini-lote.
- $X_{(t)}$ es una matriz $m \times n_{\text{entradas}}$ que contiene todas las entradas para todas las instancias.
- W_x es una matriz $n_{\text{entradas}} \times n_{\text{neuronas}}$.
- $W_y$es una matriz $n_{\text{neuronas}} \times n_{\text{neuronas}}$.
- $\vec{b}$ es el vector de sesgos (n_{neuronas} elementos).

Las salidas son funciones de todos los momentos anteriores (es decir, $t-1$, $t-2$, $t-3$, etc.)

3.11 Celdas de memoria

Una red que mantiene un estado a través del tiempo tiene **memoria** y se llama **celda de memoria** (o celda).

El estado de una celda en momento t es una función de su estado anterior y la entrada actual: $h_{(t)} = f(h_{(t-1)}, x_{(t)})$.

3.12 Entrenamiento de RNNs

El truco es desenrollar la red en el tiempo, y aplicar *backpropagation* (*backpropagation through time*, ver Fig. 8).

3.13 LSTM

Hay perdida de información en cada timestep de una RNN tradicional: eventualmente el estado de la red no tiene "memoria" de las entradas originales.

Para resolver este problema, en 1997 investigadores inventaron la celda de larga memoria corta (*Long Short-Term Memory (LSTM) cell*). La arquitectura de la celda está en la Fig. 9.

La celda tiene un **estado interno** y 3 "puertas" (*gates*):

1. Puerta de entrada: determina si la entrada tiene impacto o no en el estado interno.
2. Puerta de olvidar: determina si el estado interno debe ser borrado o no.
3. Puerta de salida: determina si el estado interno debe impactar la salida o no.

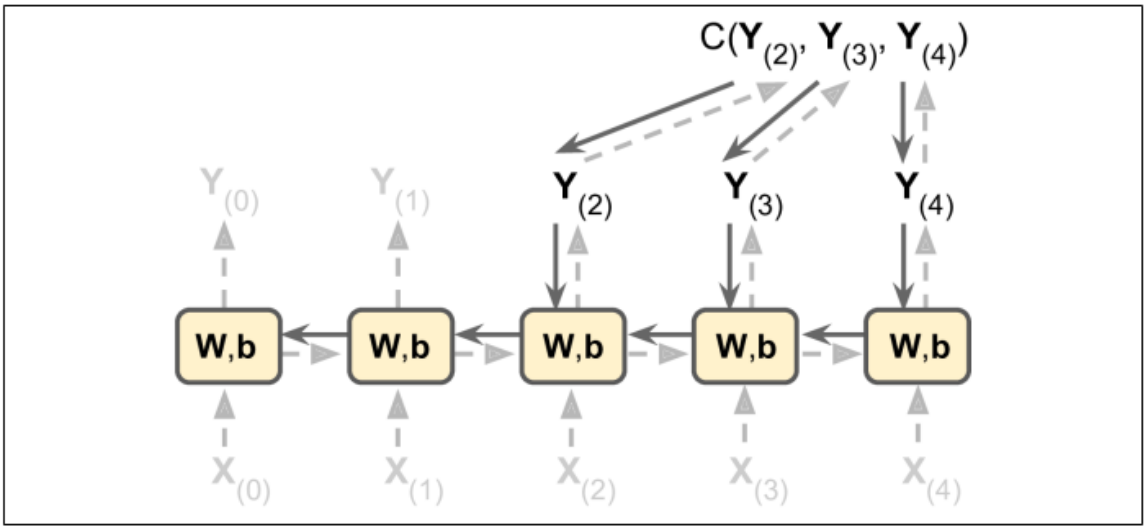

Figure 8: Backpropagation en una red RNN desenrollada en el tiempo.

