

Clase 1: Introducción al aprendizaje profundo

Inteligencia Artificial

La IA es una disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o el razonamiento lógico.

Aprendizaje automático

Está dentro de la inteligencia artificial. Es el estudio y construcción de algoritmos que pueden aprender y hacer predicciones sobre datos.

Datos → algoritmo de entrenamiento → modelo → predicción.

Existen varios tipos de aprendizaje automático:

- Aprendizaje Supervisado: hay etiquetas que supervisan el aprendizaje (perro/gato)
- Aprendizaje No Supervisado: no hay etiquetas.
- Aprendizaje Semi-Supervisado: usan datos etiquetados y datos no etiquetados (weakly learning, etiquetas malas)
- Aprendizaje por Refuerzo: aprendizaje a partir de interacción con el ambiente (jugar al ajedrez). Aprende por prueba y error a partir de un sistema de recompensas.

Deep Learning

Es una técnica para implementar aprendizaje automático. Los modelos basados en deep learning son capaces de aprender representaciones de los datos de entrenamiento en múltiples niveles de abstracción (capas), componiendo módulos simples que sucesivamente transforman dichas representaciones en otras con mayor nivel de abstracción.

Redes neuronales artificiales

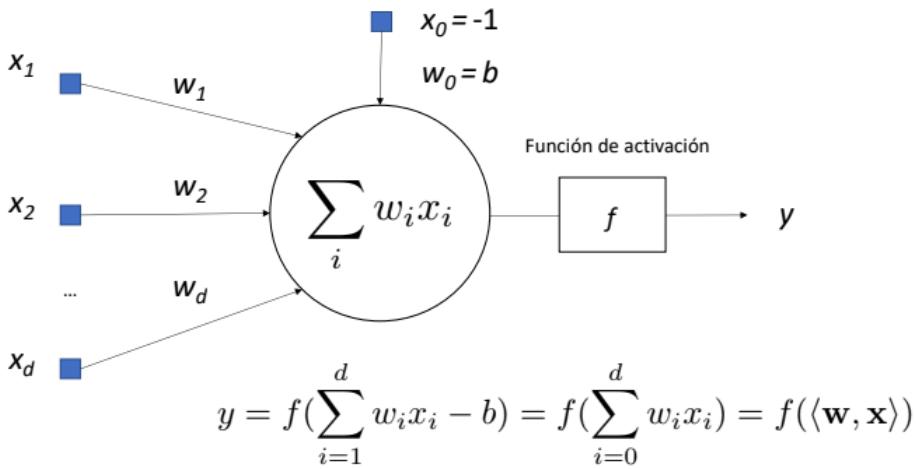
Una red neuronal $y = f(x, w)$ tiene:

- x = inputs (atributos)
- w = n parámetros a aprender (como con descenso por gradiente), es decir, los pesos
- **capas**: son las diferentes capas de neuronas, que pueden ser ocultas (en el medio)
- **función de pérdida**: mide cuán bien la red está funcionando, cuantificando la diferencia entre las predicciones de la red y los valores reales de los datos de entrenamiento. Se evalúa durante el proceso de entrenamiento después de que la red ha hecho una predicción, comparando esta predicción con la etiqueta o valor real, buscando los pesos que la minimizan, es decir, mejorar el rendimiento del modelo.
- **función de activación**: se utilizan para introducir no linealidades en la red neuronal, permitiendo que la red aprenda y represente *relaciones no lineales* en los datos. Se aplican a las salidas de las neuronas en las capas ocultas y en la capa de salida de la red neuronal.
- y = outputs (la salida, es decir, si es gato o perro)

Perceptrón Simple

La idea del perceptrón simple fue sacada de la neurona biológica. En este tenemos una sola capa. Además, le agregamos el bias (que es el $x_0 = -1$), que es un parámetro que le agregamos al modelo para moverlo del origen.

La f es la función de activación, que si por ejemplo se cumple lo clasifica como True, y sino como False.



Existen muchos tipos de funciones de activación:

- Función signo

$$\text{sgn}(z) = \begin{cases} -1 & \text{si } z < 0 \\ +1 & \text{si } z \geq 0 \end{cases}$$

- Función signo lineal

$$\text{sln}(z) = \begin{cases} -1 & \text{si } z < -a \\ \alpha z & \text{si } -a < z < a \\ +1 & \text{si } z \geq a \end{cases}$$

- Función sigmoidea

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

Entrenamiento

Necesitamos un dataset de entrenamiento y una función de pérdida.

Función de pérdida (o costo): $L(x, d; w)$

La función de costo devuelve un valor alto si nuestra predicción es incorrecta, y bajo si la predicción es correcta. Sirve como guía en el proceso de búsqueda de los parámetros (pesos) de nuestro modelo a entrenar

Algoritmo de optimización	Función de pérdida
$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N \mathcal{L}((\mathbf{x}, d)_n; \mathbf{w})$	
Parámetros del modelo entrenado	

Estrategias para buscar los parámetros óptimos

1. Búsqueda aleatoria: Empiezo con pesos w random y se va moviendo de forma aleatoria pero quedándose con el menor. No es una buena opción.
2. Método del gradiente descendente

Método del gradiente descendente

Acá actualizamos los pesos en la dirección en que garantiza una *reducción de la función de pérdida*. El vector *gradiente apunta en la dirección de mayor crecimiento de la función*. Al hacer las derivadas (cada posición i del vector) indica cómo cambiará el valor de la función en un entorno infinitesimalmente pequeño alrededor de un punto al variar el peso w_i . Garantiza la convergencia a un mínimo local de la función a optimizar (en el caso de error cuadrático como función de pérdida del perceptrón simple lineal es una función convexa, es decir que garantiza el mínimo absoluto)

Algoritmo:

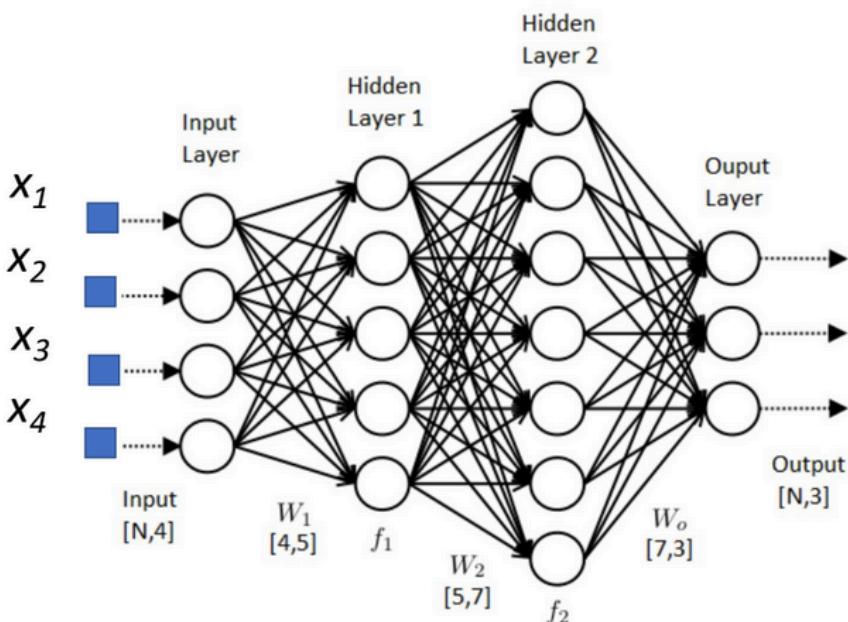
1. Inicializar los pesos W aleatoriamente
2. Dado el dataset $D = \{(x, d)_n\} = \{(x, d)_1, \dots, (x, d)_N\}$. Actualizar en la dirección opuesta al gradiente de la **pérdida promedio** (de todo el dataset): $w = w - \delta \nabla_w L(D; w)$
3. Iterar hasta satisfacer algún criterio de convergencia

Función de pérdida promedio: $L(D, w) = \frac{1}{N} \sum_{n=1}^N L((x, d)_n; w)$ (es la sumatoria sobre todos los datos de entrenamiento)

Clase 2: Perceptrón Multicapa y Backpropagation

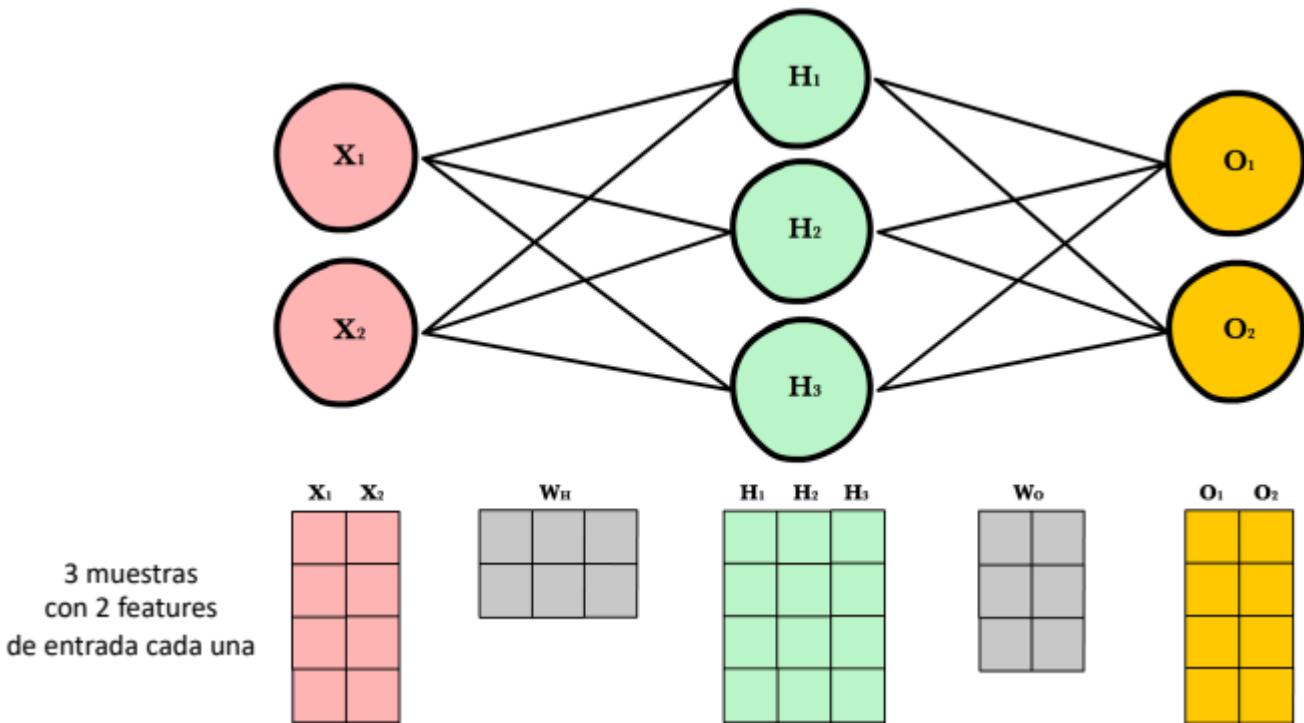
Perceptrón Multicapa (MLP)

Tengo más de una salida. La última capa del perceptrón se asocia a la cantidad de clases en las que quiero clasificar. Si las clases son excluyentes, se le dice multiclas. Si no lo son, se le dice multietiqueta.



La matriz de pesos de cada capa tiene dimensión: cantidad de neuronas de entrada * cantidad de neuronas de salida.

En este ejemplo sería la primera matriz de pesos W_1 de dimensión $4*5 = 20$. $W_2 = 5*7$. $W_3 = 7*3$



Método del gradiente para MLP

En el paso 2 tenemos $W = W - \delta \nabla_w L(D; W)$ con W una matriz con un vector de pesos w_i por capa

Variantes del método del gradiente

- Gradiente descendiente clásico
- Gradiente descendiente estocástico
- Gradiente descendiente por mini-batches. Cambia el algoritmo:
 2. Dado el dataset $D = \{(x, d)_n\} = \{(x, d)_1, \dots, (x, d)_N\}$
 3. Por cada minibatch $D_M = \{(x, d)_m\} = \{(x, d)_1, \dots, (x, d)_M\}, M < N \rightarrow$ tomamos un pedazo de D
$$W = W - \delta \nabla_w L(D_M; W)$$

Función de activación de salida: Softmax

La idea es normalizar los pesos que salen para poder obtener una “interpretación probabilística” de la salida (que todos sean no negativos y sumen 1). En general se usa en la última capa.

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{c=1}^C e^{z_c}} \begin{pmatrix} 1.2 \\ 0.9 \\ 0.4 \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{e^{1.2}}{e^{1.2} + e^{0.9} + e^{0.4}} \\ \frac{e^{0.9}}{e^{1.2} + e^{0.9} + e^{0.4}} \\ \frac{e^{0.4}}{e^{1.2} + e^{0.9} + e^{0.4}} \end{pmatrix} = \begin{pmatrix} 0.46 \\ 0.34 \\ 0.20 \end{pmatrix}$$

$P(\text{Healthy})$
 $P(\text{MCI})$
 $P(\text{AD})$

Función de pérdida: Entropía Cruzada

Entropía Cruzada entre dos distribuciones de probabilidad p (probabilidad reales) y q (probabilidad estimada de pertenecer a cada clase). Me da noción de distancia entre 2 distribuciones, p y q , que tan diferente es la predicha de la real. Luego, con distintas $q(x)$ puedo fijarme cual se parece más a la real, es decir, la que tiene menos entropía. Si uso GT como predicha, la cross entropy me debería dar 0, pues no hay distancia entre real y real.

Distribución Ground Truth

$$CE(p, q) = - \sum_x p(x) \log q(x)$$

↓
x
↑
Distribución Estimada

En etiquetas categóricas, usamos vectores canónicos que me dicen cuál es la categoría, esta es la **notación One Hot**. ¿Cómo comparo estos vectores a la predicción de mi modelo? Con entropía cruzada. En la sumatoria solo sobrevive el término que tiene la clase correcta.

Entropía Cruzada vs Accuracy

Veamos un ejemplo:

Pred	$\begin{pmatrix} 0,1 \\ 0,2 \\ 0,7 \end{pmatrix}$	$\begin{pmatrix} 0,3 \\ 0,4 \\ 0,3 \end{pmatrix}$	$\begin{pmatrix} 0,3 \\ 0,3 \\ 0,4 \end{pmatrix}$
GT	$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

Tasa de acierto = 2/3

Entropía cruzada = 4.14

Pred	$\begin{pmatrix} 0,3 \\ 0,4 \\ 0,3 \end{pmatrix}$	$\begin{pmatrix} 0,1 \\ 0,7 \\ 0,2 \end{pmatrix}$	$\begin{pmatrix} 0,1 \\ 0,2 \\ 0,7 \end{pmatrix}$
GT	$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

Tasa de acierto = 2/3

Entropía cruzada = 1.92

Con el accuracy pierdo mucha información, me puede dar $\frac{2}{3}$ de aciertos en ambos casos (no diferencia en cuánto no le pegué) cuando la entropía me da 4.14 vs. 1.92, que en el segundo caso me da mejor ya que la pifió por poco, en el primer caso la correcta le dió con 0.1 y el segundo caso con 0.3.

Con la entropía cruzada solo veo la clase correcta (la que sobrevive en la sumatoria), si el real era banana, no me importa las probabilidades que le quedó a ser naranja o manzana.

$CE(p, q) =$

$$-(1*\ln(0.1)+0*\ln(0.2)+0*\ln(0.7)) - (0*\ln(0.3)+1*\ln(0.4)+0*\ln(0.3)) - (0*\ln(0.3)+0*\ln(0.3)+1*\ln(0.4)) = 4.14$$

Teorema de aproximación universal

Teorema: Establece que una red feed-forward con una sola capa intermedia es suficiente para aproximar, con una precisión arbitraria, cualquier función con un número finito de discontinuidades, siempre y cuando las funciones de activación de las neuronas ocultas sean no lineales.

Es un teorema de existencia (dice que la solución existe pero no dice que sea fácil encontrarla)

Función de pérdida no-convexa en MLP

Gradient descent no garantiza encontrar el mínimo global en funciones no-convexas, solo garantiza mínimos locales. Sin embargo, para redes neuronales grandes, la mayoría de los mínimos locales son similares y presentan performance similar en los dataset de test. Además la probabilidad de encontrar mínimos locales 'malos' decrece con el tamaño de la red.

Focalizarse demasiado en la búsqueda del mínimo global en el dataset de entrenamiento no es útil en la práctica, ya que puede terminar en sobreajuste del modelo

¿Cómo calcular el gradiente?

1. Derivación analítica: derivar a mano y escribir el código
2. Derivación numérica : diferencias finitas
3. Derivación simbólica: se realiza utilizando las reglas estudiadas en Análisis matemático pero automatizadas (ej: Maple, Mathematica)
4. **Derivación automática (backpropagation):**
 - Se definen las derivadas para las operaciones 'primitivas' (matemáticas y de control)
 - Se construye un grafo de operaciones y se deriva siguiendo la regla de la cadena.

Backpropagation

En redes la derivada es como cambia la función de pérdida cuando me corro un poquito respecto a los pesos. La derivada nos indica entonces "la sensibilidad" de la función al cambio en la variable.

En backpropagation tenemos dos pasadas:

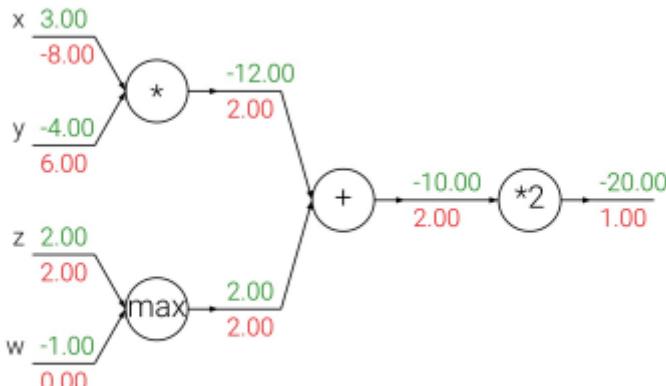
- **la forward:** instancio y corro el algoritmo. Vamos de atrás para adelante haciendo las cuentas y poniendo los resultados donde corresponde.
- **la backward:** vamos de adelante para atrás, haciendo la derivada y reemplazando por el valor que tengo. Además, multiplico en cada paso por el valor anterior.

Iniciamos con valores de pesos que ya tenemos.

En la pasada backward ¿Qué pasa cuando llegamos a una bifurcación?

Si llamamos x e y a cada bifurcación y tomamos la derivada:

- Sumas: copio el gradiente que vengo retropropagando (copio el mismo número)
- Producto: multiplico el gradiente que vengo retropropagando por la otra entrada (copio el mismo número * número que entra de la otra)
- Max = envía el gradiente sólo hacia la mayor entrada.



$$f(x, y, z, w) = ((xy) + \max(z, w)) * 2$$

La verde es la pasada forward (arranco según los valores de mis datos train) y la roja es la backward.

Veamos la pasada backward:

Arranco con 1 siempre en la pasada backward, luego, siguiendo con el ejemplito, tenemos en el nodo $*2$, dónde los podemos interpretar como $2x$, y su derivada es 2 (multiplicando por lo anterior, por regla de la cadena), por lo que el gradiente en este paso es 2.

Luego tenemos el nodo de la suma (bifurcación) y aplicamos la regla de las sumas, donde copio el gradiente que vengo retropropagando: $X+Y$, para un lado la derivada respecto de X y la otra respecto de Y , en ambos casos la derivada es 1 y se multiplica por el gradiente que veníamos teniendo que es 2.

Siguiendo con la bifurcación de arriba, tenemos X^*Y , donde la derivada de arriba queda Y , es decir -4, por 2 (el gradiente anterior) y nos queda -8. En el gradiente de abajo queda X , $3^*2 = 6$. En la bifurcación de abajo, tenemos $\max\{Z, W\}$, la derivada de arriba queda → nos quedamos con la variable mayor de los dos, y copiamos el gradiente en esa variable mayor y en la otra nos queda un 0. Todo esto es para conseguir los gradientes, en qué dirección me muevo en el siguiente paso (al lado contrario), sólo nos interesa w que es al que estamos buscando cuál minimiza.

Clase 3: Redes Neuronales Convolucionales

Procesar una imagen con un perceptrón

Hasta ahora definimos lo que es un perceptrón (con una simple capa no hacemos mucho) y un perceptrón multicapa (pueden aproximar cualquier cosa). Cuando aprendíamos, resolvíamos un problema de optimización.

¿Cómo procesamos una imagen con un perceptrón multicapa? Vectorizo la matriz (imagen) y ese vector que tiene tantos espacios como píxeles se lo doy al perceptrón (me quedan tantos inputs/features de entrada cómo pixeles hay en el vector).

Desventajas

- Se pierde la estructura original de los datos (pasamos de matriz a vector)
- Para grandes imágenes, la cantidad de neuronas (y sus consecuentes conexiones) crece exponencialmente (pues más pixeles más neuronas de entrada)
- No tenemos una noción clara de análisis multi-escala/multi-resolución (algo que resulta útil en general en análisis de imágenes)

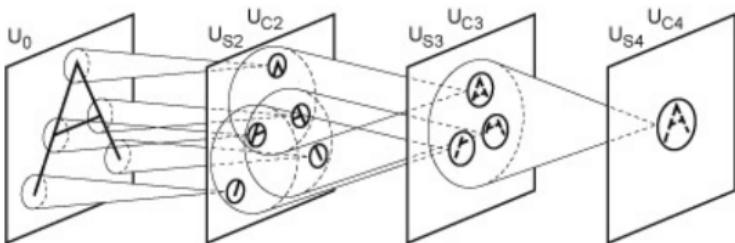
Entonces esta no es la mejor opción para clasificar imágenes.

Surgen unas redes neuronales inspiradas en el sistema visual.

Redes neuronales inspiradas en el sistema visual

En un estudio se encontró que las neuronas de la corteza visual temprana se organizan de forma **jerárquica**, donde las primeras reaccionan a patrones simples como líneas, y las posteriores capas responden a patrones más complejos combinando las activaciones que reciben. En el modelo propuesto, las neuronas en las capas de adelante tienen un mayor campo receptivo y son menos sensibles a la posición desde la cual proviene dicho estímulo.

Veamos un ejemplo:



U_0 es la imagen que estamos viendo.

Al principio (u_{c2} y u_{s2}) miro una parte de la imagen y a medida que nos metemos más profundo en la red neuronal, ampliamos el campo receptivo (es decir, lo que vemos).

La idea del deep learning tiene que ver con esto, hacer capas sucesivas de diferentes niveles de abstracción.

Convolución

1D: Es la integral del producto de una función x con una versión trasladada y reflejada de otra w

$$s(t) = \int x(a)w(t-a)da$$

Señal convolucionada Señal de entrada Filtro
 → señal continua

Señal discreta:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

2D:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

↓
La convolución es comutativa

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n)$$

Propiedad útil ya que el rango de valores válidos de m,n de una imagen es mayor en general que el del kernel.

Correlación cruzada

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n)$$

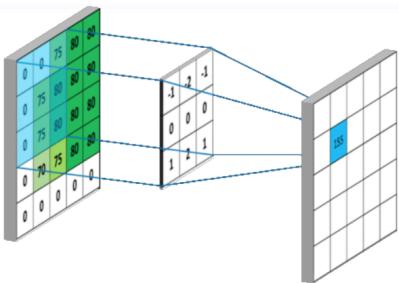
En el contexto de redes neuronales convolucionales se suele utilizar la **correlación cruzada** en lugar de la convolución para evitar el flipping del kernel.

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

Correlación cruzada

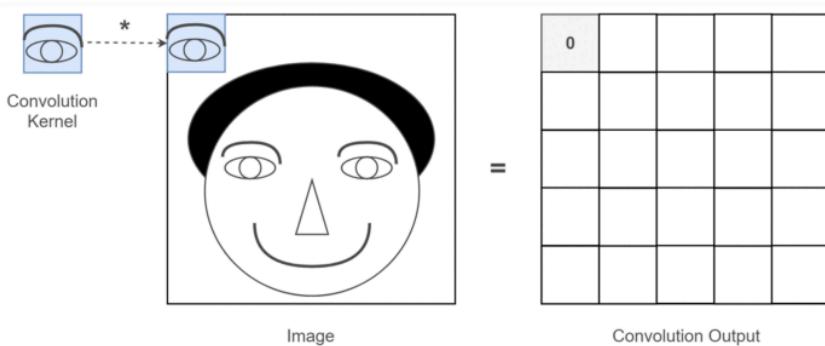
En el contexto de redes neuronales convolucionales se suele utilizar la correlación cruzada en lugar de la convolución para evitar el flipping del kernel (en la convolución el kernel se da vuelta antes de desplazarse por la imagen).

Depende qué cosas le pongo a mi kernel (números de mi matriz), obtengo distintas representaciones de la imagen cuando lo convoluciono con esta. La idea es aprender qué kernel usar con estas redes, es decir, en las primeras capas de convolución aprendemos los kernels en vez de los pesos que aprendíamos antes.



→ el del medio es el kernel, es la matriz que uso para convolucionar la imagen. Es una multiplicación de matrices. El kernel lo voy desplazando para multiplicar por cada porción de la imagen/matriz.

Si quiero detectar una figura en una imagen, uso un kernel con esta (pongo los píxeles de la figura que quiero detectar) y la convolucione con la imagen. Donde me da valores más altos está esa figura.