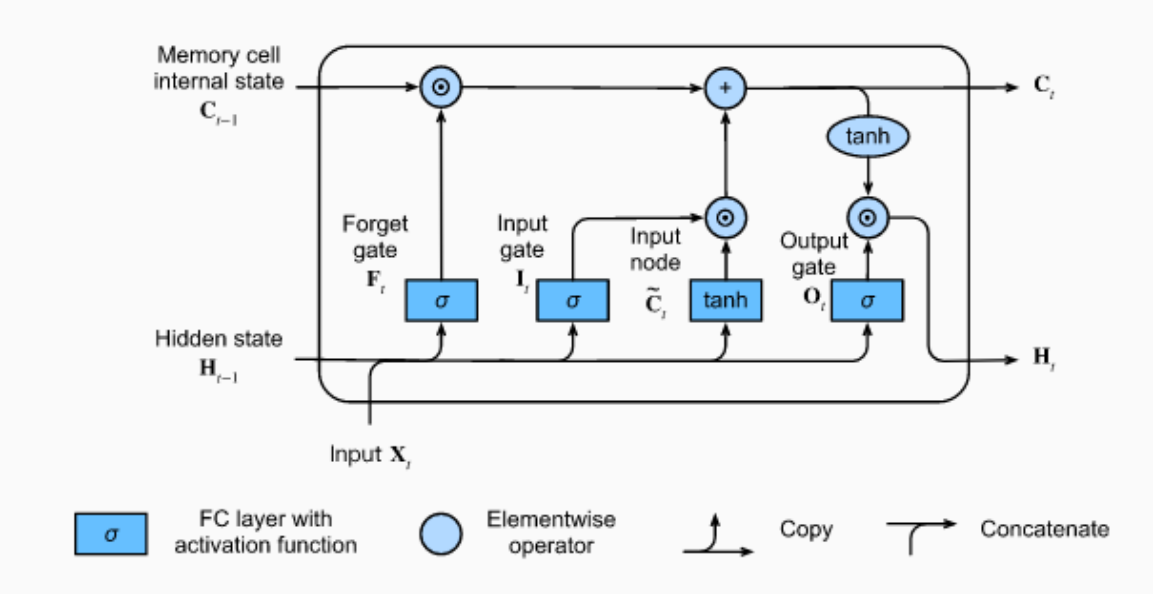

Figure 9: Celda de LSTM.

3.13.1 Las puertas

Las entradas de la celda son la entrada en el momento actual t y el estado interno del momento $t-1$. Estos valores están pasados a cada puerta de la celda.

Las puertas tienen capas totalmente conectadas (*fully connected*) con funciones de activación sigmoides (las salidas $\in [0,1]$).

También hay un nodo de entrada con función de activación tanh (rango $[-1,1]$).

Supongamos que hay h unidades escondidas, tamaño de lote n y d entradas.

- Entradas: $X_t \in \mathbb{R}^{n\times d}$
- Unidades escondidas: $H_{t-1} \in \mathbb{R}^{n\times h}$.
- Puerta de entrada: $I_t \in \mathbb{R}^{n\times h}$
- Puerta de olvidar: $F_t \in \mathbb{R}^{n\times h}$
- Puerta de salida: $O_t \in \mathbb{R}^{n\times h}$

Calculamos los valores de cada puerta así:

$$\begin{aligned} I_t &= \sigma(X_t W_{xi} + H_{t-1} W_{hi} + \vec{b}_i) \\ F_t &= \sigma(X_t W_{xf} + H_{t-1} W_{hf} + \vec{b}_f) \\ O_t &= \sigma(X_t W_{xo} + H_{t-1} W_{ho} + \vec{b}_o) \end{aligned} \tag{41}$$

donde $W_{xi}, W_{xi}, W_{xi} \in \mathbb{R}^{d\times h}$ y $W_{hi}, W_{hi}, W_{hi} \in \mathbb{R}^{h\times h}$ son los pesos y $\vec{b}_i, \vec{b}_f, \vec{b}_o$ son los sesgos. Las funciones de activación σ son sigmoides.

3.13.2 Nodo de entrada

El nodo de entrada es $\tilde{C}_t \in \mathbb{R}^{n\times h}$ y se calcula como

$$\tilde{C}_t = \tanh\Big(X_t W_{xc} + H_{t-1} W_{hc} + \vec{b}_c\Big) \tag{42}$$

donde $W_{xc} \in \mathbb{R}^{d\times h}$ y $W_{hc} \in \mathbb{R}^{h\times h}$ son pesos y $\vec{b} \in \mathbb{R}^{1\times h}$ es el vector de sesgo.

3.13.3 Estado interno

El estado interno de la celda está dado por

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t \tag{43}$$

donde $\odot$ indica el producto elemento-por-elemento (producto Hadamard).

Si la puerta de olvidar siempre es 1, y la de entrada siempre es 0, el estado interno será constante.