Ejemplos de filtros:



$$* \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} =$$



→ blurrea la imagen porque cada pixel

es el promedio de los anteriores 9



$$* \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} =$$



→ solo me quedan las rayas verticales



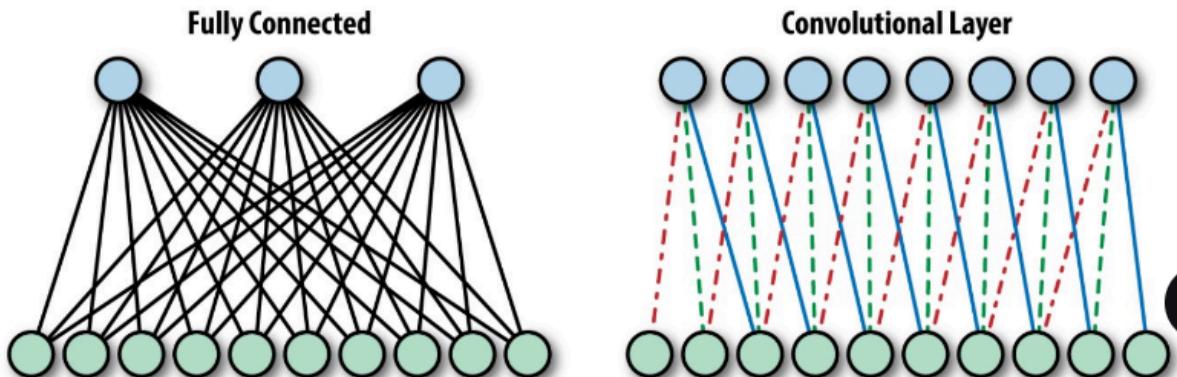
$$* \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} =$$



→ solo me quedan las rayas horizontales

Capas convolucionales

Convolución 2D

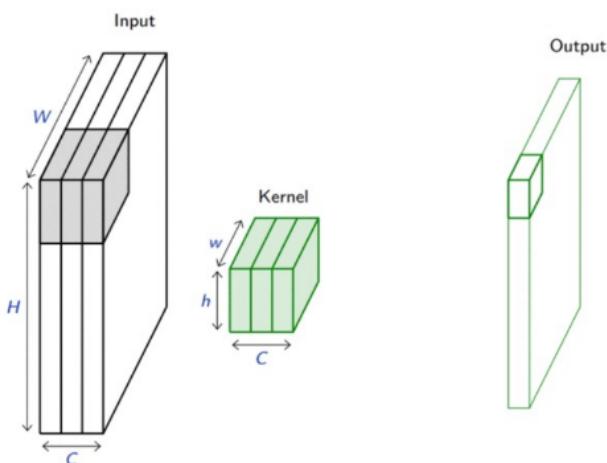


En una capa convolucional las neuronas convolucionales tienen soporte local, es decir, tienen alcance limitado a una pequeña región de la entrada en cada paso de la convolución. Además, estas neuronas tienen pesos compartidos, es decir, para calcular el valor de cada una de las nuevas neuronas se utiliza el mismo kernel, todos tienen igual peso (a diferencia de antes donde la fully connected las neuronas conectadas a todos con diferentes pesos w_i). Por lo tanto, la conectividad es local en el espacio y total en profundidad.

Las funciones de activación no lineales (ReLU, etc) se aplican elemento a elemento en las neuronas de salida. Elemento se refiere a la nueva neurona que se obtuvo de la convolución con el kernel anterior.

Notar que las neuronas convolucionales se organizan como un feature map (tienen ‘estructura’), es decir, la nueva matriz que se obtuvo de convolucionar la capa anterior.

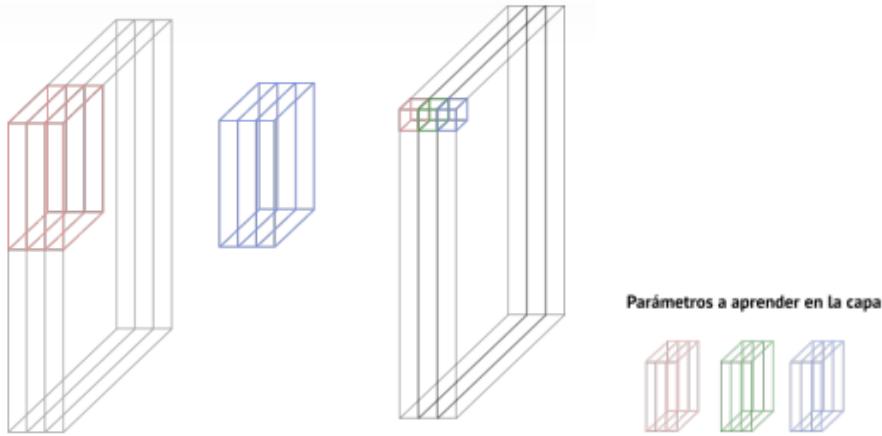
Antes teníamos la imagen como matriz y todas las neuronas de entrada se conectaban con todas las siguientes con distintos pesos. Ahora, tenemos un kernel y vamos multiplicando element-wise en cada partecita de la imagen (que son un conjunto de neuronas/pixeles) y devuelve otra capa de neuronas (el feature map, son las salidas intermedias).



3D: Input = $H \times W \times C$ donde H: altura, W: ancho, C: colores

3D: Kernel = $h \times w \times C$ donde h: altura, w: ancho, C: colores

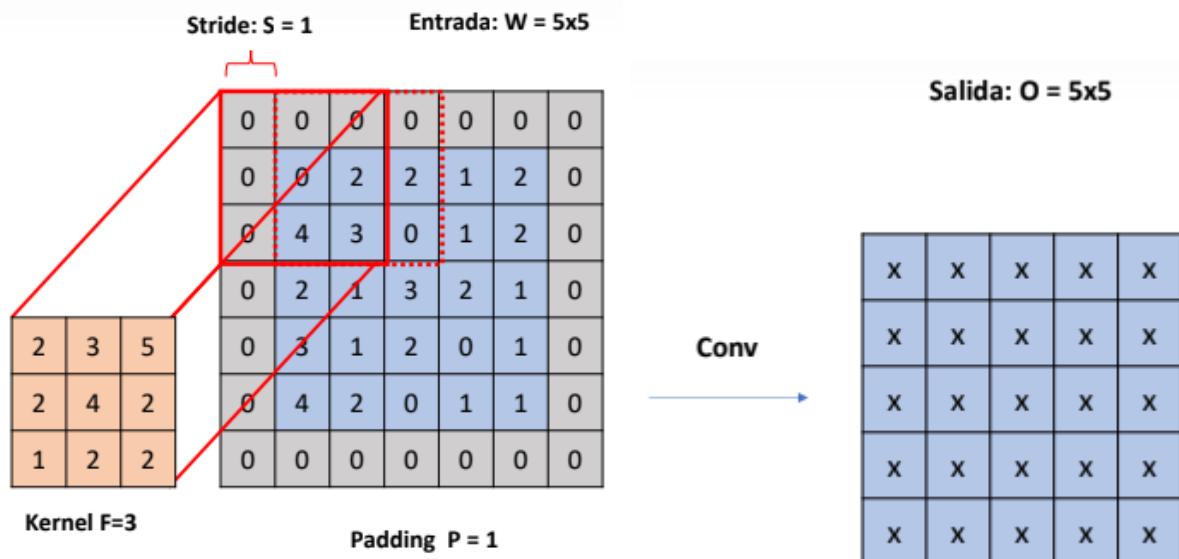
2D: Output = ...



La idea es: primero con un kernel (rosa) veo una partecita, hago el producto interno y obtengo la primera neurona de la primera rodaja de pan, con este mismo kernel voy recorriendo en todas las partecitas y consigo toda la rodaja de pan (feature map). Luego, tomo otro kernel (verde) y se vuelve a recorrer las partecitas de la misma forma, consiguiendo la segunda rodaja de pan (segundo feature map), y por último conseguimos el tercero (violeta), obteniendo así la capa de neuronas.

Los feature maps que va devolviendo cada kernel se van concatenando. Usualmente, se usa el mismo tamaño de kernel, entonces se “mira” la misma partecita de la entrada cuando haces la convolución de cada kernel. No necesariamente el tamaño del input es el mismo que el del output, ya que este puede ser más chico.

Definamos algunos conceptos...



Stride: es el tamaño del paso que doy cada vez que muevo el kernel

Padding: es el tamaño de la basura que agrego para ajustar el tamaño de la matriz (los ceros).

F: tamaño del kernel

W: tamaño de la entrada

O: tamaño de la salida, que se obtiene como:

$$O = \frac{(W - F + 2P)}{S} + 1$$

El output suele ser más chico que el input porque tenemos el dividido S (en la imagen S=1 por lo que no se achicó).

El padding y el stride los puedo variar en el ancho y alto, por lo tanto debo calcular la O para cada uno.

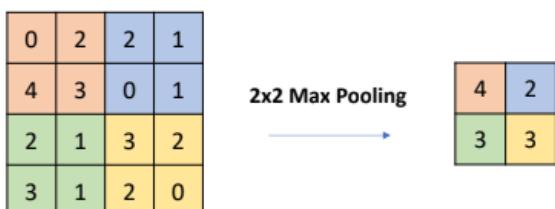
Viendo la cuenta, una convolución con P=1, S=1 y F=3 mantiene las dimensiones.

Capa de Pooling

La capa de pooling en redes neuronales convolucionales realiza una operación de reducción de dimensionalidad. Simplifica la representación de la imagen, manteniendo las características más importantes y reduciendo la carga computacional, lo que facilita la construcción de modelos de aprendizaje profundo eficientes y robustos.

Capa de Max-Pooling

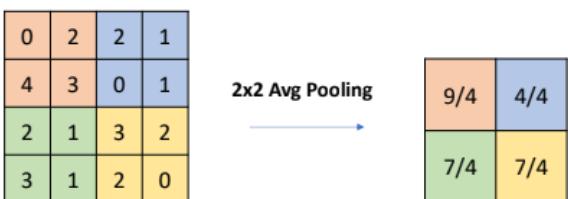
En cada “partecita” toma el máximo.



Reduce la dimensión de los feature maps. Contribuye a la invarianza respecto a pequeñas traslaciones en las imágenes de entrada

Capa de Avg-Pooling

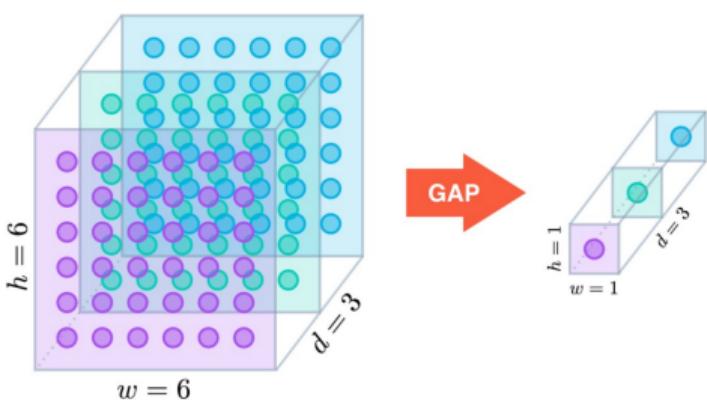
En cada “partecita” toma el promedio.



Global Average Pooling

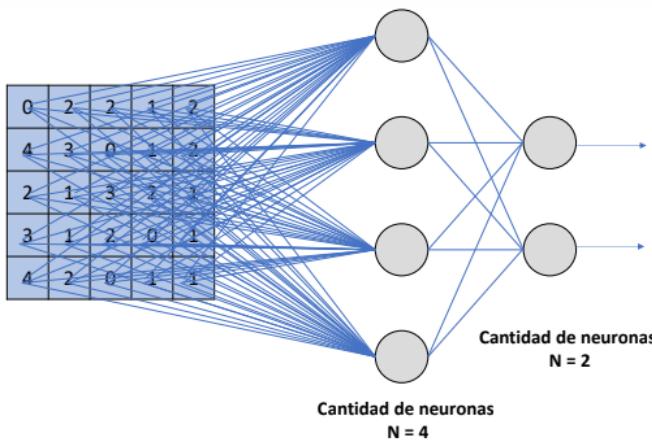
En el global average pooling (GAP), se calcula el promedio de cada feature map individualmente. Este proceso reduce cada feature map a un solo valor, lo que resulta en un vector de características final donde cada elemento representa el promedio de un mapa de características completo.

Se usa al final de toda la red, ya que es muy extremo porque rompe todo.



Capa Fully Connected (Totalmente Conectada)

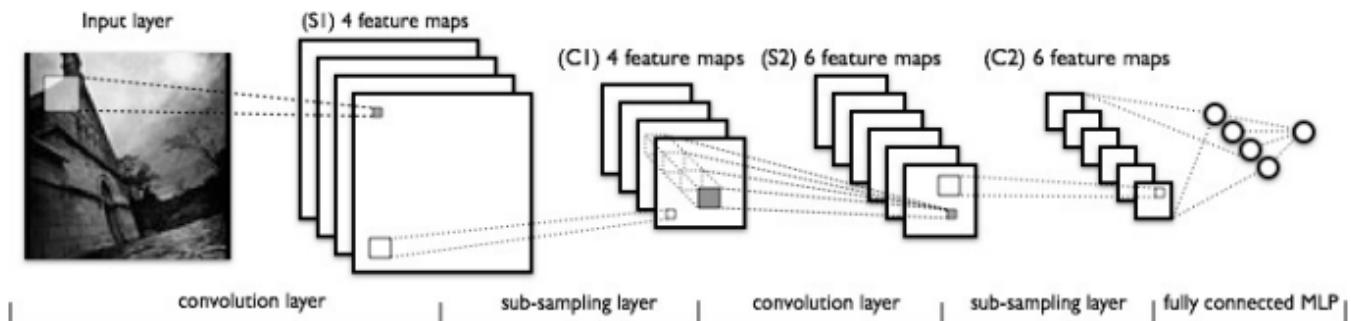
En las redes neuronales convolucionales, la última capa de todas es una capa totalmente conectada como las que teníamos antes.



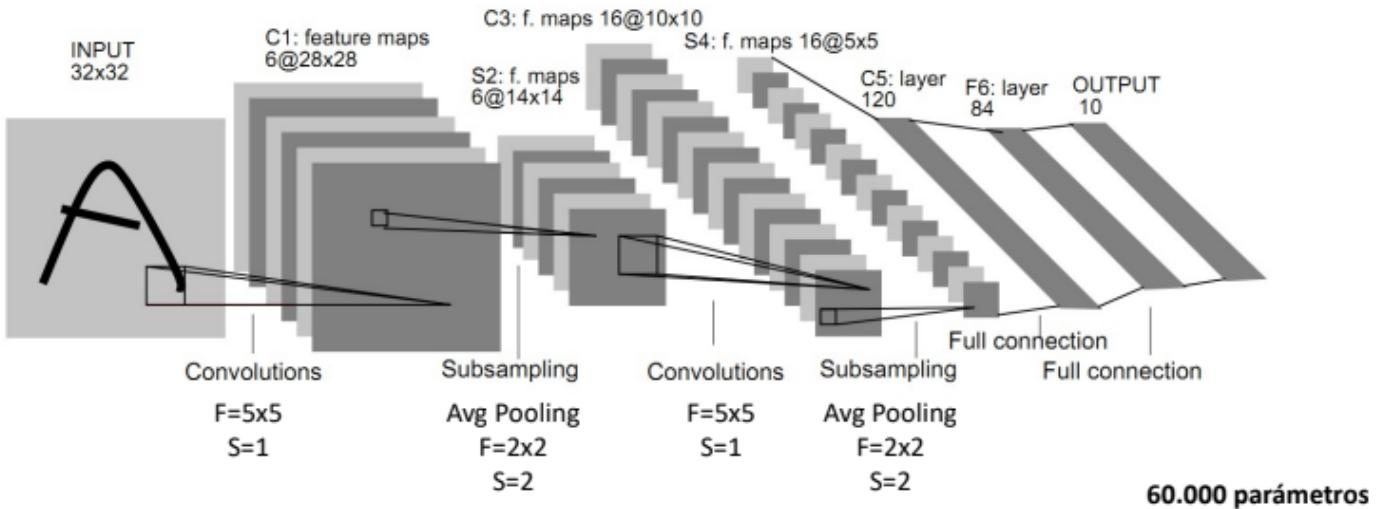
→ en este ejemplo las dos últimas capas son fully connected

connected

Estructura total (CNN)



En este ejemplo tenemos el input (que es la imagen más todos los tamaños de los kernels que vaya a usar), luego una capa de convolución, luego una capa de pooling (sub-sampling), otra convolución, otro pooling. Cuando llego al último feature map antes de la fully connected, tengo que aplanar estos feature maps, es decir, pasarlo a un vector que le da al MLP (la capa fully connected). Para pasar a vector, debería usar un comando para saber el tamaño de este (teniendo en cuenta los tamaños y cantidad de feature maps que me quedaron en la anterior capa). Luego, tenemos el output de este MLP.



Entrenamiento de CNN

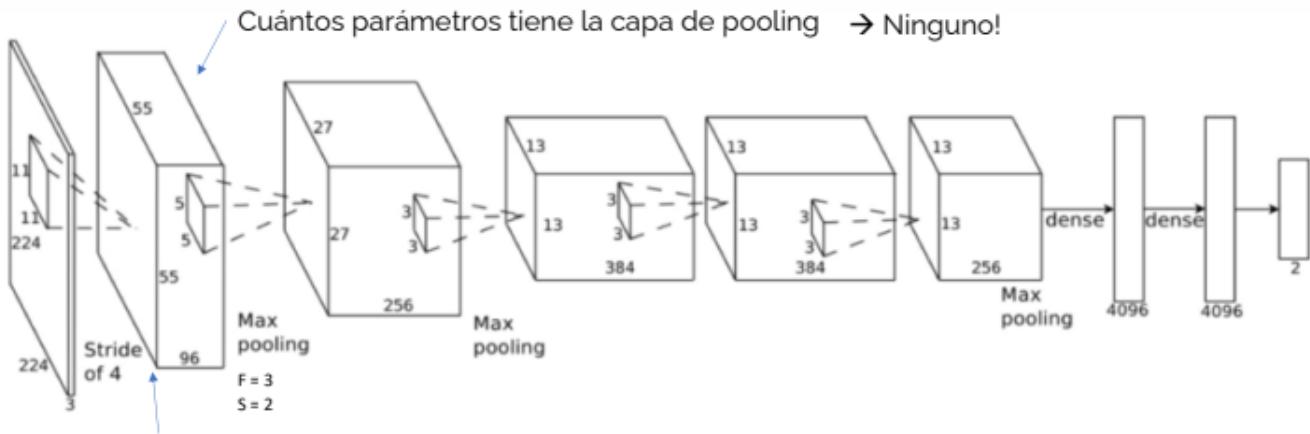
Debemos aprender los kernels de cada convolución como así también los pesos de la última capa fully connected (en la inicialización pasamos valores randoms a los kernels y pesos). Es igual que como hicimos antes pero ahora tenemos feature maps, que son más pesados que los w_i (pesos) que teníamos que entrenar antes.

Ventajas de CNN para el análisis de imágenes

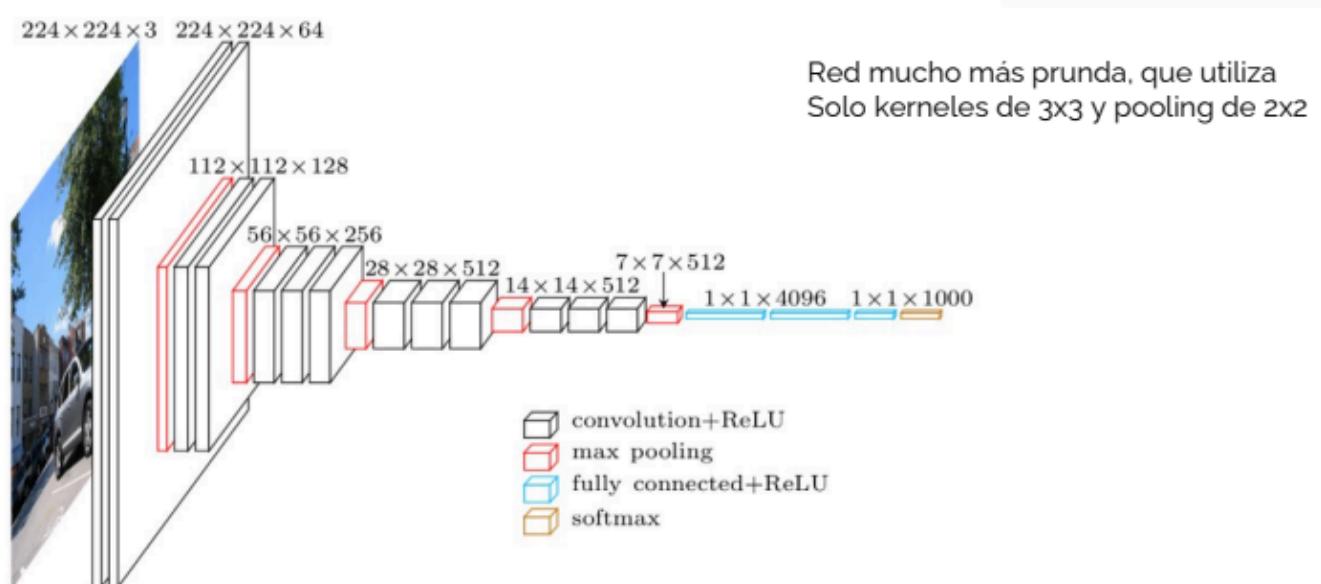
- Naturalmente adaptadas a la estructura regular de las imágenes (por medio de la operación de convolución), es decir, más o menos se mantiene la ubicación de las imágenes.
- Invariantes respecto a traslaciones: lo que significa que pueden encontrar patrones independientemente de donde se encuentran en la imagen de entrada (por ejemplo, el ojo).
- Aprendizaje end-to-end: aprende desde el input hasta el output
- Bajos requerimientos de memoria: weight sharing (comparten los pesos de los kernels) y además el pooling no tiene pesos y achica la imagen.
- Eficientes en test-time: en training quizás tarda mucho pero una vez que tenemos el modelo, predice rápido.
- Buen grado de generalización si se entrena con suficientes datos (generaliza bien a nuevas imágenes no vistas)

Algunas arquitecturas importantes

AlexNet (2012)

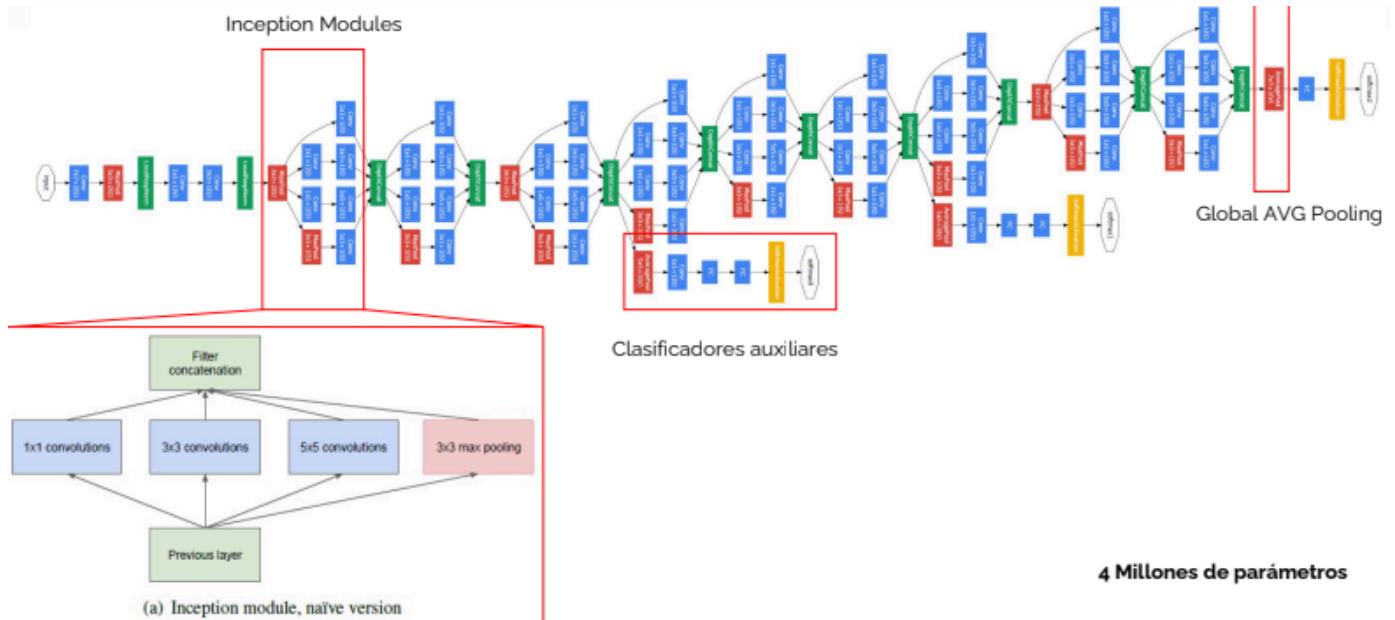


VGGNet (2014)



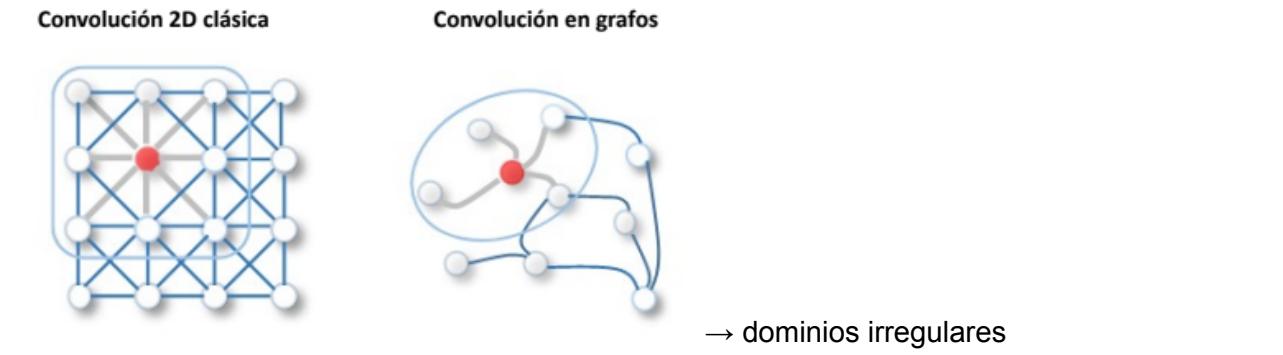
138 millones de parámetros

GoogLeNet (2014, InceptionV1)



Redes Neuronales Convolucionales en grafos

Hace la convolución con grafos, sirve en casos donde no tenemos una matriz estándar de una imagen.



Filtrado espacial : La convolución se define directamente en el dominio original del grafo

- ✓ Interpretación intuitiva y analogía simple con convoluciones en dominios estructurados
- Requiere definir un sistema de vecindad y un orden para los nodos (medio manual)

Filtrado espectral: Convoluciones en el espacio equivale a multiplicar en el dominio de Fourier

- ✓ No se requiere la noción de orden entre vecinos
- ✓ Definición precisa del concepto de convolución por medio del análisis espectral de grafos
- ✓ Se pueden modelar kernels estrictamente localizados en el grafo
- Los kernels que se aprenden en una estructura de grafo dada no pueden ser transferidos a otras

Clase 4: Entrenando una red neuronal

Funciones de activación de una red neuronal

Sigmoidea

Esta función de activación la utilizamos solamente en la última capa ya que el gradiente tiende a 0 cuando X tiende a ∞ y $-\infty$, por lo que en los entrenamientos (en backward) iríamos multiplicando por

números muy chicos. Por lo que si lo usaramos en todas las capas, las del fondo (más cercanas al output) entran pero las más cercanas no porque el gradiente se va desvaneciendo.

$$\text{ReLU}(x) = \max(0, x)$$

Esta función de activación tiene menor probabilidad de que el gradiente se desvanezca que en el caso de la función de activación sigmoidea.

Problema: el gradiente es 0 para x negativos, y puede desencadenar un proceso de Dying ReLU.

Leaky ReLU

Evita el problema de la Dying ReLU ya que también genera gradientes no nulos para valores de x negativos



Inicialización de pesos

¿Qué pasa si iniciamos todos los pesos con $W=0$? Todas las neuronas terminan aprendiendo lo mismo.

Por lo tanto, hacemos una **inicialización aleatoria** sampleando de $N(0, \sigma^2)$

¿Cómo lograr que las neuronas operen en un rango razonable al comenzar el entrenamiento?

Para una neurona $z = \sigma(w_1x_1 + \dots + w_nx_n)$ queremos que mientras más grande n (más features tengo) más pequeños los w_i porque sino no operas en un rango razonable.

La idea entonces es samplear los pesos w de una distribución normal con varianza $\sigma^2 = 1/n$ así mientras más features de entrada más chicos son los pesos (porque más chica es la panza de la normal). Útil para tanh

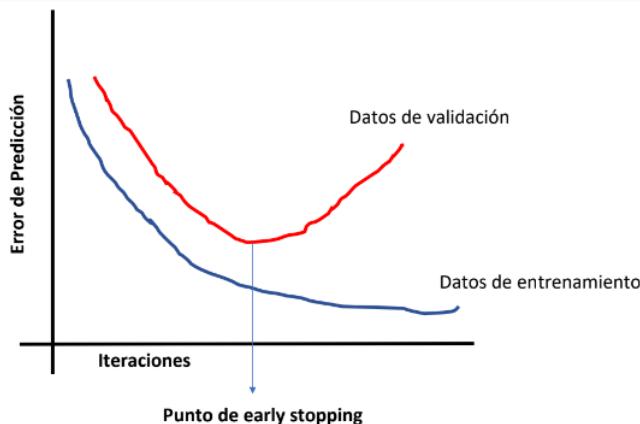
Método He-et-al (2015)

Samplear los pesos w de una distribución normal con varianza: $N(0, 2/n)$.

Útil para funciones ReLU.

Sobreajuste y regularización

Sobreajuste (overfitting)



¿Cómo evitarlo? Para evitar el sobreajuste, hay que regularizar. La regularización es cualquier modificación que hacemos a un algoritmo de aprendizaje con la intención de reducir su error de generalización, pero no su error de entrenamiento. Hay dos tipos:

- Frenar a tiempo antes de que sobreajuste (early stopping)
- Dejar que sobreajuste y en cada iteración me fijo la performance del conjunto de validación y me fijo cuando empeora

L2 Regularization (Weight Decay)

Existen muchos w que pueden minimizar una función de pérdida para un dataset de entrenamiento dado. Puedo reducir la complejidad de mi modelo acotando la cantidad de posibles modelos a construir. Una forma de hacerlo es evitando que existan w_i muy grandes, es decir, agregando una restricción sobre el valor que pueden tomar los pesos w_i .

La regularización L2 es agregar un término de regularización a la función de pérdida que sea la norma euclídea cuadrada de los pesos:

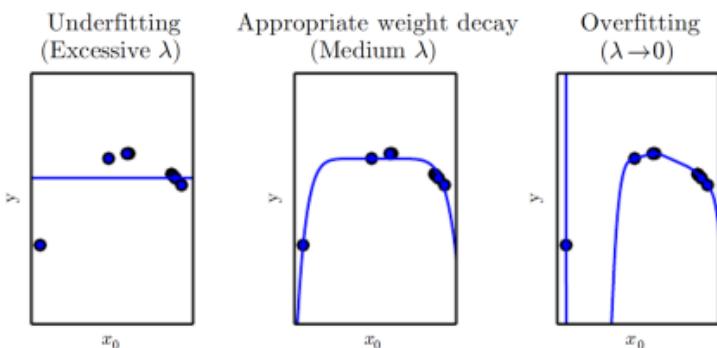
$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N \mathcal{L}((\mathbf{x}, d)_n; \mathbf{w}) + \frac{1}{2} \lambda \|\mathbf{w}\|_2^2$$

→ Agregando esto, si w es muy

grande, la función que quiero minimizar se va muy grande

Con N = Número de data samples en el training set

Influencia del parámetro λ : Ejemplo al fitear un polinomio de orden 9 sobre datos sampleados de una función cuadrática



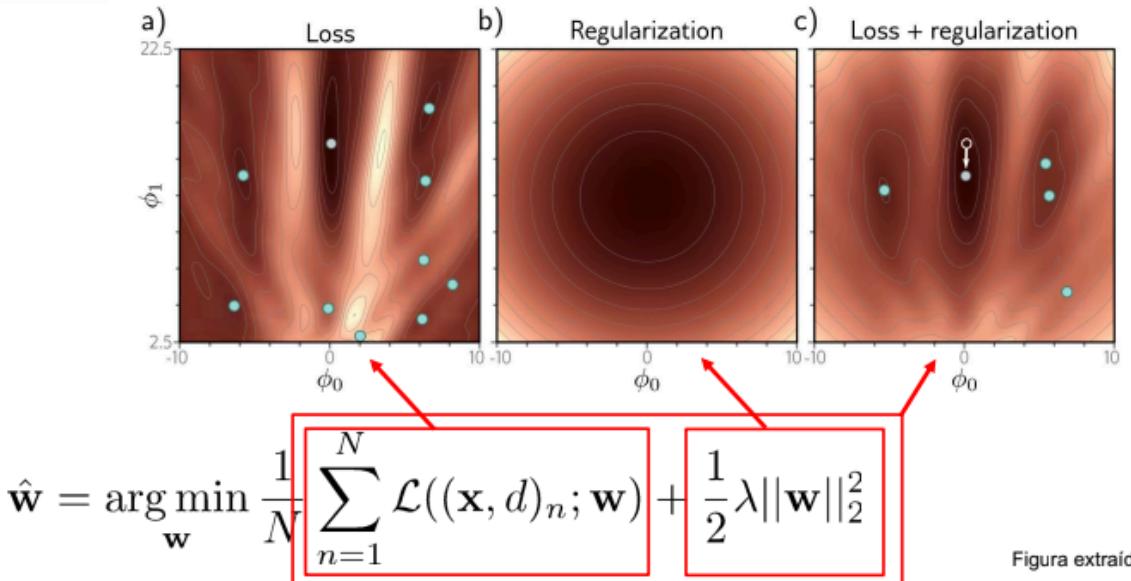


Figura extraída:

- a) Solo se minimiza la función de pérdida. Cuanto más claro, la función de pérdida es mayor. Tenemos muchos mínimos locales
- b) Solo se minimiza la regularización. Minimiza la norma del vector, eso sería el (0,0), el único mínimo global.
- c) Se minimizan ambos términos, tenemos algunos mínimos locales.

Aumentación de datos

Mientras se entrena: se puede aumentar artificialmente el dataset utilizando transformaciones en los datos y conservando las etiquetas.

En tiempo se prueba: es posible generar N versiones de la imagen de test, estimar las predicciones y promediarlas.

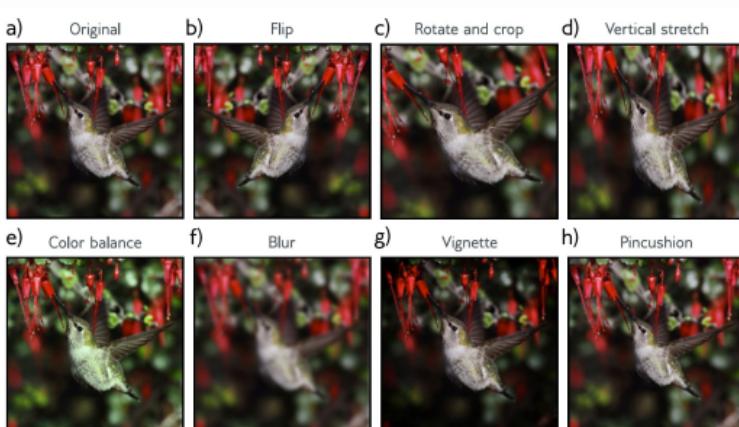
Por ejemplo, si tengo un gato mirando a la derecha, puedo girar la imagen y que mire a la izquierda, así para el modelo son diferentes.

Se puede:

- escalar la imagen (cambiar la escala, aumentarla)
- trasladar la imagen (traslado la imagen al costado)
- rotar la imagen
- flippear la imagen (espejar)
- agregarle ruido (le agrego ruido a la imagen para que sea diferente)

Se puede aplicar online (aplicar la transformación on the fly, mientras hago el código) u offline (hago la transformación en el disco y me hago un dataset nuevo con las nuevas imágenes).

Además, la aumentación de datos no es solo para imágenes, puede ser por ejemplo para una serie temporal.



Regularización: Dropout

Combinar las predicciones de múltiples modelos entrenados con el mismo fin es una forma de prevenir overfitting. Lo que queríamos es entrenar muchos modelos de redes neuronales y luego promediar las predicciones, pero es carísimo, entonces **Dropout** propone que una sola red neuronal sean muchos modelos a la vez ¿cómo?

Lo que se hace es, aleatoriamente con un probabilidad ($1-p$), apagar neuronas en la pasada forward, es decir, hago que la salida de una neurona sea 0 (hace que no confíe en todas mis neuronas). Es como entrenar muchos modelos al mismo tiempo (cada vez que apago una neurona es un modelo nuevo, quedamos más confiados del modelo). Luego en test, prendo todas y además multiplico por p a los pesos de salida de cada neurona. Esto es para reescalar el valor de salida de las neuronas para que estén en el rango que deberían estar (porque al apagar algunas perdes la escala).

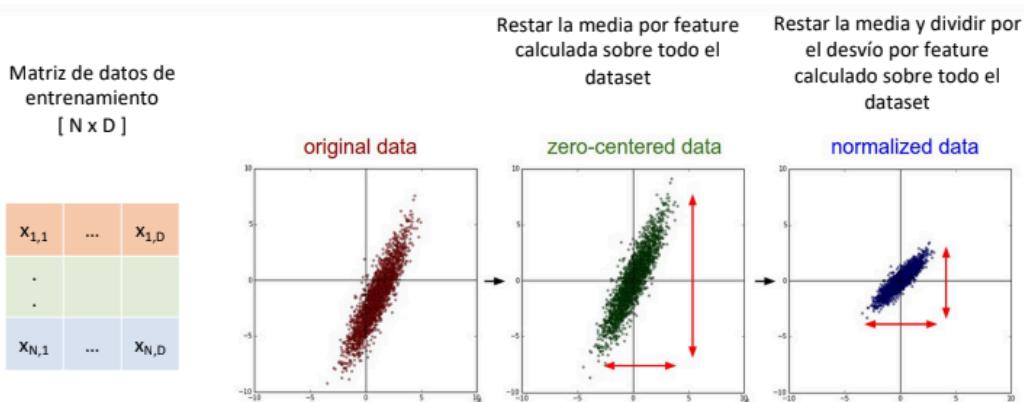
Performea mejor que L2.

¿Cómo usar Dropout para tener una idea sobre la incertidumbre de las predicciones de mi red?

Dejo prendido el dropout en test (no prendemos todas las neuronas, quedan apagadas). Pruebo la misma imagen muchas veces (20 corridas) y veo la varianza entre los 20 modelos. Si varía mucho en 20 corridas, es decir, la varianza es muy grande, es porque no era muy bueno mi modelo. Si es chica, es porque el modelo siempre predice algo “parecido” a pesar de tener algunas neuronas apagadas, es decir, veo qué tan “seguro” está mi modelo.

Normalización de los datos de entrada

Los datos se mueven en distintos rangos, es necesario normalizar..



IMPORTANTE: La media y el desvío son siempre calculados usando sólo los datos de training, pero todos los datos (test y validación también) son normalizados.

Cuando normalizo resto por la media y divido por el desvío. Esto se hace en los datos de entrenamiento, no en los de test. Así se llega al mínimo más fácil y más rápido.

El rango de las “features” (píxeles) es el mismo (0-255), por lo tanto, en general no es necesario dividir por el desvío. Normalizar por pixel no tiene mucho sentido pero se puede. También es usual considerar el valor de la media y desvío calculado sobre todos los pixeles de todas las imágenes de entrenamiento, en lugar de considerarlo por feature (pixel).

En imágenes multi-canal (p. ej. RGB), cada canal suele ser normalizado independientemente (una media y desvío por cada canal).

Acá estamos normalizando la entrada, ¿qué pasa con las capas intermedias?

Batch normalization

Ya que la normalización de los datos de entrada ayuda al entrenamiento, ¿por qué no normalizar las entradas de todas las capas ocultas?

Dados N training samples en un capa y en un minibatch:

1. Calculo media del minibatch para los valores de z, es decir, voy por neurona y computo la media de todos los valores del minibatch.

$$\mu = \frac{1}{N} \sum_{n=1}^N \mathbf{z}_n$$

→ \mathbf{z}_n es el resultado de pasar el dato n por la neurona z. Como lo hago con los N datos del batch y para todas las neuronas, por cada neurona me sale una media porque calculo el promedio para esos N datos del minibatch. Por lo tanto, en cada capa tengo un vector de promedios de tamaño cantidad de neuronas.

2. Calculo varianza del minibatch

$$\sigma^2 = \frac{1}{N} \sum_{n=1}^N (\mathbf{z}_n - \mu)^2$$

3. Normalizo la salida de la neurona antes de entrar a la función de activación

$$\hat{\mathbf{z}}_n = \frac{\mathbf{z}_n - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

4. Reescalo (porque en la función de pérdida no quiero valores entre 0 y 1, quiero que estén desplazados)

$$\mathbf{z}_n^{BN} = \gamma \hat{\mathbf{z}}_n + \beta$$

→ lo que logro es desplazar los valores que salen de la neurona a un rango que me sirva para entrenar

Efectos obtenidos al aplicar Batch Norm

- Rango de valores de salida más estable en las capas ocultas. En consecuencia, simplifica el proceso de aprendizaje de los pesos de dichas capas.
- Efecto regularizador: el cálculo de la media y varianza es por minibatch. Esto introduce ruido al proceso de entrenamiento, produciendo un pequeño efecto de regularización.
- Permite usar funciones de activación con saturación sin caer en el problema del gradiente que se desvanece.

En capas fully connected, se utiliza una media y varianza por cada neurona de la capa. μ , σ^2 , γ , β son **vectores** de tamaño número de neuronas.

En capas convolucionales, dado que los kernels son compartidos por todo el feature map, se utiliza una única media y varianza por cada canal de entrada (feature map) para normalizar. μ , σ^2 , γ , β son **escalares** (distintos por cada canal de entrada)

Batch Normalization en Test Time

Cuando pasamos a datos de testeo (si entrenamos con datos normalizados queremos testearlos así), no tenemos cómo calcular μ , σ^2 pues podemos pensar que no tenemos batches, entonces tenemos que utilizar μ y σ^2 calculados con los datos de train. Para no darle mayor importancia a alguno en específico, y seguir con "momentum" (memoria, historia) utilizamos la media móvil exponencial, con el conjunto de μ y σ^2 calculados para cada batch.

Media móvil exponencial: es una media calculada pesando lo que tenía antes. Para valores de alfa más grande tengo más memoria (momentum), es un hiperparámetro (no lo aprendes).

$$\hat{\mu}_t = \alpha \hat{\mu}_{t-1} + (1 - \alpha) \mu_t$$

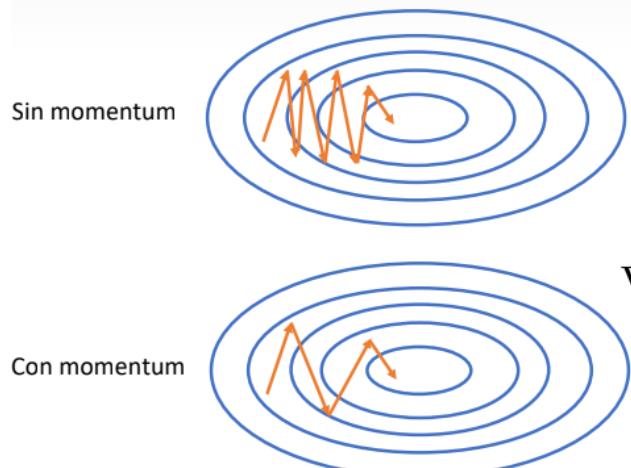
Variantes del gradiente descendiente

- Gradiente descendiente clásico
- Gradiente descendiente estocástico
- Gradiente descendiente por mini-batches
- Gradiente descendiente con Momentum

Gradiente descendiente con Momentum

Una mejora es agregarle momentum (memoria) al gradiente. La idea es llegar más rápido al mínimo. Se hace tomando promedio ponderado de los gradientes actuales y el histórico (anteriores), recuerda cómo se venía moviendo. Empieza lento y después va más rápido (acelera la convergencia del método) porque junta historia. Además, suaviza los pasos dados por el gradiente descendiente en las dimensiones más oscilantes.

$$\mathbf{V}_t = \boxed{\alpha} \mathbf{V}_{t-1} + (1 - \alpha) \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W}) \quad \rightarrow \text{uso } \mathbf{V}_{t-1}, \text{ es decir, lo anterior}$$
$$\mathbf{W} = \mathbf{W} - \delta \mathbf{V}_t \quad \mathbf{V}_0 = \vec{0}$$



→ vemos que en la segunda flechita nos movemos "más" a la derecha porque nos sirvió movernos a la derecha en la anterior (no voy tan en las curvas de nivel, sino que me fijo en lo que hice anteriormente).

Variando el learning rate

Learning rate: parámetro que representa al paso que doy con el gradiente descendiente. Cuanto mayor es, más grande es el paso y viceversa. → Una vez que ya hicimos la pasada backward y conseguimos el gradiente (que nos dice en qué dirección nos movemos) y luego calculamos los nuevos pesos donde el Learning Rate (multiplicado al gradiente) nos dice cuánto nos movemos en esa dirección.

LR Alto: el algoritmo puede oscilar caóticamente (oscila mucho)

LR Bajo: el algoritmo puede dejar de aprender (o tomar demasiado tiempo)

¿Cómo podemos variarlo?

Step Decay: Reducir el learning rate (ej: dividir por 2) cada una cantidad fija de épocas (mirar una vez todo el dataset, y todos los baches), o cuando el error en validación no mejora (se mantiene igual, es porque no captó el mínimo).

$$\delta = \delta_0 e^{-kt}$$

Exponential Decay: decaer el paso con una exp negativa

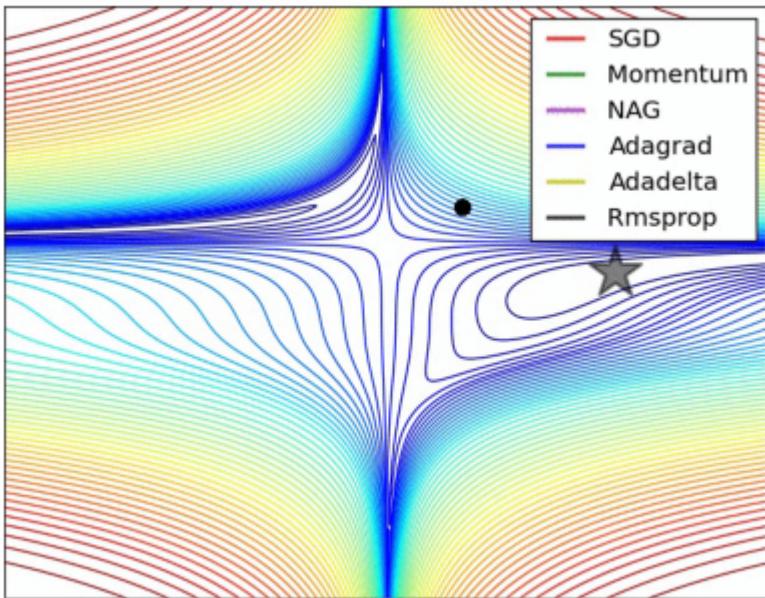
Los métodos anteriores aplican el mismo LR global a todos los parámetros W

$$\delta \nabla_{\mathbf{w}} \mathcal{L} = \delta \left(\frac{\partial \mathcal{L}}{\partial w_1}, \dots, \frac{\partial \mathcal{L}}{\partial w_D} \right)$$

Los **métodos adaptativos** escalan el LR por cada parámetro.

Por ej. siguiendo los métodos:

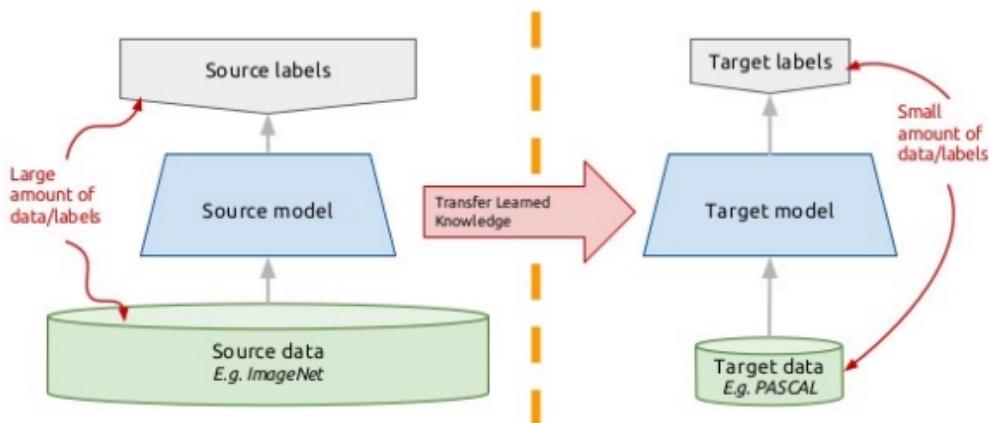
- RMSProp
- Adadelta
- Adagrad
- ADAM (Momentum + RMSProp) (El mejor según Pedro)

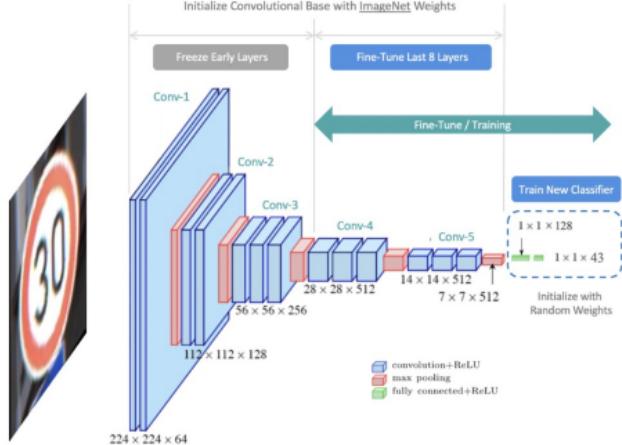


Transferencia de conocimiento (fine tuning)

Vuelvo a entrenar pero con algo diferente, antes entrene con gatos y perros, pero ahora le pongo una vaquita de san antonio. Lo que hago es frisar los pesos que ya entrené. Ya sea frisando una parte o todo, se puede frisar varios lados. Quizás los primeros pesos que entrené con perro o gato, me sirven para entrenar una vaquita de san antonio en el segundo entrenamiento. Esto es porque casi nunca un modelo se comienza a entrenar de cero (se le hace fine tuning a los modelos ya entrenados).