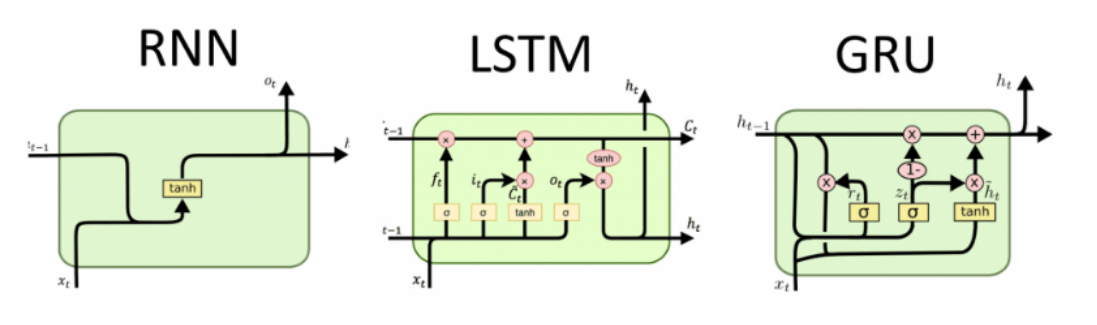

Figure 10: Comparación de RNN, LSTM y GRU.

3.13.4 Estado escondido

La salida de la celda $H_t \in \mathbb{R}^{n \times h}$ (lo que las otras capas en la red ven) está dada por

$$H_t = O_t \odot \tanh(C_t) \tag{44}$$

Se puede interpretar H_t como el estado de plazo corto en la celda, y C_t como el estado de plazo largo.

3.14 Celda GRU

GRU significa *gated recurrent unit*. Es una simplificación del LSTM (ver la comparación de RNN, LSTM y GRU en la Fig. 10).

Los cálculos necesarios son:

$$\begin{aligned} Z_t &= \sigma(X_t W_{xz} + H_{t-1} W_{hz} + \vec{b}_z) \\ R_t &= \sigma(X_t W_{xr} + H_{t-1} W_{hr} + \vec{b}_r) \\ \tilde{C}_t &= \tanh\Big(X_t W_{xg} + (R_t \odot H_{t-1}) W_{hg} + \vec{b}_g\Big) \\ H_t &= Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{C}_t \end{aligned} \tag{45}$$

Diferencias entre LSTM y GRU:

- Ambos vectores de estado están combinados en un solo vector H_t.
- Una sola puerta Z_t reemplaza las puertas de olvidar y entrada.
- No hay puerta de salida, pero hay una nueva puerta R_t que determina cual parte del estado previo contribuye al estado $\tilde{C}_t$

Las celdas LSTM/GRU todavía tienen problemas trabajando con series de datos muy largos.

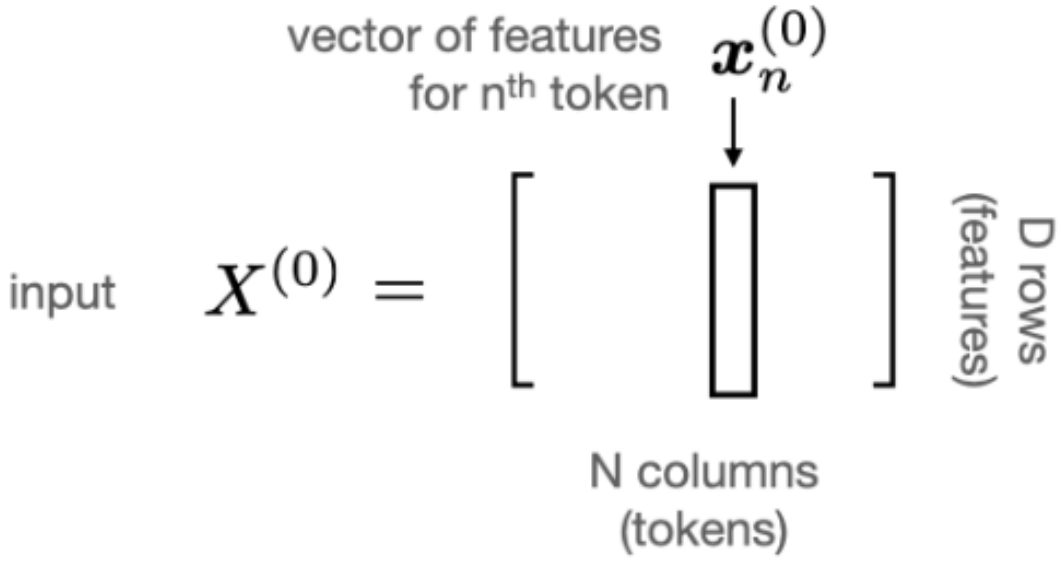

Figure 11: Entradas a un transformador.