Volver a entrenar el modelo se llama fine tuning. En el primer entrenamiento entrenamos con muchos datos de perros y gatos, y luego usamos el modelo entrenado para entrenarlo una segunda vez con datos de vaquitas de san antonio (pero muchos menos datos).





Clase 5: Redes neuronales más allá de la clasificación de imágenes

Problemas en computer visión

Clasificación

En una imagen hay una única etiqueta, en este caso, gato.



CAT

No spatial extent

Segmentación semántica

Separa *toda* la imagen en distintas etiquetas (a cada píxel le asigna una etiqueta).



GRASS, CAT,
TREE, SKY

No objects, just pixels

Detección de objetos

Detecta múltiples objetos, pero no *toda* la imagen, solamente los objetos que yo quiero.

Object Detection



DOG, DOG, CAT

Segmentación de instancias

Es como una mezcla de Detección de objetos y Segmentación semántica. Lo que hace es segmentar por objeto que me interesa (los pixeles del objeto) y lo puede hacer con múltiples objetos (y no tiene que ser toda la imagen)

Instance Segmentation

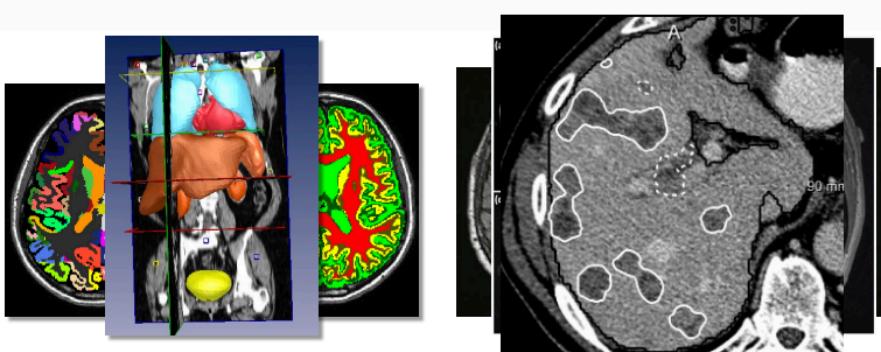


DOG, DOG, CAT

Ejemplos de segmentación de imágenes

Imágenes médicas

segmentación de estructuras anatómicas o mide estructuras del cerebro para ver si hay patologías.



En la primera imagen separamos a los órganos por color y en la segunda nos fijamos por ejemplo si hay un tumor.

Imágenes satelitales

Por ejemplo segmenta la imagen para la búsqueda de piletas:



Imágenes 2D vs Imágenes 3D

Las imágenes pueden ser procesadas en 2D (píxel) o en 3D (voxel).

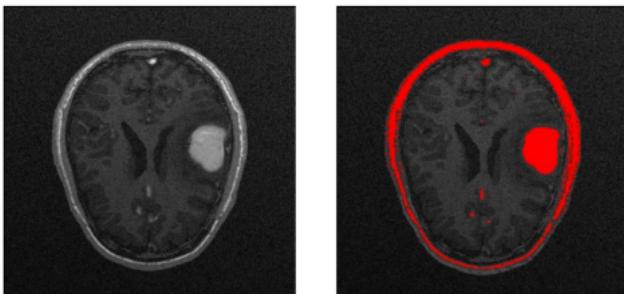
Múltiples canales de entrada

No todo en las imágenes es RGB, sino que puede tener distintos canales de entradas, como en las imágenes satelitales y en tomografías.

Técnicas clásicas de segmentación de imágenes

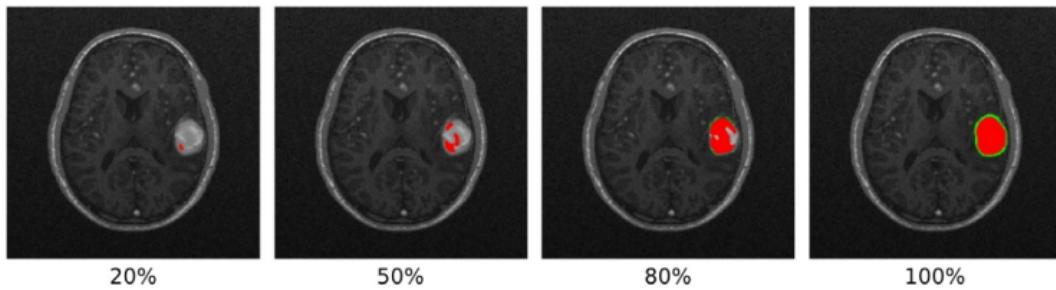
Umbralado

Si supera cierto umbral, califica como tumor al píxel y sino no. Es muy simple y no muy bueno.



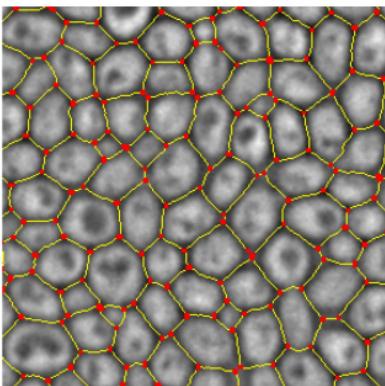
Crecimiento de regiones

Un médico dice donde hay un tumor con una semilla y va creciendo (agregando pixeles vecinos) hasta que se encuentra o se choca con un borde. Es una técnica muy algorítmica.



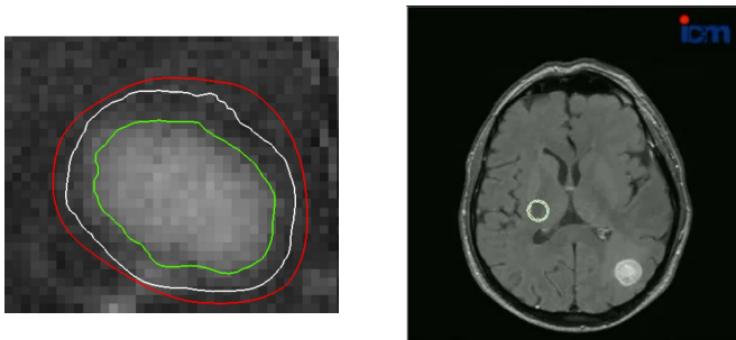
Watershed

Inunda los valles y cuando se tocan bordes, termina una región y empieza otra.



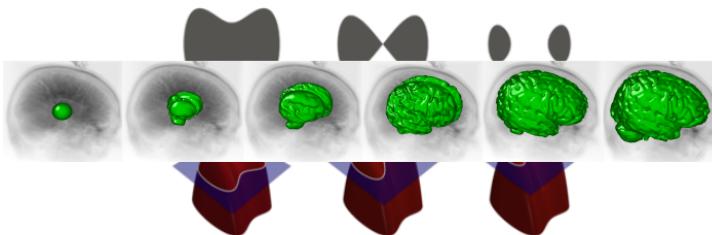
Modelos deformables explícitos

También se los conoce como contornos activos o Snakes. Consiste en una curva elástica aproximada que, colocada sobre una imagen, empieza a deformarse a partir de una forma inicial con el fin de delimitar las regiones de interés en la escena.



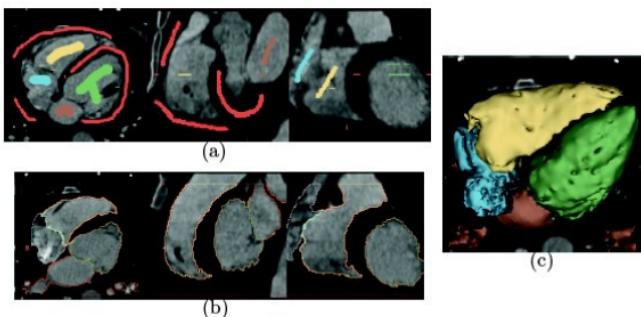
Level-sets

En lugar de representar directamente un contorno como una curva paramétrica $C(t)$, los level sets utilizan una función de nivel $\phi(x,y,t)$ que define una superficie en un espacio de mayor dimensión.



Segmentación interactiva

La segmentación interactiva permite a los usuarios intervenir directamente en el proceso de segmentación para mejorar la precisión de los resultados.

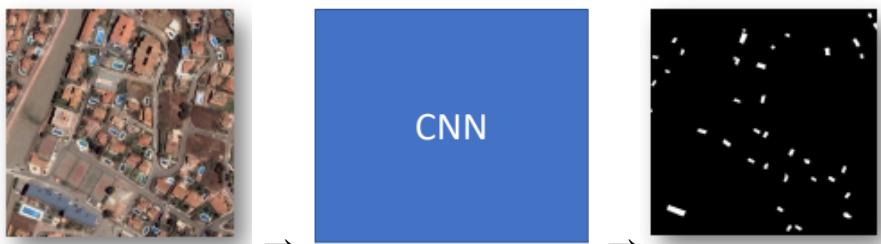


Redes neuronales convolucionales (CNN) para la segmentación de imágenes

Esto viene para mejorar todas las técnicas anteriores. Acá si hay un aprendizaje de algo, antes no usábamos un modelo que aprendía.

Hay una estructura, el dato de train para aprender es la imagen y la segmentación (etiqueta). En el caso de tener 4 clases, cada píxel tiene la probabilidad de ser de cada clase.

Segmentación de imágenes binaria

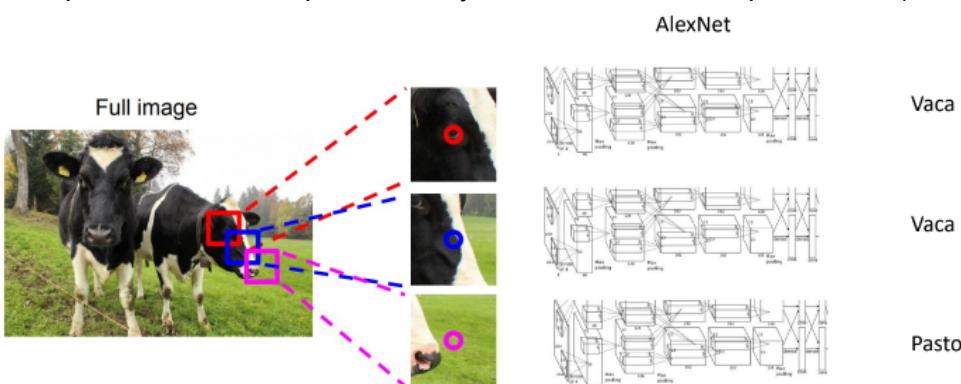


Segmentación de imágenes multiclase



Primera solución: sliding window

Lo que se hace es tener una ventanita, la voy moviendo y clasifico esa ventanita por clase, a diferencia de antes que clasificaba pixel por pixel. Esto puede ser un problema, porque puede ser muy ineficiente ya que no se aprovechan los features compartidos (mando cada parchesito a mi red, cuando voy a mirar cada parche solo me impronta este y no todo el contexto que lo rodea).



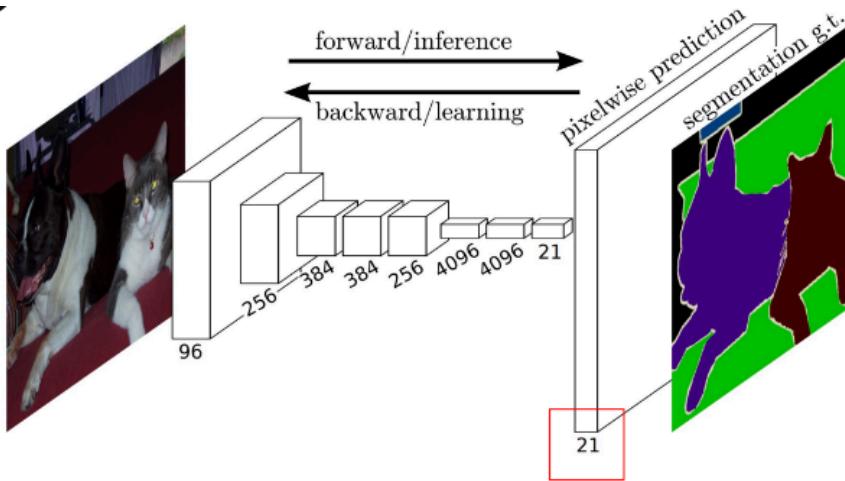
Vemos en la imagen que tenemos tres redes distintas a pesar de que los primeros dos parches calificaron como vaca (no comparten features).

Redes totalmente convolucionales

No hay perceptrones, ninguna capa está totalmente conectada, son totalmente convolucionales.

El último feature map (última capa) tiene tantos canales como posibles clases. Cada pixel es un vector de tamaño cantidad de clases, y en cada espacio tenemos la probabilidad de estar en esa clase. Computamos la función de pérdida para cada pixel y promediamos.

Tenemos que hacer que la imagen de salida tenga el mismo tamaño que la de entrada para poder comparar pixel a pixel. Para esto, voy a tener que extraer los últimos feature maps (en la imagen el chiquito de 21, acá son 21 clases) y llevarlos al tamaño de la imagen original (pixelwise prediction).

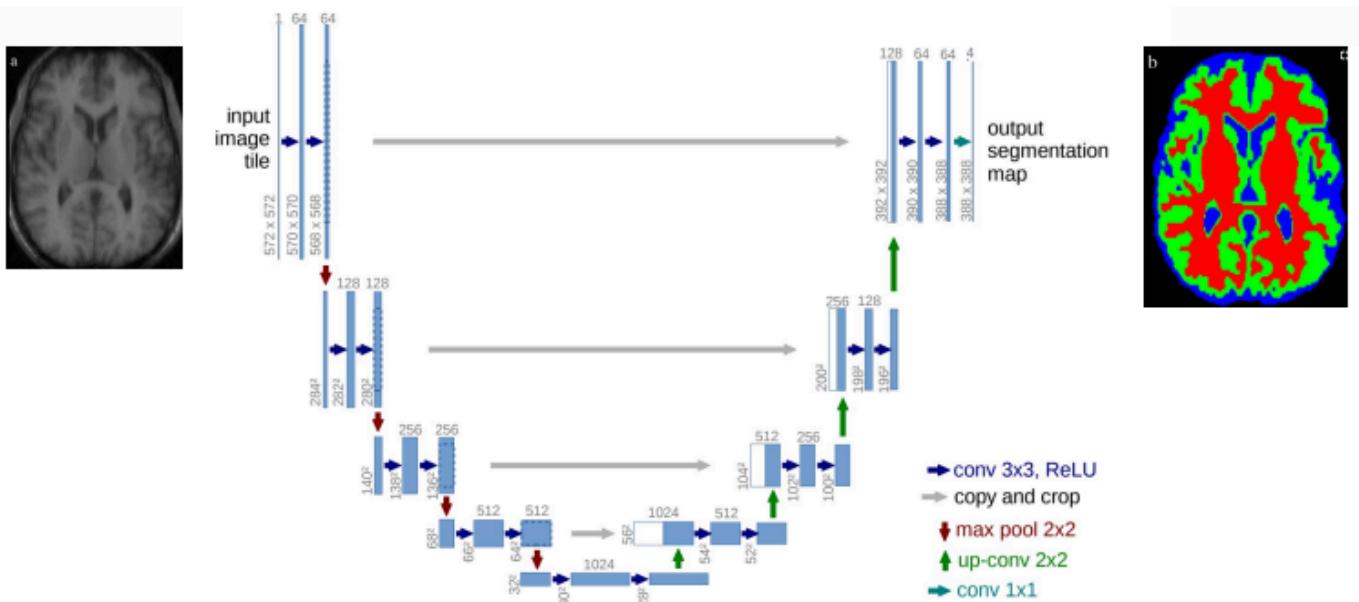


UNet: Arquitectura encoder-decoder para segmentación de imágenes

Es una estructura especial para segmentación de imágenes.

Entra la imagen, hacemos convoluciones (flechas azules), luego hacemos pooling donde bajamos la calidad de la imagen (flecha roja). Luego, vamos subiendo de a poco la resolución con up-convolution (flechas verdes), que es como la inversa de la convolución. Así hasta llegar hasta la resolución de la imagen inicial.

La flechita gris copy and crop sería “cortar” una parte de mi feature maps y lo pego en mi parte de decoder para no perder tanta data de la imagen haciendo los features maps. *Ventajas de hacer este skip-connection:* recupera detalles ya que con up-conv los había perdido. Agarro los detalles y lo concateno con lo que va subiendo, se ve en la parte blanca cuando hago el up-conv son los feature maps que copié del encoder. Esto es porque no hace falta calcular el gradiente de nuevo, lo va recuperando.

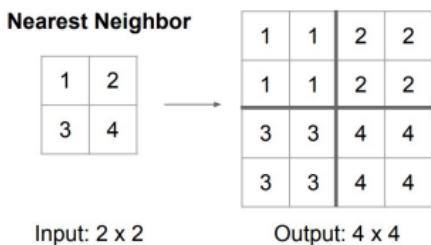


Operaciones de upsampling: Unpooling

Entendemos que es en la parte de up-conv, ahí hacemos unpooling también para conseguir de nuevo la resolución original. En unpooling no hay parámetro a aprender, recién la hay en la parte de la inversa convolucional.

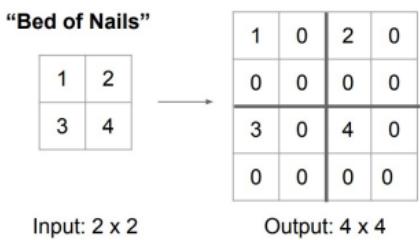
Nearest Neighbor

La idea es reescalar poniendo el mismo número que había en ese espacio 4 veces (que podía haber sido el máximo el average de antes, dependiendo que tipo de pooling había hecho antes).

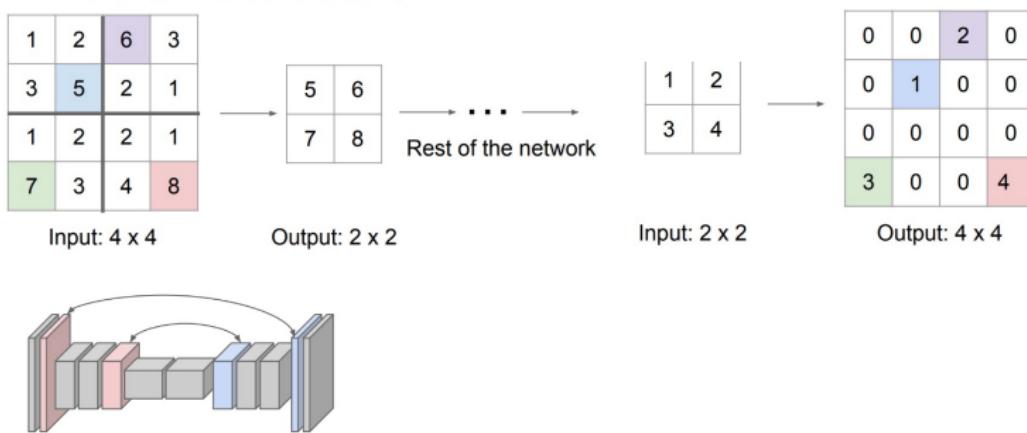


Bed of Nails

La idea es reescalar poniendo el mismo número en el primer espacio y después ceros.



MaxUnPooling



Durante MaxPooling, nos guardamos la posición original del valor máximo.

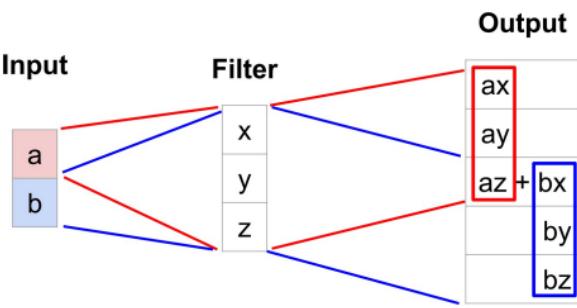
Durante MaxUnPooling, usamos esa posición para poner el valor que quedó y luego ceros.

Transpose convolution

Ejemplo para una convolución 1D.

La idea es: quiero ir de una entrada de tamaño 2 a una de tamaño 5. Voy a tener un kernel que aprendo, y lo que voy a hacer es poner un Stride y un tamaño del kernel de tal forma que me de el tamaño de la salida que quiero. Entra feature map de tamaño 2, sale uno de 5. Es para agrandar, genero una salida aprendida.

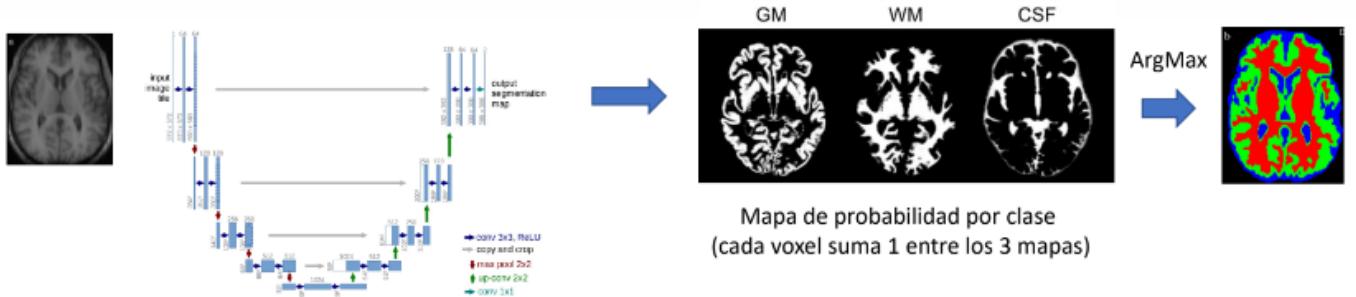
Hago pooling y convolución a la vez. Por cada 1 pixel de la entrada, nos desplazamos 2 en la salida (stride = 2). La salida se construye pesando los valores de la entrada con los distintos pesos de un kernel. En las zonas de overlap, se suma la salida (así me queda de tamaño 5)



Cálculo de la función de pérdida para segmentación

Después de UNet devuelve la pertenencia a cada clase. Luego, se calcula la función de pérdida por pixel (ej. Entropía Cruzada) de los mapas de probabilidad de salida comparándolos con la clase real (la versión One-hot del ground-truth), y luego se promedia por todos los píxeles de la imagen.

La salida son imágenes donde la cantidad de rodajas de pan es cantidad de clases, donde en cada una en cada pixel está la probabilidad de pertenecer en esa clase. A esto se le llama mapa de probabilidad de salida (tercera imagen). → para la predicción, en cada píxel tomamos argMax (entre los mapas) de probabilidad, para saber cuál categoría tenía más probabilidad y eso lo comparamos con el One-hot del ground-truth.



Ventajas de redes totalmente convolucionales

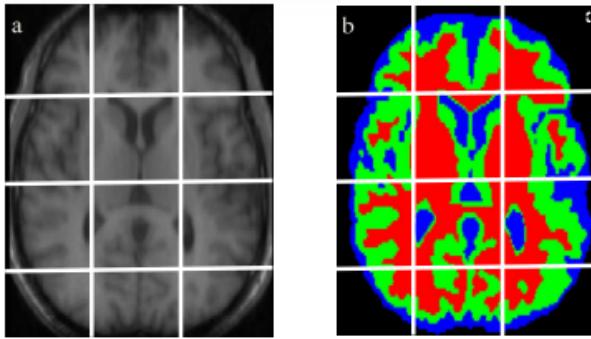
- Las capas densas pueden ser implementadas como capas convolucionales → ¿Cómo?
- La gran ventaja de una red totalmente convolucional es que puede procesar imágenes de **cualquier tamaño**, y puede ser entrenada y evaluada en imágenes de diferentes tamaños. El tamaño de entrada termina siendo igual al de salida
- El tamaño de la imagen nueva tiene que respetar la cantidad de pooling realizados, es decir que hago 3 pooling de 2x2, mi imagen debería ser múltiplo de 8x8 ($2^3 \times 2^3$) ya que si no respeta esta condición al momento que realizamos pooling nos va a quedar un número no entero entonces tendremos que redondear y cuando reconstruyamos la imagen nos va a quedar de una dimensión distinta a la de entrada.

Entrenamiento por parches

Antes, en segmentación, en train, solo había 1 gb para guardar todo el cerebro, por eso parcheábamos para que entre y luego lo reconstruímos. Los mini-batches se componen de parches, no de imágenes completas.

En test time, si la red es totalmente convolucional, basta con insertar la nueva imagen que cumpla con el tamaño adecuado y la predicción será de igual tamaño (cuidado con los múltiplos de), es decir, puedo

testear con imágenes de distintos tamaños (pero que cumplan con el múltiplo de 2^n), esto es porque acá no se requiere tanta memoria porque no se calcula el gradiente. Ahora, si la red no es del tamaño acorde, se puede hacer el parcheo fuera de la red, y luego reconstruir la segmentación



Estrategias de muestreo de parches

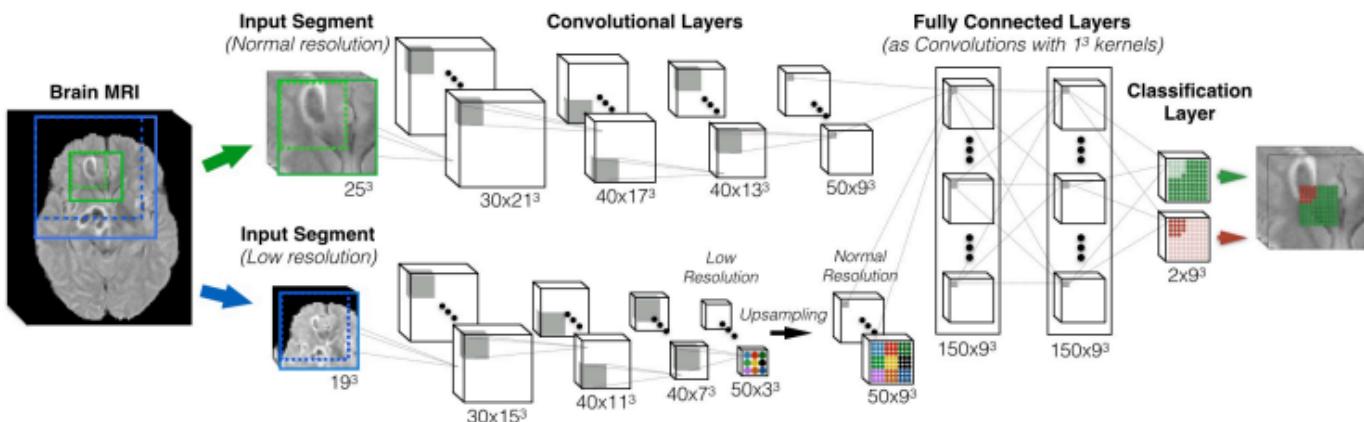
Para garantizar un entrenamiento equilibrado, es importante que cada mini-batch esté compuesto por parches centrados en diferentes etiquetas de forma equilibrada, es decir, no tener por ejemplo en un mini batch solo de una clase, sino tener de clases equilibradas. Ayuda a solucionar el problema del desbalanceo de etiquetas.

Procesando imágenes 3D

Las arquitecturas son las mismas, pero utilizando convoluciones 3D en lugar de 2D. Si la entrada era antes un tensor de dimensiones (BatchSize, Channels, Width, Height,) entonces ahora pasará a ser (BatchSize, Channels , **Depth**, Width, Height).

DeepMedic: ConvNets con branches multi-resolución

??

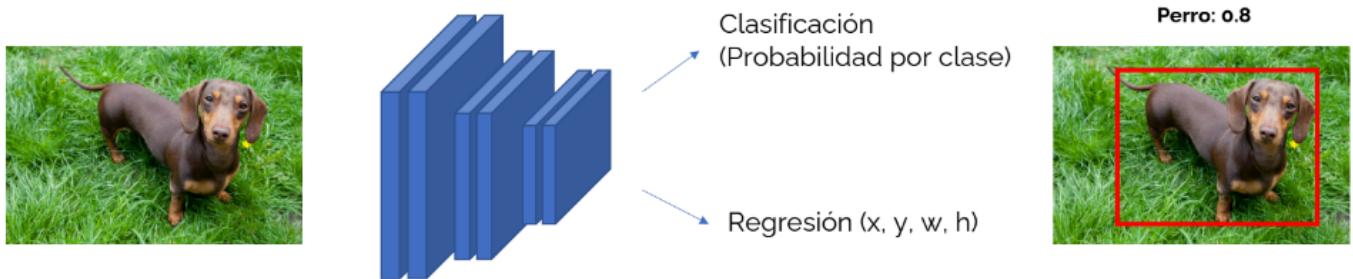


nnU-Net: a self-configuring method for deep learning- based biomedical image segmentation

Construyeron un método de autoconfiguración de la red, quizás no se como armarla, o la cantidad de capas o los hiperparámetros entonces lo configura automáticamente.

Arquitectura básica para localización

Red con salidas múltiples



Tenemos dos problemas:

- clasificación: a qué clase pertenece
- regresión: que parámetros pongo para que la cajita me encierre al perro (me lo localice).

El dataset, ahora, es una imagen con la etiqueta de la clases y la etiqueta de la caja:

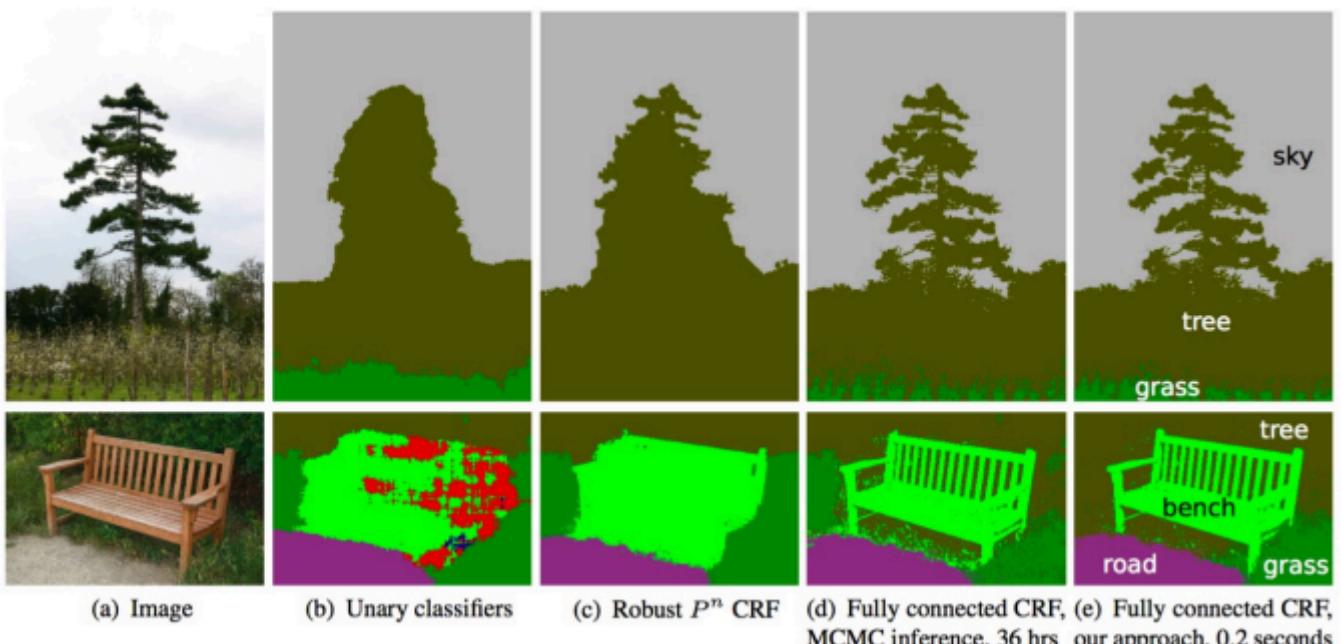
Dataset = (X, Y, c) donde X = imagen, c = clase, Y = bounding box = (x_0, y_0, w, h)

(centro de la caja, ancho, altura)

En la parte de regresión usamos la función de pérdida error cuadrático, para clasificación cross-entropy y después las combinamos (las sumamos).

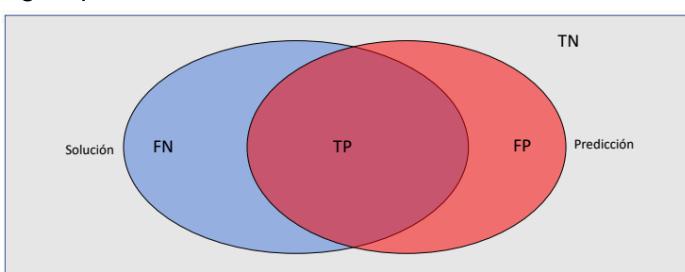
Medidas para evaluar la calidad de la segmentación

Acá vemos cómo medir cuál de las siguientes segmentaciones es mejor



Intuición de cuál es la mejor

Quiero que mi predicción esté totalmente solapada con la solución. La intuición es buscar el área que logra que esto suceda.



Tasa de acierto (accuracy)

Tasa de acierto sobre el total de los píxeles

$$Accuracy = \frac{TP + TN}{FN + TP + FP + TN}$$

Problema: Requiere datos balanceados, entonces no es buena métrica cuando tenemos datos desbalanceados

¿Entonces qué medida usamos?

Medidas de solapamiento

Coeficiente Dice/ F1-Score

Se aplica en datos binarios. El coeficiente de Dice vale 1 cuando el overlapping es perfecto, es decir, no existen FP y FN. Generalmente usamos esta métrica.

$$Dice = \frac{2 | Solución \cap Predicción |}{|Solución| + |Predicción|} = \frac{2 TP}{FN + 2 TP + FP}$$

Coeficiente Jaccard / IoU (Intersection over union)

Es similar al anterior. Se aplica en datos binarios.

$$Jaccard = \frac{|Solución \cap Predicción|}{|Solución \cup Predicción|} = \frac{TP}{FN + TP + FP}$$

Relación entre Dice/F1 y Jaccard/IoU

$$Jaccard = \frac{Dice}{2 - Dice}$$

En general, Jaccard tiende a penalizar las malas clasificaciones más que Dice en términos cuantitativos. Dice y Jaccard están correlacionados positivamente.

Soft Dice as a loss function

El coeficiente de Dice se puede usar como función de pérdida (1- pues cuanto más chico mejor). Se hace por cada feature map y se promedia.

$$Dice = \frac{2 | A \cap B |}{|A| + |B|}$$

$$Loss\ Soft\ Dice = 1 - Dice$$

$$|A \cap B| = \begin{bmatrix} 0.01 & 0.03 & 0.02 & 0.02 \\ 0.05 & 0.12 & 0.09 & 0.07 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{element-wise multiply}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} \xrightarrow{\text{sum}} 7.41$$

prediction target

$$|A| = \begin{bmatrix} 0.01 & 0.03 & 0.02 & 0.02 \\ 0.05 & 0.12 & 0.09 & 0.07 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix}^2 \xrightarrow{\text{sum}} 7.82$$

$$|B| = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}^2 \xrightarrow{\text{sum}} 8$$

$$Dice = \frac{2 * 7.41}{7.82 + 8} = 0.936$$

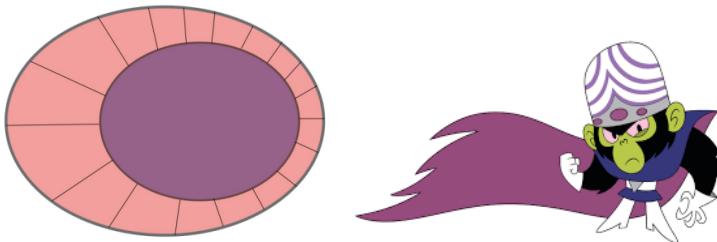
$$Loss Soft Dice = 1 - 0.936 = 0.064$$

Source: <https://www.jeremyjordan.me/semantic-segmentation/>

Medidas distancia entre contornos

Distancia media entre contornos

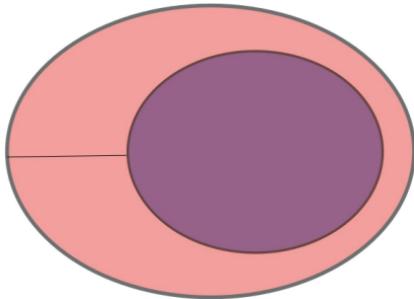
Mide la distancia entre los contornos (una es predicción y otra es la real). Miro el contorno de lo que quiero segmentar (violeta), miro de lo que segmenté realmente (rosa) y voy pixel x pixel del contorno y para cada pixel busco cual es el más cercano en el otro contorno, luego hago un promedio de estas distancias. Esta es 0 cuando los contornos están totalmente overlapeados



Distancia de Haussdorf (máxima distancia entre contornos)

Igual que antes pero acá simplemente en vez de hacer el promedio, tomo la distancia máxima. Cuando tengo pixeles por todos lados, lo que tengo son muchos contornos de la predicción (cada pixel es una parte del contorno nuevo).

La distancia de Hausdorff va a ser muy grande cuando tenemos pixeles outliers (espurios), pero en ese caso el dice me daría perfecto, por lo tanto son "complementarios".

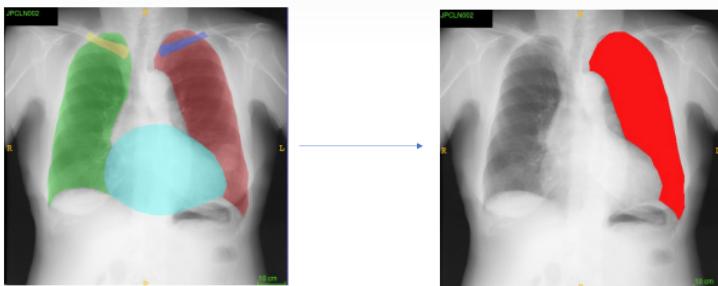


Medidas de calidad en segmentación multiclas

Para cada posible clase $c \in C$, se binariza el problema siguiendo:

- PredicciónBinarizada(x) = 1 si Predicción(x) = c (pertenece a la clase c)
- PredicciónBinarizada(x) = 0 si Predicción(x) $\neq c$ (no pertenece)

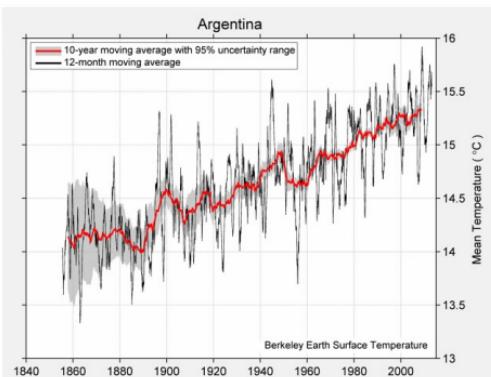
Luego se computa una medida para cada etiqueta $c \in C$ y se analiza el comportamiento por etiqueta y en promedio.



Redes neuronales convolucionales para el procesamiento de series de tiempo

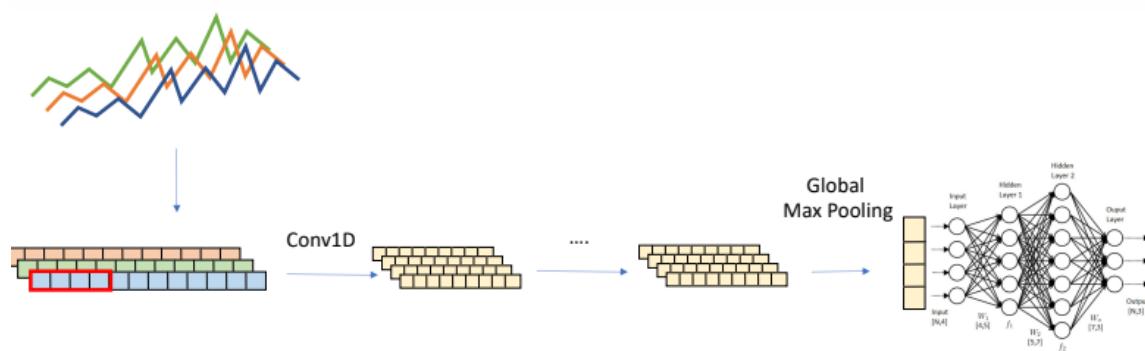
Series de tiempo

Una serie de tiempo es una secuencia de datos observados en momentos sucesivos a lo largo del tiempo. Estos datos suelen estar espaciados a intervalos uniformes, como horas, días, meses o años. Se puede procesar una serie de tiempo con CNN o con perceptrones, aunque no son los mejores modelos. El modelo predice un paso y podemos ir repitiéndolo, a medida que avanza voy cayendo.



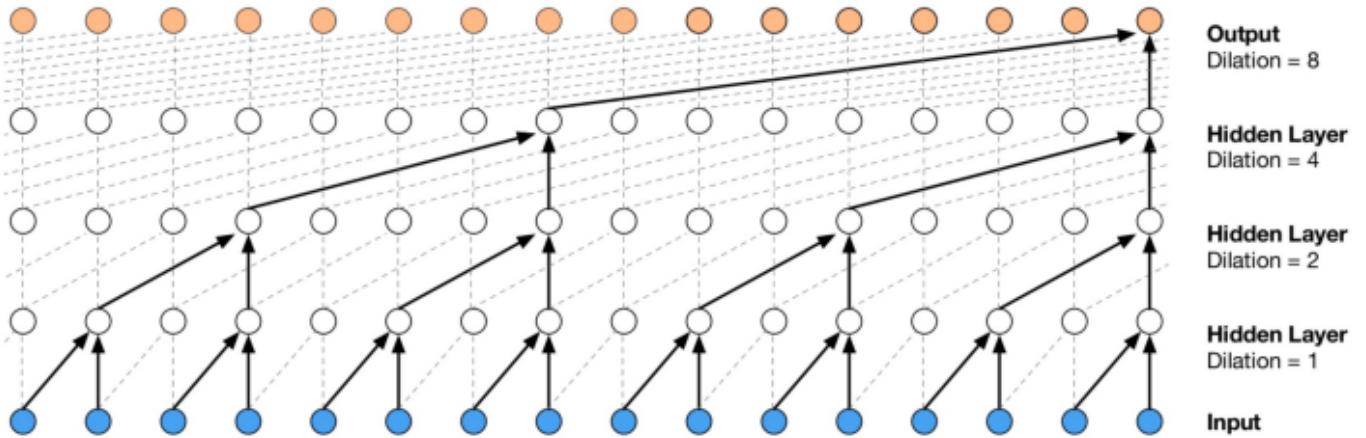
Clasificación de series de tiempo

Queremos ver si la serie de tiempo contiene un evento anormal. Para esto, la procesamos con una red neuronal.



Predictión en series de tiempo: más allá de un paso

Lo que sucede es: tengo datos de entrenamiento con los que entreno. Luego, predigo un dato y esa predicción la uso para seguir entrenando y así sucesivamente.

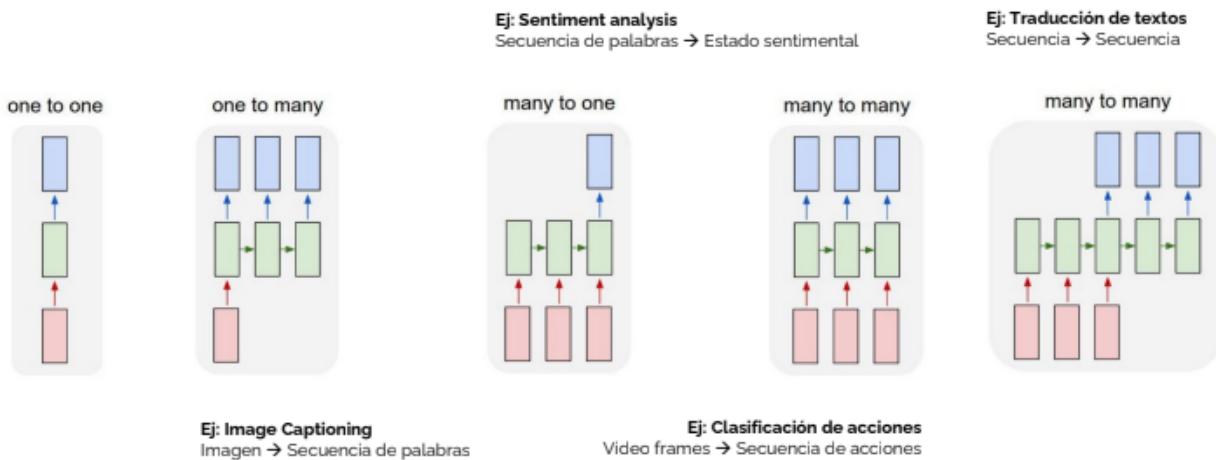


Redes recurrentes

Las redes recurrentes son redes neuronales diseñadas para procesar datos secuenciales. Así como las redes totalmente convolucionales pueden escalar a cualquier tamaño de imagen, las redes recurrentes pueden en principio procesar cualquier longitud de secuencia. La clave son los pesos compartidos (weight sharing) a través del tiempo.

Varios tipos:

- one to one
- one to many
- many to one
- many to many (video)
- many to many (secuencia)



Las redes recurrentes tienen memoria a la hora de calcular los pesos.

Función parametrizada por W (la misma para todos los instantes t)

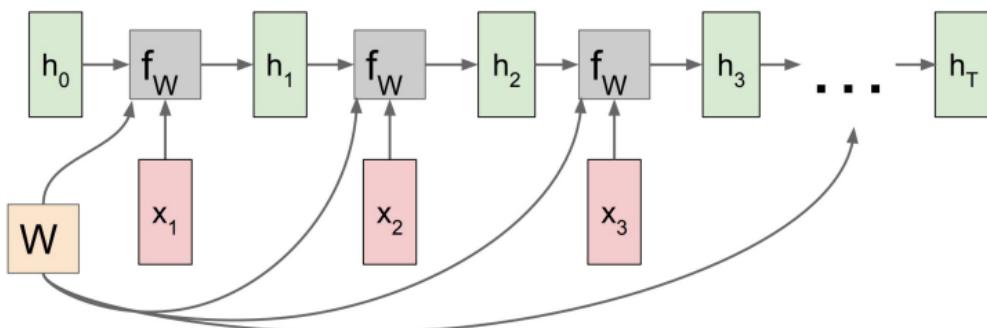
$$h_t = f_W(h_{t-1}, x_t)$$

Nuevo Estado | Antiguo Estado Entrada

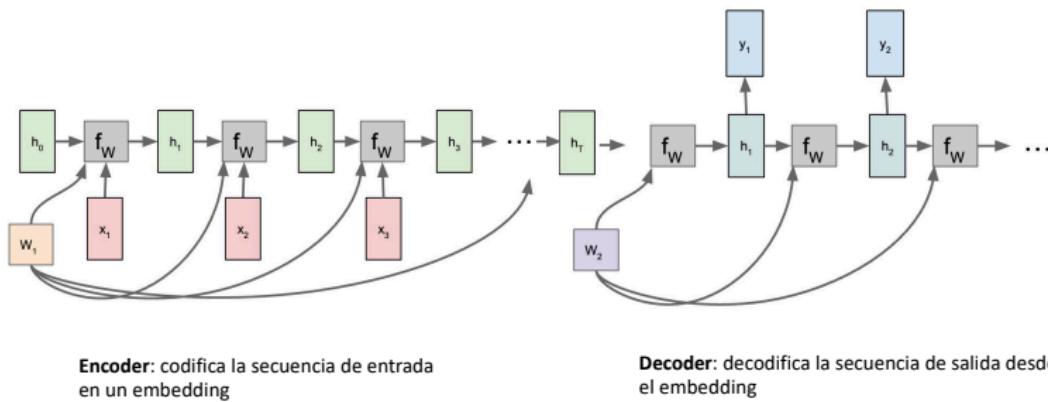
Se entrena mediante Backpropagation through time. 😞

Grafo de cómputo

A través de la modificación de W modificas tus estados.



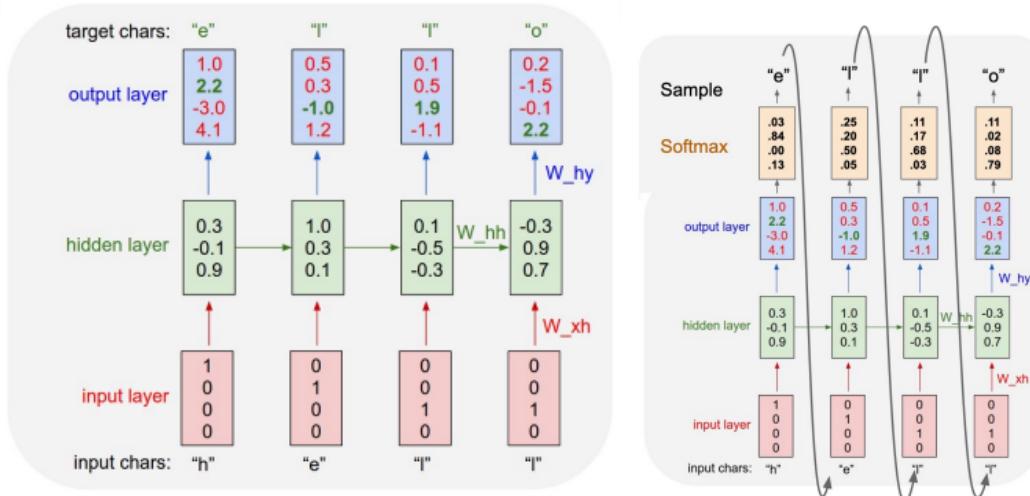
Sequence to Sequence: Many to one + one to many



Encoder: codifica la secuencia de entrada en un embedding

Decoder: decodifica la secuencia de salida desde el embedding

Ejemplo: predicción de caracteres

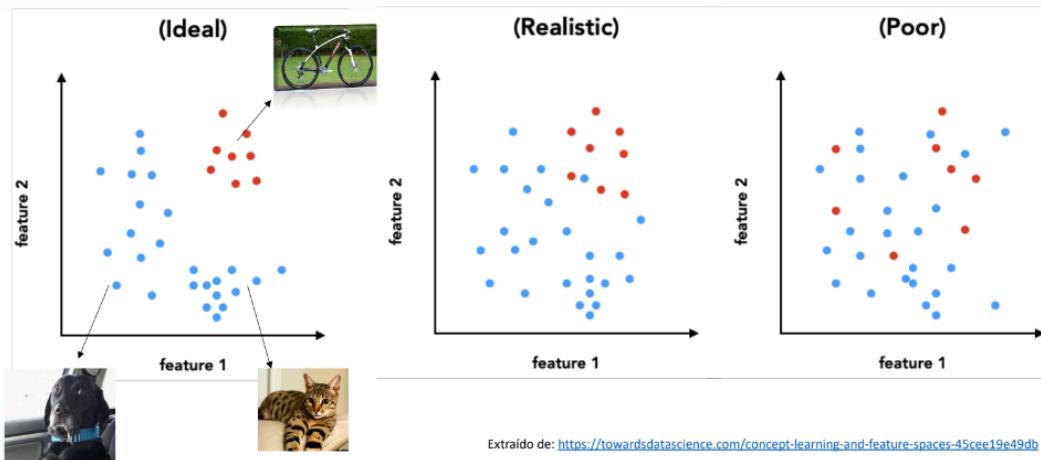


Clase 6: Problemas abiertos en DL y ML

Nuevas representaciones

¿Cómo aprender nuevas representaciones de los datos? Nos referimos a features (como feature maps). Tengo imagen de bicicleta, gato y perro y quiero una buena representación que separe a la bicicleta de los otros dos. Lo que no quiero es que esté completamente mezclado.