3.15 Atención y transformadores

El uso de transformadores (que incluyen el concepto de **atención**) ha revolucionado el mundo de NLP en los últimos años. Varios modelos de tipo LLM (como ChatGPT) son transformadores. Veremos una definición técnica por ahora.

3.15.1 Formato de entrada

Los datos deben ser una secuencia de N fichas (tokens) $\boldsymbol{x}_n^{(0)}$ de dimensión D (número de *features*). Organizamos la secuencia en una matriz $X^{(0)}$ $D \times N$ (ver Fig. 11).

Este podría ser una secuencia de palabras *embedded* (incorporado) en vectores.

Hay dos etapas principales en un transformador:

1. *Multi-head self-attention* (procesando fichas)
2. MLP (procesando *features*)

3.15.2 Atención

La salida de la primera etapa es otra matriz $D \times N$ $Y^{(m)}$. El vector de salida en ubicación n, $\boldsymbol{y}_n^{(m)}$ esta dado por

$$\boldsymbol{y}_n^{(m)} = \sum_{n'=1}^{N} \boldsymbol{x}_{n'}^{(m)} A_{n',n}^{(m)}. \tag{46}$$

La matriz A se llama la matriz de **atención**, de $N \times N$, y normalizada por columna $\sum_{n'=1}^{N} A_{n',n}^{(m)} = 1$.

La idea es que A toma valores altos para elementos donde la ficha n' está muy relacionada con la ficha n.

Entonces, tenemos

$$Y^{(m)} = X^{(m-1)} A^{(m)} \tag{47}$$

3.15.3 Auto-atención

¿De dónde viene la matriz de atención? La idea de un transformador es que podemos generar esa matriz de la secuencia de entrada: auto-atención.

Una forma simple sería medir la similitud entre dos fichas por el uso del producto punto de sus vectores de *features*:

$$A_{n,n'} = \frac{\exp\left(\boldsymbol{x}_n^T \boldsymbol{x}_{n'}\right)}{\sum_{n''=1}^{N} \exp\left(\boldsymbol{x}_{n''}^T \boldsymbol{x}_{n'}\right)} \tag{48}$$

Pero el problema es que este método mezcla información sobre similitudes en la secuencia con el contenido de la misma secuencia.

Como alternativo, usamos una transformada lineal $U\boldsymbol{x}_n$:

$$A_{n,n'} = \frac{\exp\left(\boldsymbol{x}_n^T U^T U \boldsymbol{x}_{n'}\right)}{\sum_{n''=1}^{N} \exp\left(\boldsymbol{x}_{n''}^T U^T U \boldsymbol{x}_{n'}\right)} \tag{49}$$

La desventaja aquí es que el numerador es simétrica. Quizás queremos asociar una palabra rara con una muy común, pero no al revés.

Podemos generalizar con usar dos transformadas lineales:

$$A_{n,n'} = \frac{\exp\left(\boldsymbol{x}_n^T U_{\boldsymbol{q}}^T U_{\boldsymbol{k}} \boldsymbol{x}_{n'}\right)}{\sum_{n''=1}^{N} \exp\left(\boldsymbol{x}_{n''}^T U_{\boldsymbol{q}}^T U_{\boldsymbol{k}} \boldsymbol{x}_{n'}\right)}. \tag{50}$$

El vector $\boldsymbol{q}_n = U_{\boldsymbol{q}} \boldsymbol{x}_n$ corresponde a los *queries* (consultas), y el vector $\boldsymbol{k}_n = U_{\boldsymbol{k}} \boldsymbol{x}_n$ corresponde a las *keys* (llaves).

Las matrices $U_{\boldsymbol{q}}$ y $U_{\boldsymbol{k}}$ son $K \times D$, donde K es otro parámetro.

3.15.4 *Multi-head self-attention (MHSA)*

En el cálculo arriba hay una sola matriz de atención para describir la similitud entre fichas. Sería útil tener más flexibilidad (similitud en distintas "dimensiones").