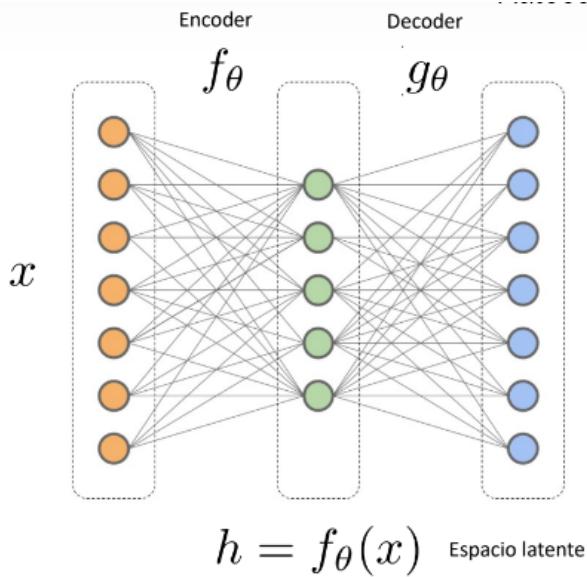
Queremos aprender representaciones buenas de nuestros datos, suelen ser representaciones de baja dimensión que clusterizan. Depende mucho de la tarea que quiero resolver. Lo que hacemos es un embedding, que significa embeber en un espacio al dato que tengo (reduzco la dimensionalidad).



Aprendizaje de representaciones no supervisado

Autocodificadores: pensada para aprender representaciones no supervisadas, la usamos sin etiquetas ya que la etiqueta es el propio dato que entró (entra imagen de un 2, devuelvo imagen de un 2, no un 2). Tengo mi dato x y la idea es que entra en una arquitectura como un cuello de botella (puede haber capas convolucionales). No hay skip connections. Desde ese cuello de botella, aprendo a reconstruir lo mismo que entró.

Tienen un encoder, tita son los pesos w de ese modelo. Lo vamos a entrenar con una función de pérdida muy simple. Lo que termina aprendiendo es una representación de menor dimensionalidad.



→ al espacio latente (capa del medio) le decimos cuello de botella. Todo pasa por este cuello de botella donde se reconstruye la imagen, el tamaño de la entrada es igual al de la salida.

$$L(x; \theta) = (g_\theta(f_\theta(x)) - x)^2$$

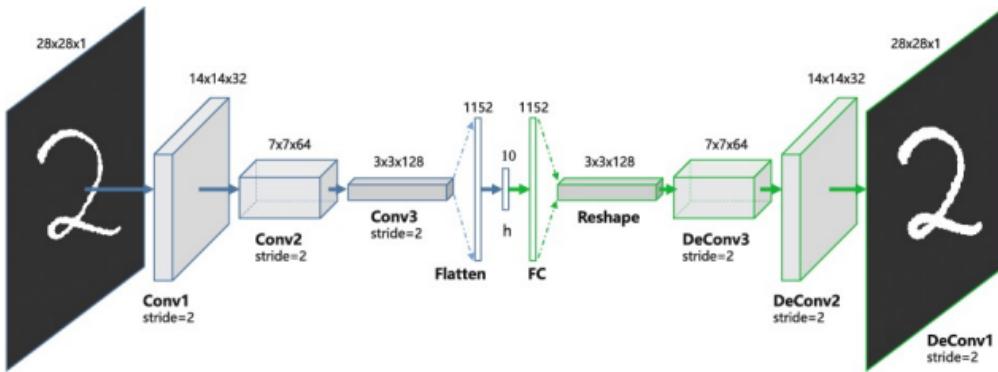
Decoder Encoder Imagen original

Función de pérdida:

Autocodificadores convolucionales

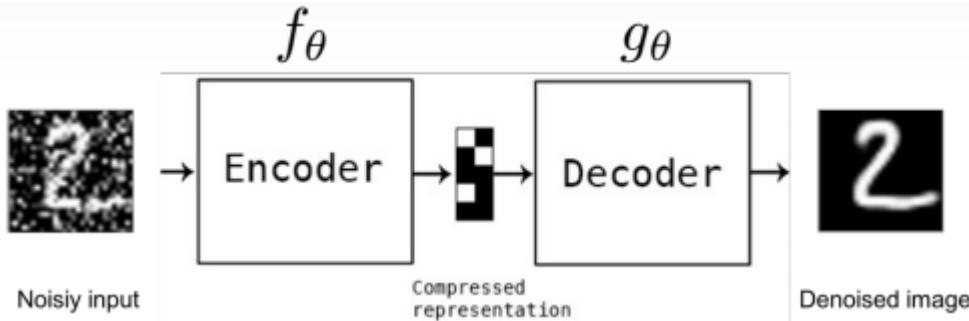
La idea es la misma, tenemos un cuello de botella.

Puedo hacerlo convolucional, metemos en el cuello de botella una fully connected de la cantidad de elementos que quiero tener en mi embedding y a partir de ahí reconstruyo igual que antes. Con esto ya puedo aprender sin tener ninguna etiqueta, meto imagen sale imagen



Denoising autoencoders

Lo mismo pero ahora entra una imagen con ruido y la devuelve sin ruido.



La imagen del 2 con ruido cuando pase por el encoder y pase al cuello de botella va a estar muy cerca de las representaciones del 2 sin ruido, entonces como lo que hace el decoder es pasar del cuello de botella a la imagen reconstruida, va a dar una imagen de un 2 sin ruido porque las representaciones estaban muy cerca, y se supone que las entreno con imágenes sin ruido entonces sale sin ruido.

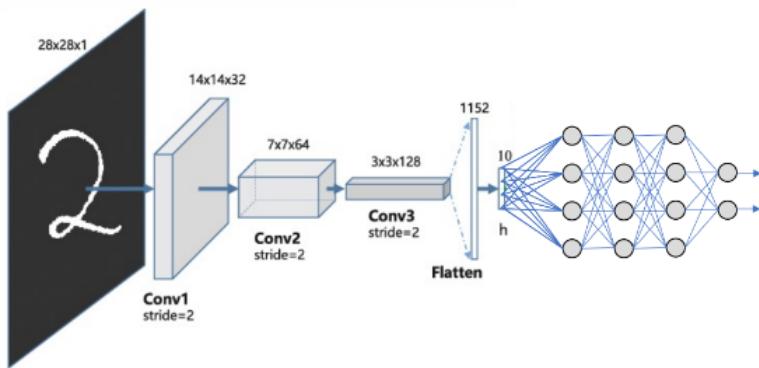
$$L(x; \theta) = \underbrace{(g_\theta(f_\theta(\hat{x})) - x)^2}_{\text{Decoder}} \quad \underbrace{\text{Encoder}}_{\text{Imagen original}} \quad \text{Imagen con ruido}$$

La función de pérdida:

Cuando le agrego ruido, lo muevo de la distribución. Lo que hago es samplear el soporte de la distribución que estoy aprendiendo a modelar, aprendo una representación más suave.

Aplicaciones

Aplicación 1: Inicialización de modelos



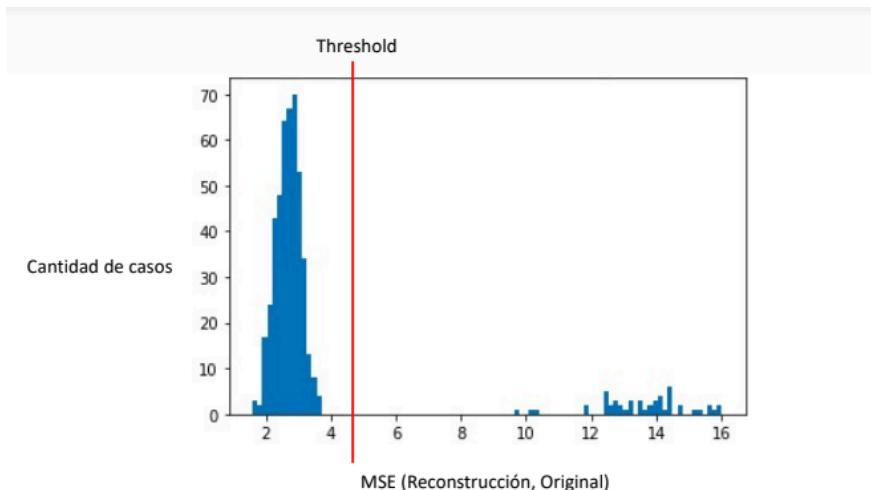
Entreno con datos sin etiqueta y le corto la cabeza, conecto un MLP y hago que aprenda a clasificar los dígitos ahora con datos con etiqueta (aunque muchos menos que antes). Lo que logré con el pretraining es aprender características que son útiles para alguna tarea (hacer denoising, reconstruir el numero, etc) pero estas features fueron aprendidas con un montón de datos (no etiquetados) entonces inicializo mi modelo con los pesos aprendidos en el pre-training y entreno (fine tuneo) con los datos etiquetados continuando así el entrenamiento para la tarea de clasificación.

Aplicación 2: Detección de anomalías

¿Cómo usar un autocodificador para realizar detección de anomalías?

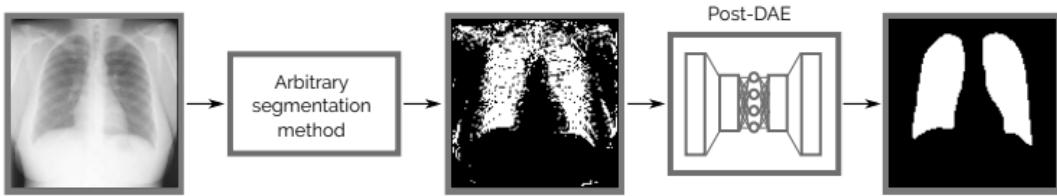
Entreno un autocodificador con casos 'normales', cosas que conoce, es decir, con datos no atípicos. Por ejemplo, entrena con radiografías del tórax, lo que no es "normal" sería la imagen de un gato, no sabría que reconstruir.

Dado un dataset de prueba, lo reconstruyo utilizando el autocodificador → entra imagen sale imagen (esa es mi reconstrucción). Podemos trabajar con el MSE porque todo está armado para una distribución específica, si le meto otra distribución puede ser que no sepa reconstruir o lo que puedo reconstruir no tiene nada que ver con lo que entró. Entonces el MSE es una buena métrica para comparar lo que entró y lo que reconstruí. Por lo tanto, computo el MSE entre la reconstrucción y los datos originales, donde fijamos un umbral para decir que datos son normales y que datos no (outlier)). Es decir, al reconstruir la imagen vemos si se reconstruyó cualquier cosa (era un outlier) o no.



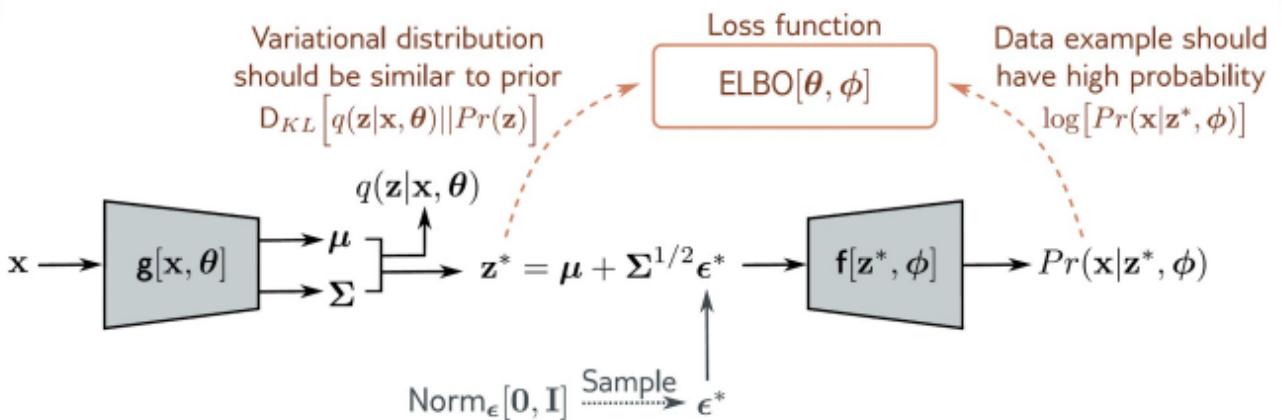
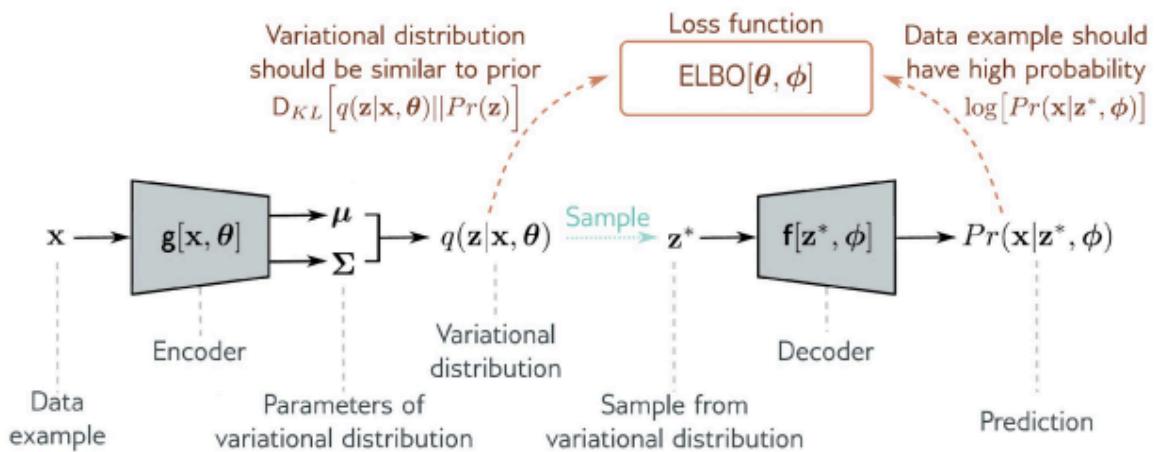
Aplicación 3: Autoencoders como etapa de postprocesamiento

Por ejemplo, segmentación de estructuras anatómicas: Tenemos una imagen, la segmentamos y queda con mucho ruido, la pasamos por el denoising autoencoder (Post-DAE) y me queda una segmentación más suave.



Variational autoencoders

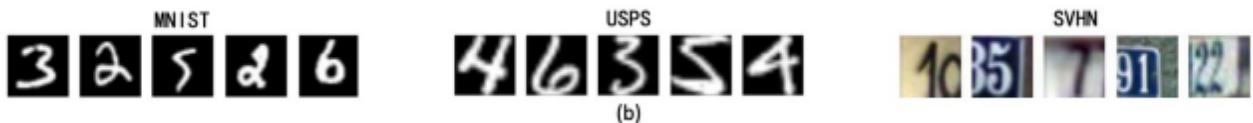
Entra el dato, ahora el encoder tiene 2 vectores, uno con media y otro con varianza. Asumo que es una normal y tengo distribución Q. Sampleo z, no proyectó a vector ahora, proyectó a distribución y se entrena con la función de pérdida ELBO.



¿Cómo generalizar a múltiples dominios de datos?

Adaptación de dominio

Se produce un cambio de dominio (diferente distribución) en los datos de entrada, pero la tarea que queremos resolver es la misma en ambos dominios.

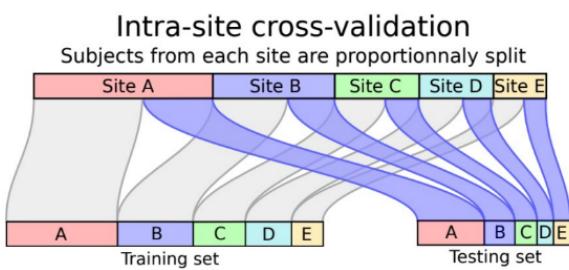


Todos clasifican dígitos pero son diferentes dominios.

Problema multisitio

Si tengo etiquetas en todos los dominios

Solución 1: Entrenar con datos con etiquetas de todos los dominios

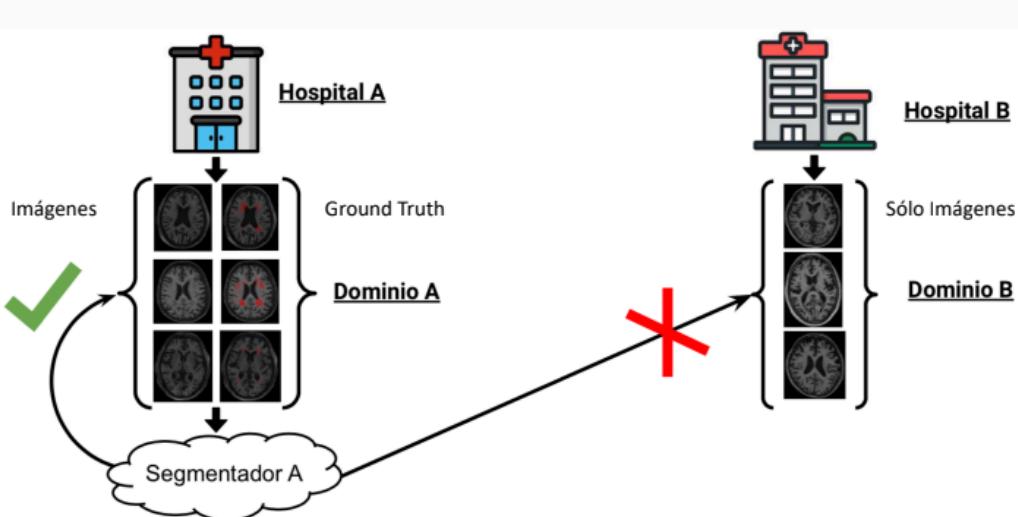


→ me quedan datos de todos los dominios en

training y test (se parece a stratified)

Solución 2: Utilizo transferencia de aprendizaje por medio de fine-tuning. Es decir, hacer un pre-training con un algunos dominios y luego lo finetuneas con algunos dominios nuevos.

Si NO tengo etiquetas en todos los dominios



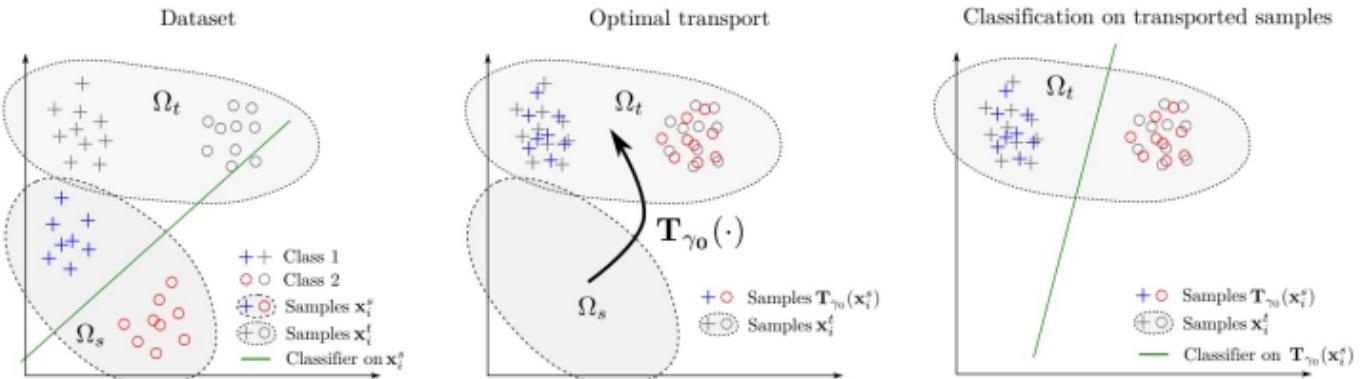
En el hospital A tenemos las imágenes y las etiquetas. En el hospital B solo las imágenes.

Al provenir las imágenes de diferentes dominios, esto quiere decir que el segmentador de A (el modelo) no me va a servir para segmentar imágenes en el hospital B.

Adaptación de dominio NO supervisada

Ideas para resolverlo :

1. Mapear la distribución de los datos del Sitio 1 al Sitio 2, y re-entrenar nuestro modelo.

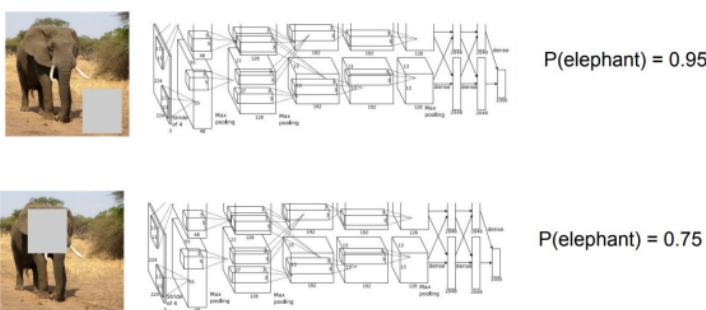


2. Aprendizaje de features invariantes al dominio (no varían en los dominios).

¿Cómo interpretar los modelos entrenados?

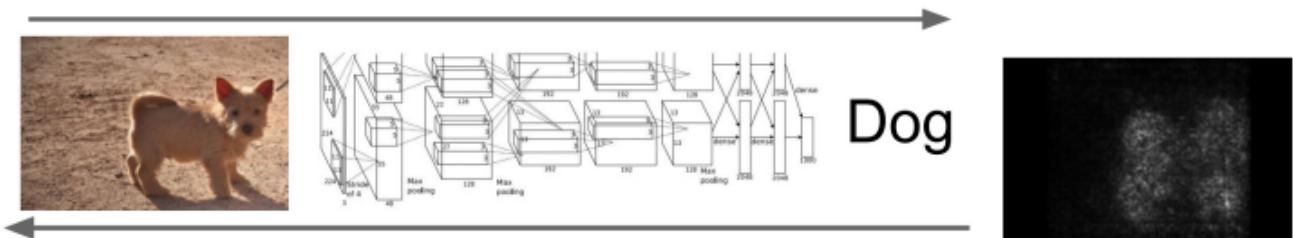
Mapas de saliencia por medio de oclusión

Tengo modelos entrenados y le paso imágenes con cuadrados tapando diferentes partes para fijarme qué es lo que ve mi red neuronal al momento de predecir. La probabilidad saliente me dice la probabilidad de que se encuentre el objeto en la imagen, por lo tanto al taparle la cara al elefante, la imagen tiene menor probabilidad de contenerlo.



Mapas de saliencia por retropropagación

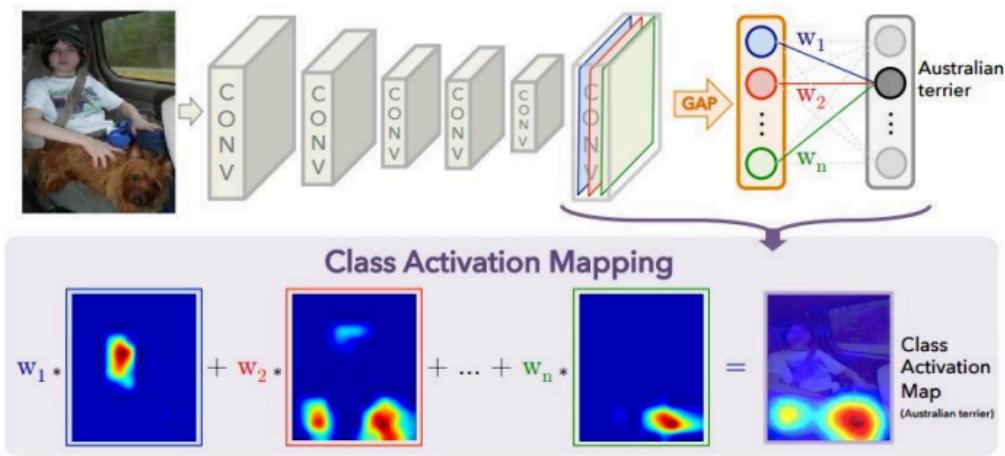
Computamos el gradiente del score de la clase de interés respecto a los píxeles de entrada (acá nos interesa el gradiente del x). No cambia la imagen, hace la pasada forward y en la backward veo el valor de mi gradiente para cada píxel y con eso me armo un mapa donde más blanco, más valor tiene el gradiente, “mi red presta atención aquí para clasificar la imagen”:



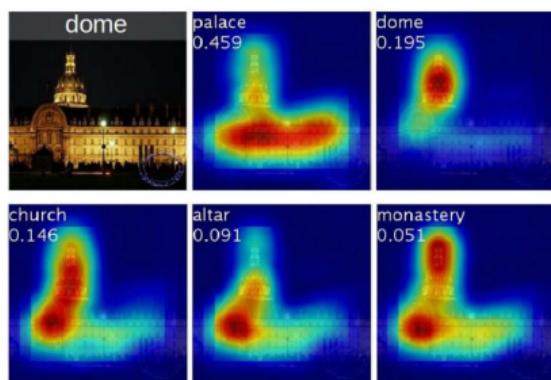
Tomamos su valor absoluto y el máximo por los canales RGB.

Class activation maps

Se necesita tener una sola capa densa post convolución. Me quedo con la última capa convolucional y multiplico por el peso w_1 correspondiente a la última capa.



En la última capa tenemos una feature map por clase, la idea es multiplicar cada feature por su peso w_i y sumarlas, que devuelve una probabilidad que es para saber qué es lo que la red le dió más bola para predecir cada clase.



¿Cómo evitar el sesgo en los modelos de deep learning?

Sesgo en los modelos de IA

Existen sesgos de géneros en traducción de textos, asume que se habla de hombres. No diferencia novia indú de un disfraz.

Sesgo en el reconocimiento de rostros: había mucho error en clasificar hombres y mujeres negros en comparación a blancos.

Datos sesgados: los sesgos en los datos a menudo reflejan desequilibrios profundos y ocultos en las infraestructuras institucionales y las relaciones de poder social (siempre hay más datos de hombres blancos, como Pedro 🧑).

Atributos protegidos

Son variables contra las cuales queremos proteger el sesgo algorítmico como género, etnia, país de procedencia, edad, etc. Por ejemplo, en un algoritmo que recomienda si deberíamos o no otorgarle un crédito a una persona, el género podría ser una variable protegida porque no queremos que la tenga en cuenta. Es decir, es el atributo que no vamos a querer que el modelo use para clasificar.

Definición formal de “fairness” en IA

Métrica: Paridad demográfica

La clasificación tiene que ser independiente del atributo protegido.

$$P(\hat{Y} = 1 / A = 0) = P(\hat{Y} = 1 / A = 1) \rightarrow \text{no importa mi atributo protegido, la predicción me da igual}$$

donde:

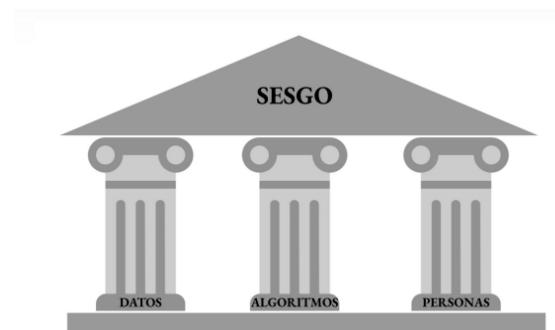
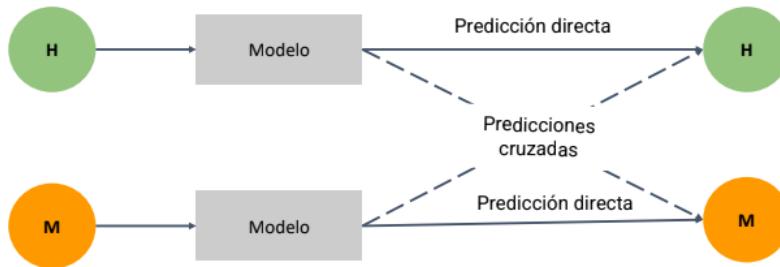
- A = Atributo protegido
- \hat{Y} = Predicción
- Y = Ground Truth

Ej: La probabilidad de darle un crédito a una persona de género femenino es la misma que a una persona de género masculino. No le doy importancia al atributo género.

Es difícil que pase esto, porque si lo saco puede pasar que haya correlación del atributo que saque con otros. Por ejemplo, si tenemos atributo protegido hombre o mujer (años 80's), y lo sacamos, pero nos queda como atributo la altura (que suele ser mayor en hombres que en mujeres), y esta influye en las probabilidades de nuestras predicciones, no se cumpliría la métrica anterior, aún sin tener el atributo protegido.

¿Qué pasa con el balance de datos?

Experimento: Utilizamos imágenes de una base datos pública de 112.000 radiografías torácicas [Wang 2017] con diagnósticos e información demográfica disponible. Generamos particiones de entrenamiento independientes para género masculino y femenino, con igual cantidad de pacientes por enfermedad y género (48568).



→ Imagen para 🙄. El sesgo puede ser de los datos, del algoritmo o de las personas.

Clase 7: Procesamiento del Lenguaje Natural, Transformers y LLMs

Procesamiento del Lenguaje Natural

El Procesamiento del Lenguaje Natural se ocupa de las interacciones entre las computadoras y los idiomas humanos (naturales). Implica desarrollar algoritmos y modelos que puedan procesar, analizar y generar lenguaje humano.

Clasificación de Texto

Asignar una categoría predefinida a un texto dado.

Ejemplo: Clasificar correos electrónicos como 'spam' o 'no spam'.

Análisis de Sentimientos

Determinar la actitud o el tono emocional de un texto.

Ejemplo: Analizar opiniones en redes sociales para saber si los comentarios sobre un producto son positivos, negativos o neutrales.

Reconocimiento de Entidades Nombradas (NER)

Identificar y clasificar entidades mencionadas en un texto en categorías como nombres de personas, organizaciones, lugares, etc.

Ejemplo: Extraer nombres de empresas y ubicaciones de un artículo de noticias.

Traducción Automática

Convertir texto de un idioma a otro.

Ejemplo: Traducir un documento del inglés al español.

Resumen de Texto

Generar un resumen breve de un texto largo.

Ejemplo: Crear un resumen de un artículo de investigación.

Generación de Texto

Generar texto coherente y relevante a partir de una entrada dada.

Ejemplo: Completar automáticamente una frase o párrafo en un procesador de textos.

Respuesta a preguntas

Descripción: Responder preguntas específicas

Ejemplo: Responder preguntas de un usuario utilizando información contenida en una base de datos de conocimientos.

Análisis de sentimientos

Evaluar la polaridad de las emociones en un texto.

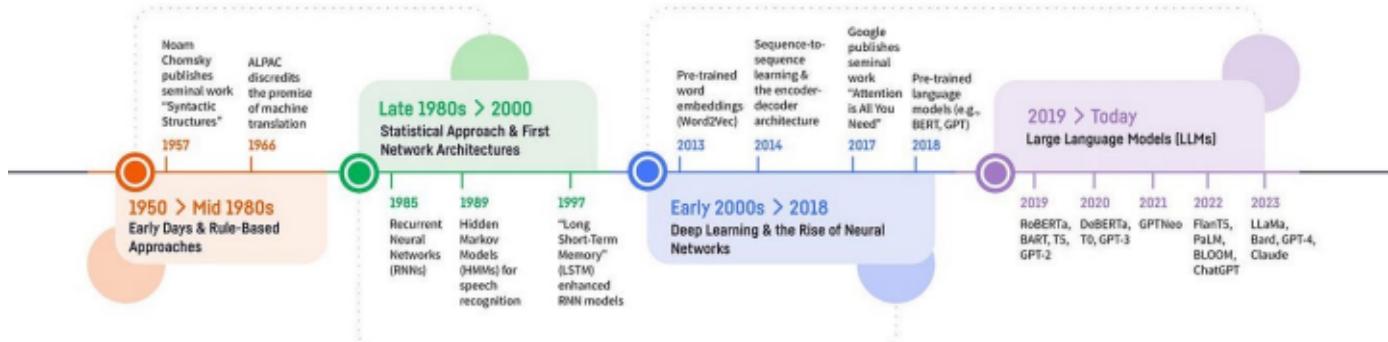
Ejemplo: Determinar si una reseña de producto es positiva, negativa o neutral.

Desambiguación Semántica

Determinar el significado correcto de una palabra que tiene múltiples significados según el contexto.

Ejemplo: Determinar si la palabra 'banco' se refiere a una institución financiera o a un asiento en un parque.

Breve historia de los modelos de lenguaje



Tokenization

Es una forma de convertir palabras en representaciones numéricas. Corta el texto en unidades que pertenecen a un vocabulario de posibles tokens.

Sample Data:

"This is tokenizing."

Character Level
[T] [h] [i] [s] [i] [s] [t] [o] [k] [e] [n] [i] [z] [i] [n] [g] [.]
Word Level
[This] [is] [tokenizing] [.]
Subword Level
[This] [is] [token] [izing] [.]

Character Level: Toma a los tokens como caracteres. Permite modelar cualquier palabra, pero requiere que el modelo aprenda las relaciones entre caracteres, lo cual puede ser muy complicado ya que se pierde el contexto del texto.

Word level: Tomamos las palabras como tokens. El problema es que si las palabras no existen en el corpus de entrenamiento, no las puede modelar. Además, las palabras que terminan en diferente sufijo son modeladas independientemente (e.g. walk, walked) y en realidad yo querría que sean similares.

Subword level: Tomamos palabras, caracteres y partes de palabras como tokens. Es un buen compromiso entre modelar a nivel carácter y a nivel palabra. El vocabulario final incluye tanto palabras como fragmentos. Requiere de un proceso de entrenamiento (token learner) y de segmentación (token segmenter). Es el más usado en la práctica para LLMs.

- Algoritmos conocidos de tokenizadores son:

- Byte-Pair Encoding (Senrich et al, 2016)
- Unigram Language Modeling (Kudo, 2018)

Byte-Pair Tokenizer

Veamos el algoritmo del tokenizer:

1. Empieza con un vocabulario compuesto por todos los caracteres individuales.
 2. Examina el corpus de entrenamiento eligiendo los dos símbolos más frecuentes, y los combina en uno nuevo.
 3. Reemplaza todas las ocurrencias de ambos caracteres por el nuevo símbolo
- Itera k veces, agregando k nuevos símbolos a los caracteres originales

Ejemplo de tokenizer (token learner) con 18 palabras en corpus de entrenamiento:

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	Frecuencia = 9
2 n e w _	
corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

merge	current vocabulary
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, __)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

Estos últimos tokens son los tokens finales del corpus de entrenamiento.

Luego, el token segmenter se aplica en el corpus de test (nuevo texto) iterativamente, fusionando en el orden de aparición

1. Se segmentan las frases en caracteres individuales
2. Reemplazo cada instancia de (e,r) por (er) ya que es el más frecuente
3. Reemplazo cada instancia de (er,_) por (er_)

Así sucesivamente voy reemplazando los tokens más frecuentes

Al final, “newer_” (palabra muy frecuente) será un solo token mientras que “low” “er_” será dos tokens (baja frecuencia) porque se encuentran más frecuentemente separados que juntos.

Finalmente, me queda mi corpus de prueba segmentado por los tokens que aprendió en training.

Word Embeddings

¿Cómo podemos representar las palabras para procesarlas con modelos neuronales?

Notación one-hot!

Los tokens son horizontales (fila es un vector one-hot). Además, las columnas son el orden en el que aparecen las palabras:

The quick brown fox jumped over the brown dog

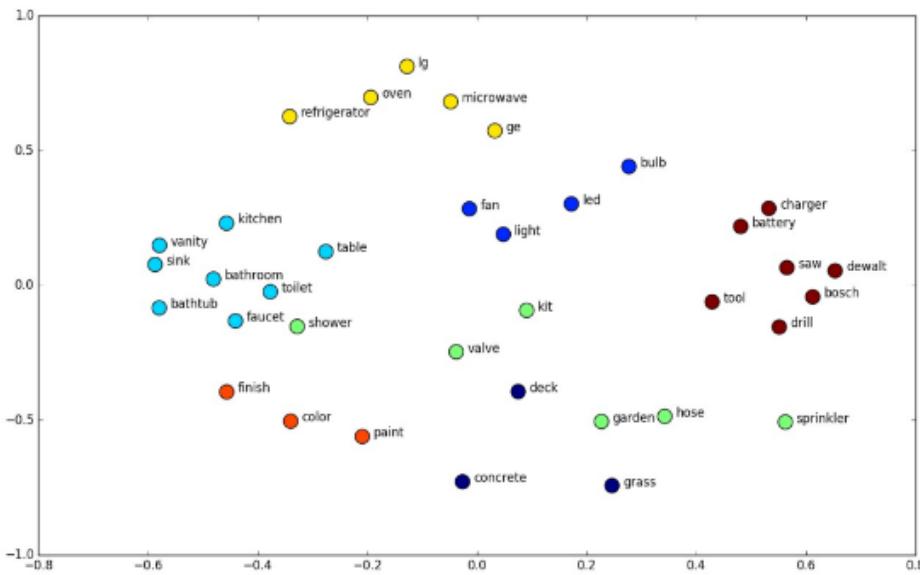


time	The quick brown fox jumped over the brown dog										... kangaroo house
	cat	the	quick	brown	fox	jumped	over	dog	bird	flew	
	0	1	0	0	0	0	0	0	0	0	0 0
	0	0	1	0	0	0	0	0	0	0	0 0
	0	0	0	1	0	0	0	0	0	0	0 0
	0	0	0	0	1	0	0	0	0	0	0 0
	0	0	0	0	0	0	1	0	0	0	0 0
	0	0	0	0	0	0	0	1	0	0	0 0
	0	1	0	0	0	0	0	0	0	0	0 0
	0	0	0	1	0	0	0	0	0	0	0 0
	0	0	0	0	0	0	0	0	1	0	0 0

Dictionary Size = K

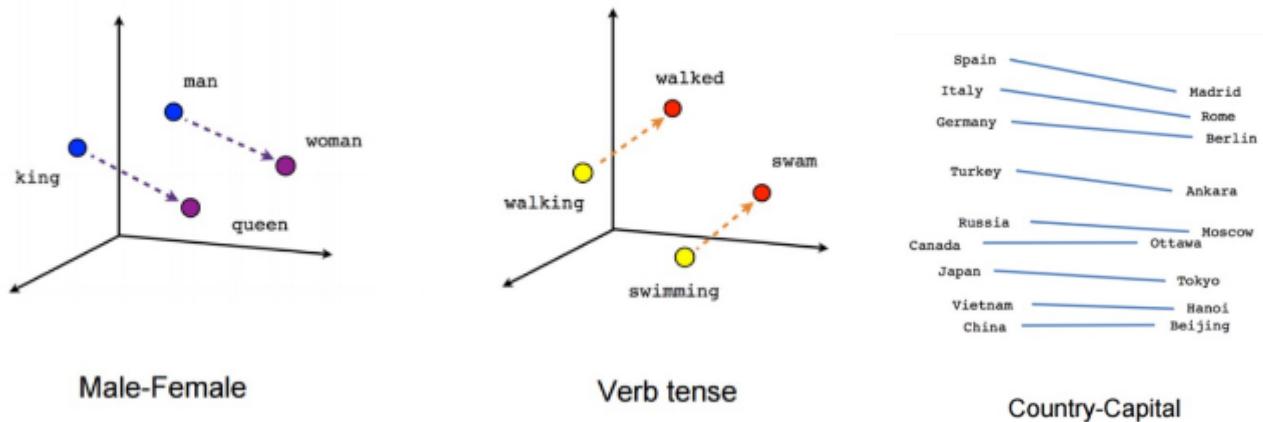
¿Qué sentido tiene la distancia entre palabras representadas con notación one-hot? No tiene sentido de distancia, necesitamos que las palabras que sean parecidas estén “cerca” pero con notación one hot no lo están.

Solución: Word Embeddings



Hacerle un word embedding a un vector one hot es para bajarle la dimensión y poder “acercar” las palabras que se parecen. Estos nuevos vectores suelen ser cortos (dimensión usualmente varía entre 50 y 1000). Son densos (distinto de la notación one-hot que es esparsa), es decir, no solo tienen 0s y 1s.

Midiendo similitud semántica



Una de las métricas de similitud es la similitud coseno (mido el ángulo de los vectores). Se mueve entre -1 (apuntan en dirección opuesta) y 1 (misma dirección). 0 indica ortogonalidad. Me permite ver que tan similares son dos vectores.

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \cdot \sqrt{\sum_{i=1}^n B_i^2}},$$

Esto lo vamos a representar con una matriz de embeddings.

Matriz de embeddings: al multiplicar por el vector one-hot x_n , recupera el embedding correspondiente v_n (esta matriz de distancias).

$$\begin{array}{c} D \times 1 \text{ (Embedding dim)} \quad \mathbf{v}_n = \mathbf{E} \mathbf{x}_n \quad K \times 1 \text{ (Dictionary Size)} \\ \mathbf{E} \quad D \times K \end{array}$$

donde \mathbf{E} es aprendida a partir de un corpus de datos y \mathbf{x}_n es el vector one-hot.

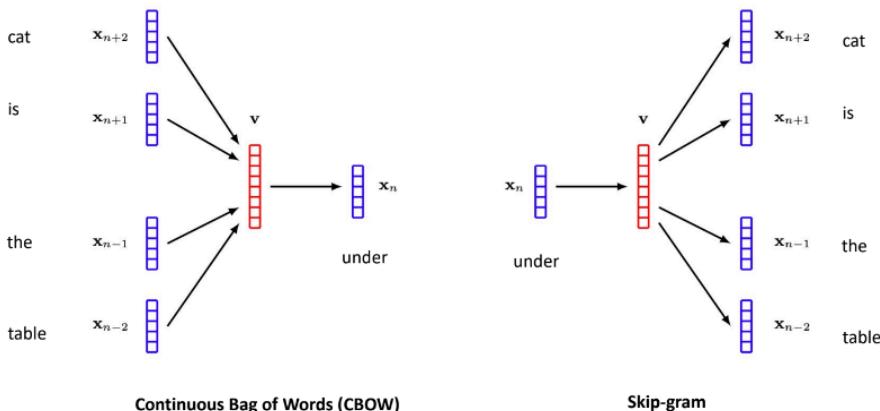
Word2Vec

Es una arquitectura de red neuronal para aprender embeddings. Puede ser visto como una red neuronal de dos capas. Aprende embeddings estáticos (no cambian con el contexto de las palabras, por ejemplo, no diferencia las palabras banco según el contexto), distinto a los embeddings contextuales que veremos luego, donde el embedding cambia en función del contexto. La idea es que palabras que aparezcan rodeadas del mismo contexto, tendrán embeddings similares (los vectores sean parecidos).

La razón por la que los embeddings son estáticos es porque el valor de cada neurona en (v) mira cada token independientemente para calcular su valor y por lo tanto no tiene en cuenta el contexto en el que aparece.

CBOW: no le importa el orden de las palabras. Voy a separar mi texto en ventanitas por ejemplo de 5 tokens. Entrenamos modelo tapando la palabra central que es la que trato de predecir a partir del contexto (las otras 4 palabras). Son 2 capas neuronales, como no me importa el orden, sumo los 4 embeddings, a ese vector lo proyectó y después sale la salida. Lo entreno con entropía cruzada, softmax, etc. y aprendo la matriz \mathbf{E} .

Skip Gram: Toma la palabra para predecir el contexto, de una palabra tiene que predecir 4. El número de palabras de contexto es un hiperparámetro. Además, otro hiperparámetro es qué palabra uso para predecir el contexto (si uso la primera, la del medio, etc).

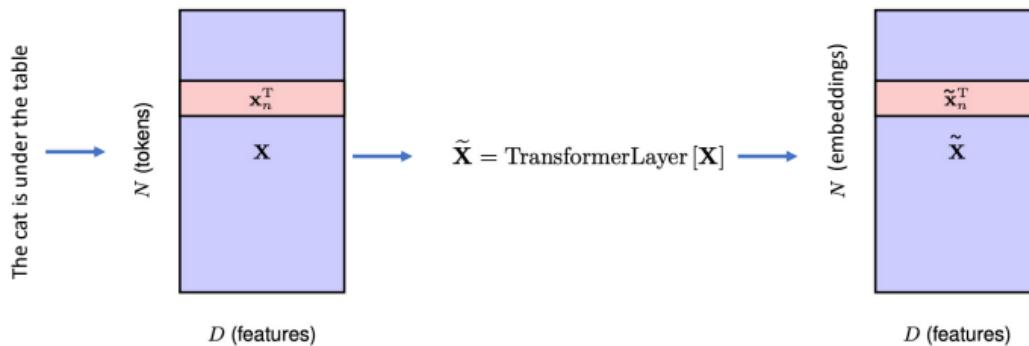


Transformers

Vimos que los embeddings tenían problemas (como lo del banco), por esto ahora el objetivo es transformar este texto en una nueva representación útil para tareas que me surjan.

En principio empiezan como tokens y van a una nueva matriz. Lo que cambia ahora es que las palabras están afectadas por el contexto. Me devuelve nuevas representaciones para cada uno de los tokens. La idea es apilar muchas capas.

Una capa Transformer



Podemos luego apilar múltiples capas para obtener representaciones más ricas donde los tokens incorporen información por medio de 'prestar atención' a otros tokens.

Atención

1. La entrada puede ser muy larga, por lo que la representación puede también serlo
Ej: embeddings de 1024 (el tamaño de la representación del embedding del token) x 37 tokens = 37888 features → MLP? va a ser muy pesada la red, entonces no sirve
2. Frases de diferente longitud (por ejemplo con el chat GPT). Por lo tanto, sería ideal que se compartan los parámetros (similar a una CNN).
3. Ambigüedad del lenguaje. La palabra 'it' necesita contexto para saber qué representa → Atención

$$\mathbf{x}_1, \dots, \mathbf{x}_N \longrightarrow \text{TransformerLayer} [\mathbf{X}] \longrightarrow \mathbf{y}_1, \dots, \mathbf{y}_N$$

Los \mathbf{y}_i contienen información de los \mathbf{x}_i (son una combinación lineal), donde prestar más 'atención' a unos tokens que otros (destaca las palabras que dan el contexto).

$$\mathbf{y}_n = \sum_{m=1}^N a_{nm} \mathbf{x}_m$$

Coeficiente de atención del token de salida n al de entrada m

$$a_{nm} \geq 0$$

$$\sum_{m=1}^N a_{nm} = 1$$

Por ejemplo,



→ Los tonos de verde representan la importancia que se le dio al token en la salida.

Cuando estos coeficientes se determinan a partir del mismo token de entrada → Self-attention

La tríada Query – Key – Value (??)

Conceptos provenientes del campo de Information Retrieval.

Veamos que es cada uno:

- Query: vector que define qué información se necesita (lo que quiero). Las características que quiero buscar en los pokemons
- Key: La asociación de lo que pido en la query
- Value: Lo que devuelvo

En nuestra primera versión de self-attention, usaremos las inputs x_i como keys y queries.

¿Cómo comparar si una key y una query son similares? con el siguiente producto escalar $a_{nm} = x_n^T x_m$

$$\rightarrow a_{nm} = \cancel{x_n^T} \cancel{x_m}$$

→ esto no cumple las características anteriores, donde pedíamos que sea mayor a 0 y que la suma sea 1.

Dot-product self attention

$$a_{nm} = \frac{\exp(x_n^T x_m)}{\sum_{m'=1}^N \exp(x_n^T x_{m'})}$$

Por esto, queremos normalizar con un softmax, así cumple las condiciones:

Pasándolo a notación matricial:

$$\mathbf{Y} = \text{Softmax} [\mathbf{X} \mathbf{X}^T] \mathbf{X}$$

$N \times D$

$N \times D \quad D \times N \quad N \times D$

→ las a_{nm} son las que me dan $A = \mathbf{X} \mathbf{X}^T$

Pero acá no tenemos nada para entrenar, entonces:

Los agregamos!

$$\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$$

$\mathbf{N} \times D$ $\mathbf{N} \times D$ $D \times D$ → agregamos los parámetros entrenables

$$\mathbf{Q} \quad \mathbf{K}^T \quad \mathbf{V}$$

$$\mathbf{Y} = \text{Softmax} \left[\mathbf{X} \mathbf{U} \mathbf{U}^T \mathbf{X}^T - \mathbf{X} \mathbf{U} \right]$$

¿Para qué compartir pesos entre Q, K y V? Puedo tener tres matrices distintas para aprender para cada uno.

$$\mathbf{N} \times D_k \quad \mathbf{Q} = \mathbf{X} \mathbf{W}^{(q)}$$

$$\mathbf{N} \times D_k \quad \mathbf{K} = \mathbf{X} \mathbf{W}^{(k)}$$

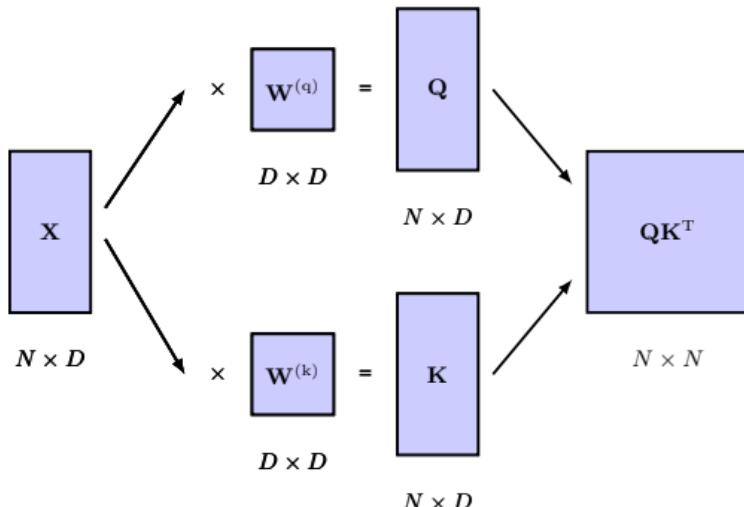
$$\mathbf{N} \times D_v \quad \mathbf{V} = \mathbf{X} \mathbf{W}^{(v)}$$

Y me queda:

$$\mathbf{Y} = \text{Softmax} \left[\mathbf{Q} \mathbf{K}^T \right] \mathbf{V}$$

$$\mathbf{N} \times D_v \quad \mathbf{N} \times D_k \quad D_k \times N \quad \mathbf{N} \times D_v$$

Veamos las dimensiones de las matrices:



Problema: los gradientes de Softmax se hacen muy pequeños para grandes valores de \mathbf{QK}^T → Para solucionar esto lo escalamos: dividimos a cada elemento de la matriz \mathbf{QK}^T por $\sqrt{D_k}$ es decir por la raíz cuadrada de la dimensión de la matriz $\mathbf{XW}^{(q)} = \mathbf{Q}$ (lo mismo para k)

$$\mathbf{Y} = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \text{Softmax} \left[\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{D_k}} \right] \mathbf{V}$$

Algoritmo de lo que dijimos:

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$
 Weight matrices $\{\mathbf{W}^{(q)}, \mathbf{W}^{(k)}\} \in \mathbb{R}^{D \times D_k}$ and $\mathbf{W}^{(v)} \in \mathbb{R}^{D \times D_v}$

Output: Attention($\mathbf{Q}, \mathbf{K}, \mathbf{V}$) $\in \mathbb{R}^{N \times D_v} : \{\mathbf{y}_1, \dots, \mathbf{y}_N\}$

```

 $\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)}$  // compute queries  $\mathbf{Q} \in \mathbb{R}^{N \times D_k}$ 
 $\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)}$  // compute keys  $\mathbf{K} \in \mathbb{R}^{N \times D_k}$ 
 $\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)}$  // compute values  $\mathbf{V} \in \mathbb{R}^{N \times D}$ 
return Attention( $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ ) = Softmax  $\left[ \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D_k}} \right] \mathbf{V}$ 

```

Todo esto que vimos de atención lo hicimos para un solo patrón (por ejemplo tiempo verbal, etc).