Un transformador aplica el mecanismo de arriba H veces (se llaman cabezas (*heads*)), y después proyecta a $D \times N$:

$$Y^{(m)} = \mathrm{MHSA}_\theta(X^{(m-1)} = \sum_{h=1}^{H} V_h X^{(m-1)} A_h^{(m)} \tag{51}$$

donde las H matrices V_h son $D \times D$. Las H matrices de atención son

$$[A_h^{(m)}]_{n,n'} = \frac{\exp\left((\boldsymbol{q}_{h,n}^{(m)})^T \boldsymbol{k}_{h,n'}^{(m)}\right)}{\sum_{n''=1}^{N} \exp\left((\boldsymbol{q}_{h,n''}^{(m)})^T \boldsymbol{k}_{h.n}^{(m)}\right)} \tag{52}$$

donde

$$\boldsymbol{q}_{h,n}^{(m)} = U_{\boldsymbol{q},h}^{(m)} \boldsymbol{x}_n^{(m-1)} \qquad \boldsymbol{k}_{h,n}^{(m)} = U_{\boldsymbol{k},h}^{(m)} \boldsymbol{x}_n^{(m-1)} \tag{53}$$

El proceso completo contiene los parámetros $\theta = U_{\boldsymbol{q},h}, U_{\boldsymbol{k},h}, V_h{}_{h=1}^{H}$, es decir $3H$ matrices de tamaños $K \times D$, $K \times D$ y $D \times D$ respectivamente.

3.15.5 MLP a lo largo de los *features*

Aplicamos un modelo MLP al vector de *features* en cada ficha n en la secuencia:

$$\boldsymbol{x}_n^{(m)} = \text{MLP}_\theta(\boldsymbol{y}_n^{(m)}) \tag{54}$$

Los parámetros θ del MLP son iguales para todo n. Típicamente tienen 1 o 2 capas ocultas, con dimensión igual al número de *features* (o mayor).

3.15.6 Bloque de transformador

Para conectar MHSA con MLP, usamos dos construcciones más: conexiones residuales y normalización de capa (LayerNorm).

La idea de una conexión residual es aplicar una función combinado con el mapeo de identidad:

$$x^{(m)} = x^{(m-1)} + \text{res}_\theta(x^{(m-1)}). \tag{55}$$

Normalización ya hemos visto:

$$\overline{x}_{d,n} = \frac{1}{\sqrt{\text{var}(\boldsymbol{x}_d)}}(x_{d,n} - \text{mean}(\boldsymbol{x}_d)) = \text{LayerNorm}(X)_{d,n} \tag{56}$$

donde $\text{mean}(\boldsymbol{x}_d) = (1/N)\sum_{n=1}^{N} x_{d,n}$ y $\text{var}(\boldsymbol{x}_d) = (1/N)\sum_{n=1}^{N}(x_{d,n} - \text{mean}(\boldsymbol{x}_d))^2$

Entonces, la arquitectura de un bloque del transformador esta dada en Fig. 12.

3.15.7 Información de posición

Hasta ahora hemos tratado los datos como un conjunto sin orden. En predicción de texto (como ChatGPT) hay que distinguir entre la frase "gatos comen peces" y "peces comen gatos".

Entonces, hay que incluir información sobre la posición en la ficha. Una posibilidad es incluir la posición en el *embedding* en alguna forma (hay varias opciones).

3.15.8 Predicción de la próxima palabra

Esta aplicación de un transformador se llama *auto-regressive language modelling*. La meta es predecir la próxima palabra en la secuencia dada las palabras previas $w_{1:n-1}$.

$$X_n^{(m)} = Y_n^{(m)} + \mathrm{MLP}(\bar{Y}_n^{(m)})$$

$$\bar{Y}^{(m)} = \mathrm{LayerNorm}(Y^{(m)})$$

$$Y^{(m)} = X^{(m-1)} + \mathrm{MHSA}(\bar{X}^{(m-1)})$$

$$\bar{X}^{(m-1)} = \mathrm{LayerNorm}(X^{(m-1)})$$

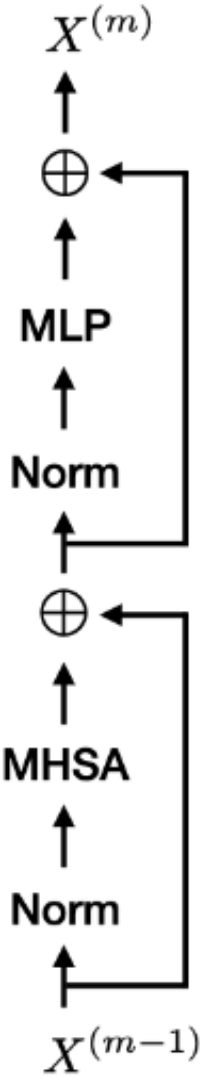

Figure 12: Un bloque de un transformador.

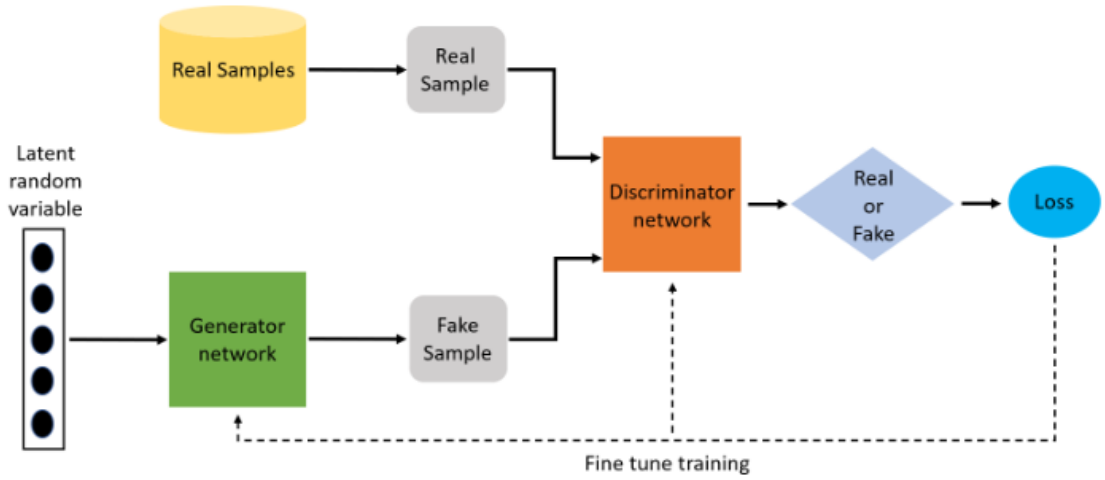

Figure 13: Arquitectura básica de una GAN.

Aplicamos el transformador M veces a la secuencia de entrada. Consideramos la representación en ficha $n-1$ ($\boldsymbol{x}_{n-1}^{(M)}$) y generamos la probabilidad de la próxima palabra con softmax:

$$p(w_n = w|w_{1:n-1}) = p(w_n = w|\boldsymbol{x}_{n-1}^{(M)}) = \frac{\exp\left(\boldsymbol{g}_w^T \boldsymbol{x}_{n-1}^{(M)}\right)}{\sum_{w=1}^{W} \exp\left(\boldsymbol{g}_w^T \boldsymbol{x}_{n-1}^{(M)}\right)} \tag{57}$$

donde W es el tamaño del vocabulario (todas las palabras posibles(!)), el w-esima palabra es w y $\boldsymbol{g}_{w\,w=1}^{W}$ son los pesos del softmax.

El entrenamiento del transformador en esta aplicación parece muy costoso: predecimos la primera palabra, pasamos esta a través del transformador y predecimos la próxima, pasamos estas dos a través del transformador y predecimos la próxima, etc. Resulta en N aplicaciones del transformador.

Un truco para evitar eso, es aplicar el transformador a la secuencia completa, y usar *masking* para evitar que palabras futuras afecten la representación de las palabras que aparecen antes. Se puede implementar eso con $A_{n,n'} = 0$ cuando $n > n'$ (matriz triangular superior).

3.16 Redes generativas adversarias

En una GAN (*generative adversarial network*) hay dos redes en una competencia: el generador intenta crear datos sintéticos, y el discriminador intenta identificar cuales datos son reales y cuales son sintéticos (ver Fig. 13).

La clave para una GAN es la función de coste. Es un juego de *minmax*. El algoritmo es lo siguiente:

for numero de iteraciones de entrenamiento **do** **for** k pasos **do**

- https://arxiv.org/pdf/1406.2661.pdf

4 Backpropagation

[USE MLP AS EXAMPLE TO EXPLAIN]

5 Autodiff

[Use this: https://github.com/ageron/handson-ml3/blob/main/extra_autodiff.ipynb]

6 TensorFlow

[Work through beginning of TF guide]

7 Keras, construcción de modelos

[Quick intro to Keras API, as described in the book]

8 Ejemplo básico: MNIST

8.1 Con una red *feed-forward*

8.2 Con una red convolucional

9 Proyecto 1: AirCalc

10 Proyecto 2: clasificador de actividades (RNN)

11 Proyecto 3: envejecimiento de fotos

12 PINNs