Multi-head attention

Podrían existir múltiples patrones de atención que sean útiles al mismo tiempo. En NLP, uno podría estar asociado al tiempo verbal y otro a la semántica del vocabulario. Solución: colocar múltiples cabezas (nosotros) de atención en paralelo, con parámetros independientes (que es lo que hacen los transformers).

$$\begin{array}{l}
 \mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)} \\
 \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(k)} \\
 \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)}
 \end{array}
 \quad
 \begin{array}{l}
 \mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h) \\
 \mathbf{H}_h \in \mathbb{R}^{N \times D_v}
 \end{array}$$

Estas cabezas (las H_i que representa la cabeza i) se concatenan en una única matriz. Y luego el resultado es linealmente transformado usando una nueva matriz de pesos $\mathbf{W}^{(o)}$ que también la aprendemos, dependiendo de qué patrones me importan más.

$$\mathbf{Y}(\mathbf{X}) = \text{Concat} [\mathbf{H}_1, \dots, \mathbf{H}_H] \mathbf{W}^{(o)}$$

$$N \times D_v \quad N \times D_v$$

$$N \times D_v \quad N \times HD_v \quad HD_v \times D_v$$

Usualmente, $D_v = D/H$ para que la salida sea de dimensión $N \times D$ (recordar que D era la dimensión del embedding de entrada)

Algoritmo de lo que vimos:

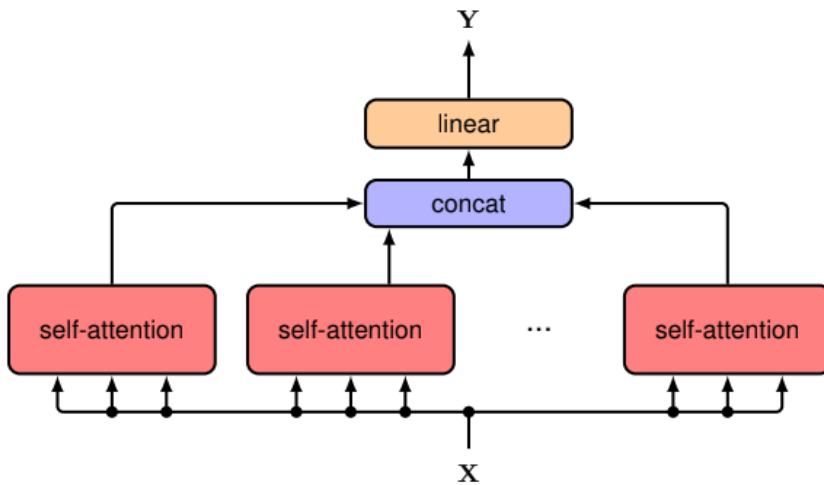
Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$
 Query weight matrices $\{\mathbf{W}_1^{(q)}, \dots, \mathbf{W}_H^{(q)}\} \in \mathbb{R}^{D \times D}$
 Key weight matrices $\{\mathbf{W}_1^{(k)}, \dots, \mathbf{W}_H^{(k)}\} \in \mathbb{R}^{D \times D}$
 Value weight matrices $\{\mathbf{W}_1^{(v)}, \dots, \mathbf{W}_H^{(v)}\} \in \mathbb{R}^{D \times D_v}$
 Output weight matrix $\mathbf{W}^{(o)} \in \mathbb{R}^{HD_v \times D}$

Output: $\mathbf{Y} \in \mathbb{R}^{N \times D} : \{\mathbf{y}_1, \dots, \mathbf{y}_N\}$

```

// compute self-attention for each head
for  $h = 1, \dots, H$  do
     $\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)}$ ,  $\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(k)}$ ,  $\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)}$ 
     $\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$  //  $\mathbf{H}_h \in \mathbb{R}^{N \times D_v}$ 
end for
 $\mathbf{H} = \text{Concat} [\mathbf{H}_1, \dots, \mathbf{H}_H]$  // concatenate heads
return  $\mathbf{Y}(\mathbf{X}) = \mathbf{H}\mathbf{W}^{(o)}$ 

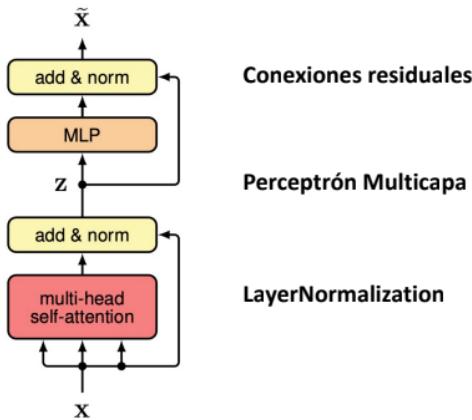
```



Si usamos el mismo token de entrada para determinar los coeficientes de atención, estas cabezas son las self-attention.

Transformer Layer

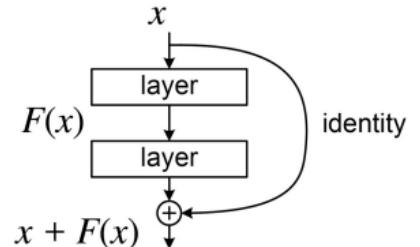
Incorporamos tres elementos adicionales:



Conexiones residuales

Ayudan a entrenar modelos más profundos.

La idea es procesar la X en una $F(X)$. Metemos un bypass entonces lo que sale de una capa residual es $X + F(X)$, predigo el residuo que me falta para transformar X en $F(X)$. Los short cuts ayudan a que el gradiente funcione mejor (mejoran la dinámica). Esta misma idea la aplico en transformers, $Y(X) + X$.



Layer Normalization

Similar a BatchNormalization, pero normalizando a nivel de features en lugar de a nivel de batch sample, es decir, tengo una media y un desvío por cada feature.

En BatchNorm, si el batch es muy pequeño, las estimaciones de media y desvío son muy ruidosas.

La misma función de normalización se puede usar en training y test, dado que se computa sobre las features para cada sample independientemente.

Conexiones Residuales + Layer Normalization

$$\mathbf{Z} = \text{LayerNorm}[\mathbf{Y}(\mathbf{X}) + \mathbf{X}]$$

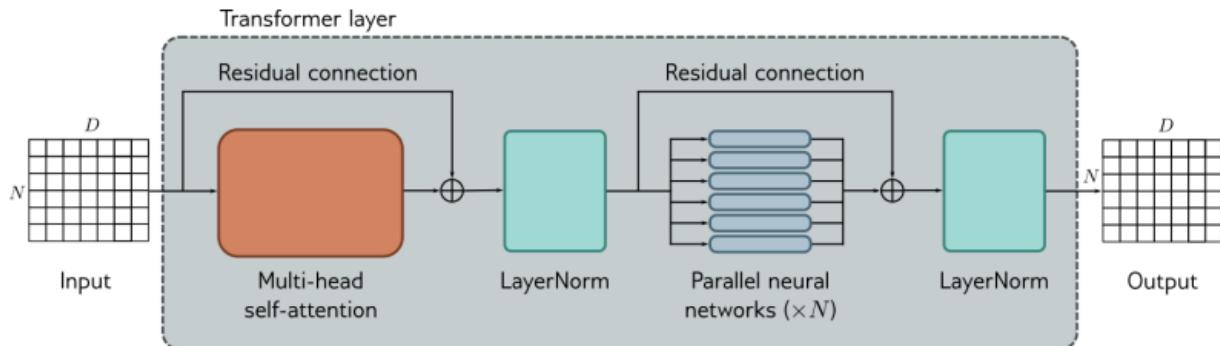
MLP Compartido

Observando la composición de la capa de atención, de no ser por la función Softmax (aparece en la cuenta de las H_i), el resto es lineal. Solución: Usar un MLP no lineal, con D inputs y D outputs. Por lo tanto usamos un MLP no lineal procesando los embeddings que pasan por las capas del transformer.

Conexiones Residuales + Layer Normalization + Shared MLP

$$\tilde{\mathbf{X}} = \text{LayerNorm} [\text{MLP} [\mathbf{Z}] + \mathbf{Z}]$$

El perceptrón es compartido por todos los tokens para permitir procesar secuencias de tamaño arbitrario.



Positional encoding

Los transformers son equivariantes respecto a permutaciones en la entrada. La atención que le presta es la misma, no me importa el orden de los tokens.

¿Cómo tomar en cuenta el orden para definir el significado? Solución: Positional encoding.

Los positional embedding codifican la posición del token en el mismo dato de entrada. Estos son vectores del mismo tamaño que el embedding original, por lo tanto se suman al token original. Los r_n tienen un valor diferente para cada posición.

$$\tilde{\mathbf{x}}_n = \mathbf{x}_n + \mathbf{r}_n$$

Token original Positional embedding

Tres ideas de qué es r_n :

- Idea 1: asociar un entero 1, 2, 3, 4, ... a cada posición. El problema es que la magnitud se incrementa sin restricción de valor (me quedan valores muy chicos y valores muy grandes)
- Idea 2: asociar un valor entre (0,1). El problema es que si fueran secuencias de diferente longitud, los valores cambiarían.
- Idea 3: positional encodings basados en sinusoidales

$$r_{ni} = \begin{cases} \sin\left(\frac{n}{L^{i/D}}\right), & \text{if } i \text{ is even,} \\ \cos\left(\frac{n}{L^{(i-1)/D}}\right), & \text{if } i \text{ is odd.} \end{cases}$$

Transformer Models

En qué contextos se usan estas Transformer layers.

Los tokens son procesados por sucesivas capas de transformers.

Encoder: Transforma embeddings en nuevas representaciones que pueden ser usadas para downstream tasks como clasificación. Por ejemplo, tengo un tweet, que el transformer lo procesa y hago una predicción de si es hate speech o no.

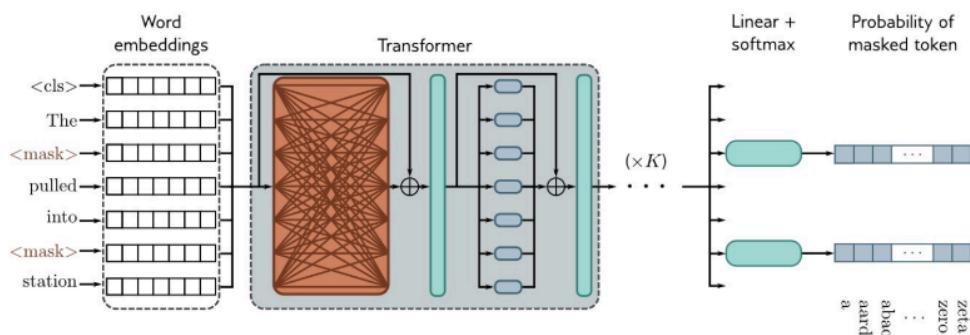
Decoder: Predice el próximo token para continuar el texto de entrada, es decir, quiere generar texto. Por ejemplo, el autocompletador de texto en WhatsApp.

Encoder-decoder: Son usados en tareas de secuencia a secuencia, donde una cadena es convertida en otra. Por ejemplo, al traducir texto.

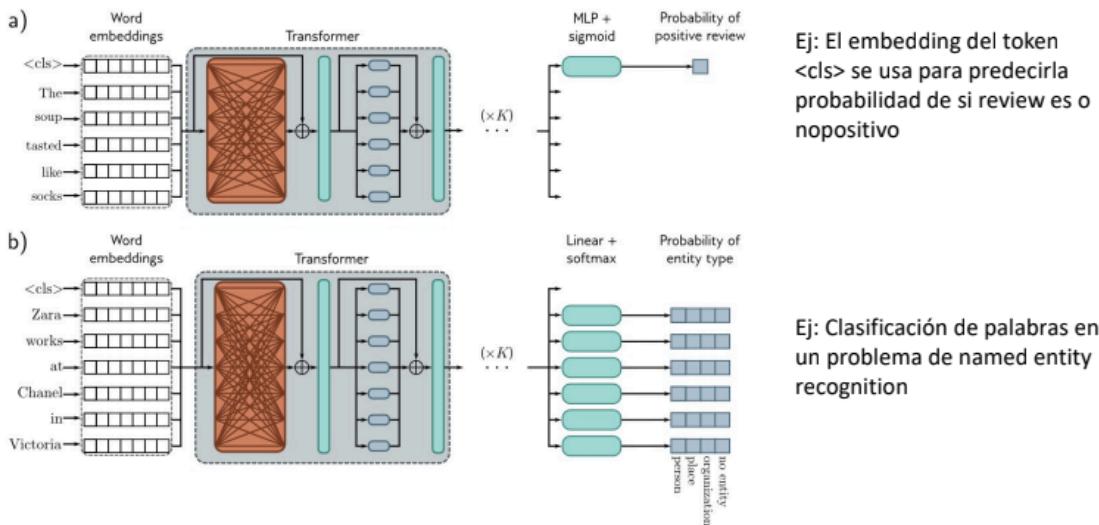
BERT (encoder)

Tiene dos etapas, la primera es de preentrenamiento. Defino una tarea de pretexto que consiste en predecir palabras faltantes. Las salidas son las probabilidades por token enmascarado (un vector de tamaño diccionario de tokens con la probabilidad de que el token faltante sea ese). La segunda capa es una etapa de fine tuning donde la entreno para tareas downstream.

La primera etapa:

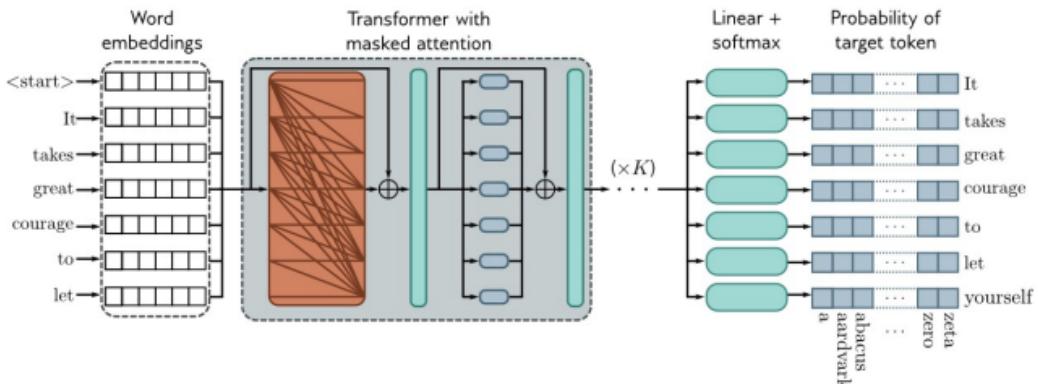


La segunda etapa (fine tuning):



GPT 3 (decoder)

La arquitectura es muy similar al modelo Encoder (una serie de bloques Transformer operando sobre embeddings a nivel token) Pero el objetivo es diferente: predecir el próximo token en una secuencia. Las salidas son las probabilidades por cada token (un vector de tamaño diccionario de tokens con la probabilidad de que el token siguiente sea ese).



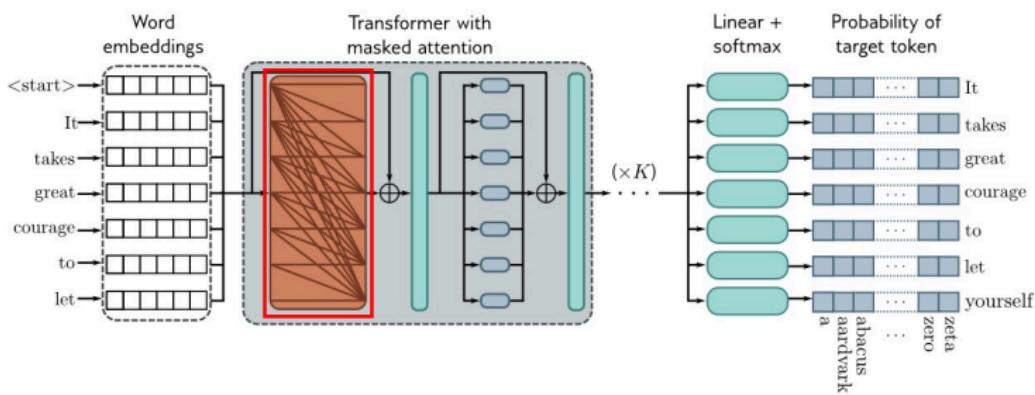
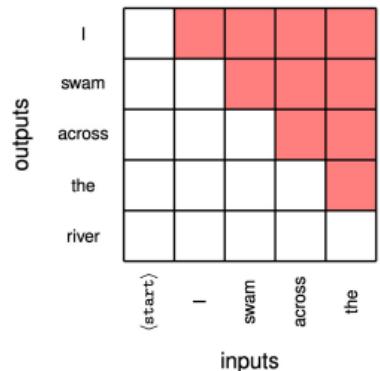
→ acá ya usa masked self-attention

Masked Self-Attention

Al entrenar un modelo decoder, la loss se modela para maximizar la probabilidad de la próxima palabra en un esquema autoregresivo. Si pasamos la frase entera, el modelo tiene acceso a toda la secuencia y no queremos eso (no aporta nada, no aprende), porque queremos *predecir* la siguiente palabra.

Solución: masked-self attention → setear a 0 los coeficientes de atención de las palabras futuras (así no veo lo que viene).

En la figura en rojo se muestran los coeficientes en 0. Por ejemplo, al predecir "Across", el modelo solo puede atender a (<start>, I, swam)



→ en lo rojo se pone la masked self-attention

Few Shot Learning

En few-shot learning, se proveen una serie de ejemplos de contexto y luego el modelo completa con lo más probable habiendo “aprendido” de los ejemplos anteriores.

Ejemplo:

Poor English input: I eated the purple berries.

Good English output: I ate the purple berries.

Poor English input: Thank you for picking me as your designer. I'd appreciate it.

Good English output: Thank you for choosing me as your designer. I appreciate it.

Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.

Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.

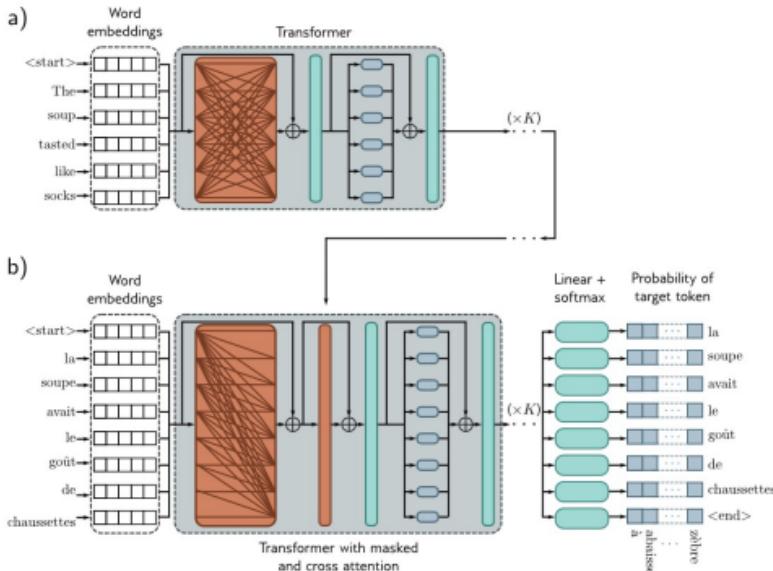
Poor English input: I'd be more than happy to work with you in another project.

Good English output: I'd be more than happy to work with you on another project.

(result from Brown et al., 2020)

Le doy ejemplos de poor english y good english y con eso espero que aprenda a predecir si pertenece a uno o al otro. Esto no es fine tuneo porque no cambio el valor de los parámetros.

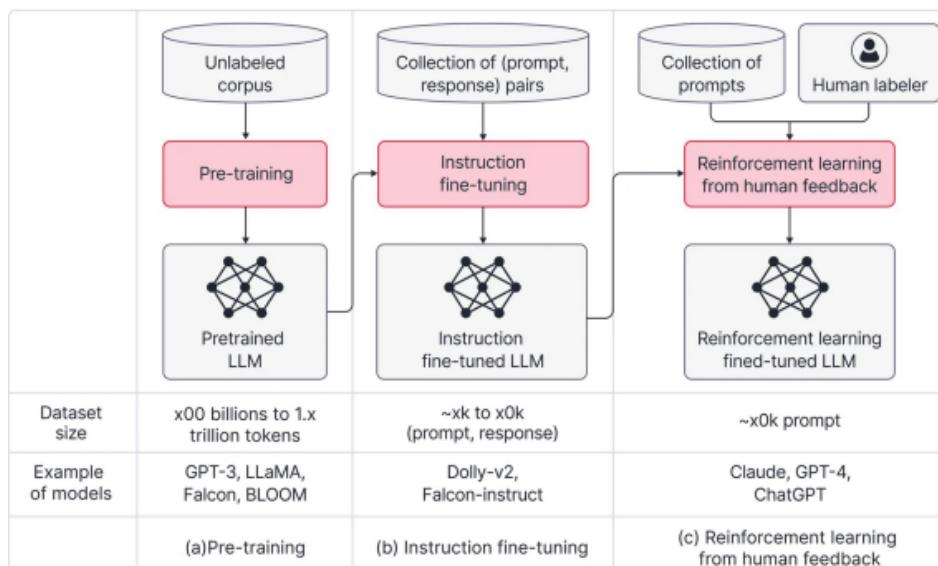
Modelos Encoder-Decoder



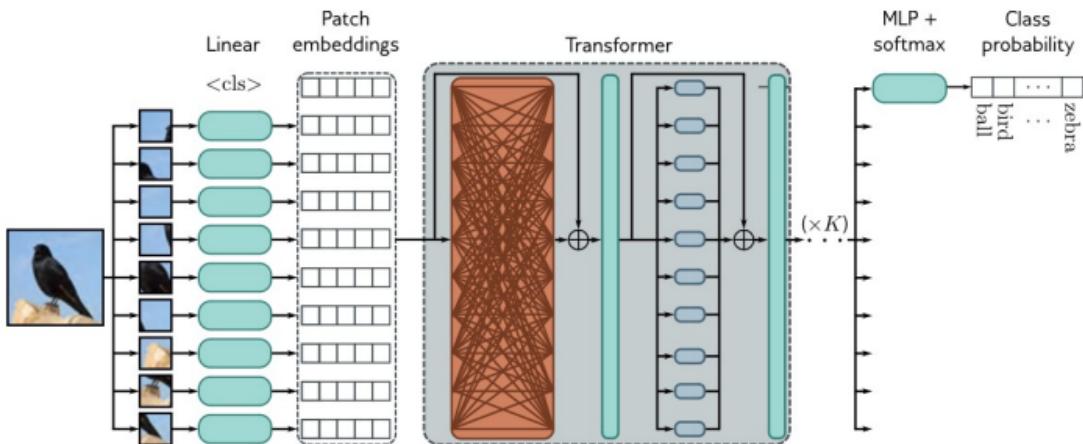
Quiero traducir de inglés a francés. La 1era oración (en inglés) se pasa por el encoder, generando embeddings para cada token. La 2da oración se pasa por el decoder que usa masked self attention, pero también presta atención a los embeddings de salida del encoder usando cross-attention (rectángulo naranja). La loss se computa como en el encoder, maximizando la probabilidad de cada palabra traducida.

Entrenando LLMs (Large Language Models)

Tienen 3 pasos: pre training + instruction fine tuning + RLHF (feedback con humano)



Vision Transformers



Agarro una imagen, la parcheo, genero embeddings y después uso la arquitectura normal de los transformers. El modelo es entrando con aprendizaje supervisado.

Modelos multimodales

Puedo mapear casi cualquier cosa (imagen, texto) en el espacio de embeddings. La idea es poder hacer un embedding compuesto.