Deep Learning

Eduardo Morales y Hugo Jair Escalante

INAOE

Contenido

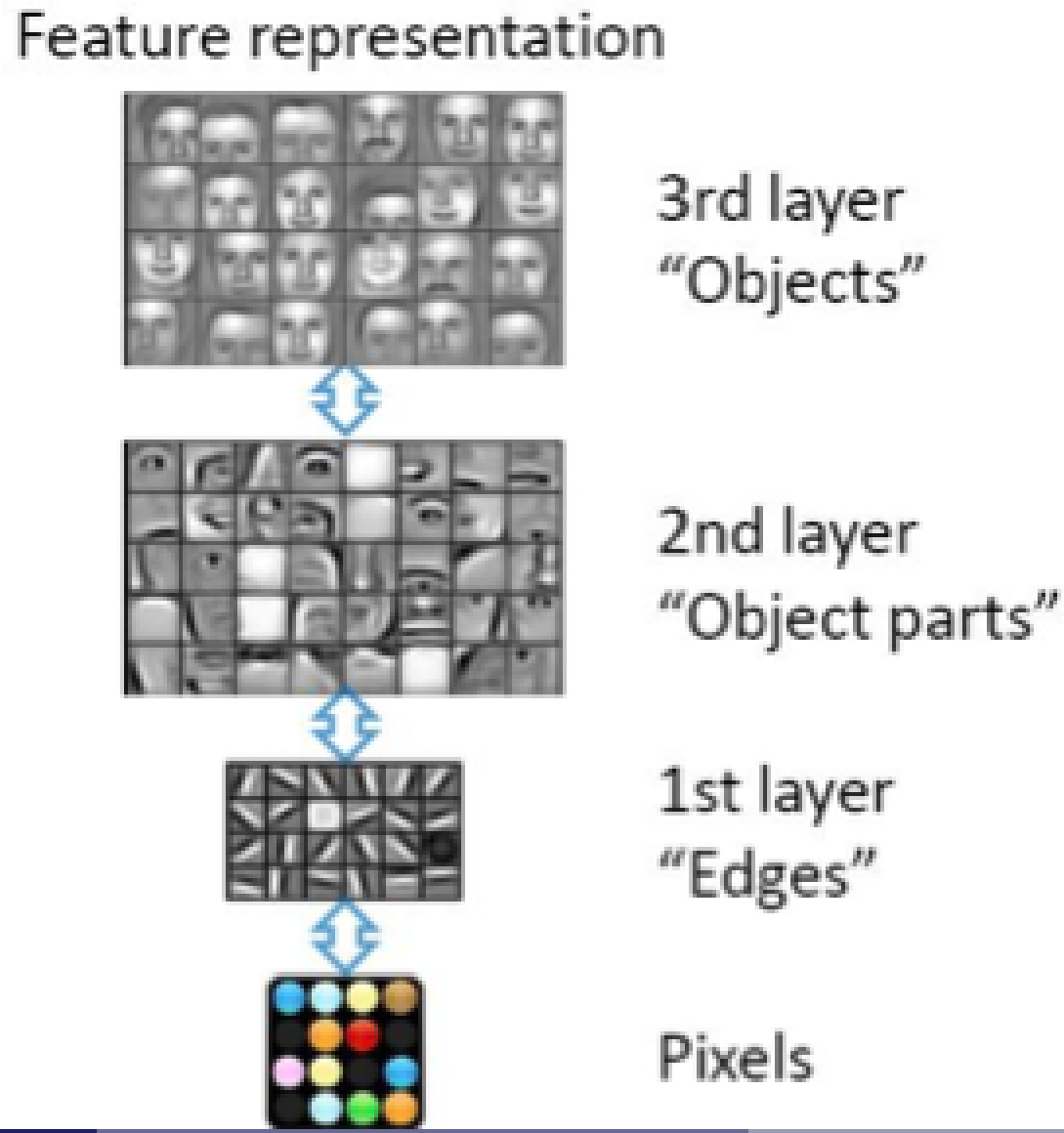
Deep Learning

- El poder tener una computadora que modele el mundo lo suficientemente bien como para exhibir inteligencia ha sido el foco de investigación en IA desde hace más de medio siglo
- La dificultad en poder representar toda la información acerca del mundo de forma utilizable por una computadora, ha orillado a los investigadores a recurrir a algoritmos de aprendizaje para capturar mucha de esta información
- Durante mucho tiempo han existido dominios que se han resistido a ser resueltos de manera general como entender imágenes o lenguaje

Deep Learning

- Una idea que ha rondado a los investigadores desde hace tiempo es la de descomponer los problemas en sub-problemas a diferentes niveles de abstracción
- Por ejemplo, en visión podemos pensar en extraer pequeñas variaciones geométricas, como detectores de bordes, a partir de los pixeles, de los bordes podemos pasar a formas locales, de ahí a objetos, podemos pensar en varias capas intermedias

Deep Learning



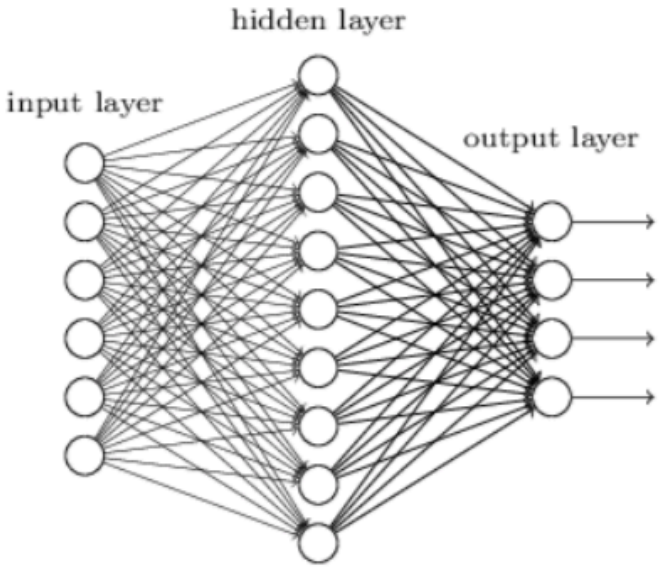
Deep Learning

- El foco de aprendizaje profundo o *Deep Learning* es el de descubrir automáticamente estas abstracciones entre los atributos de bajo nivel y los conceptos de alto nivel
- La profundidad de la arquitectura se refiere al número de niveles

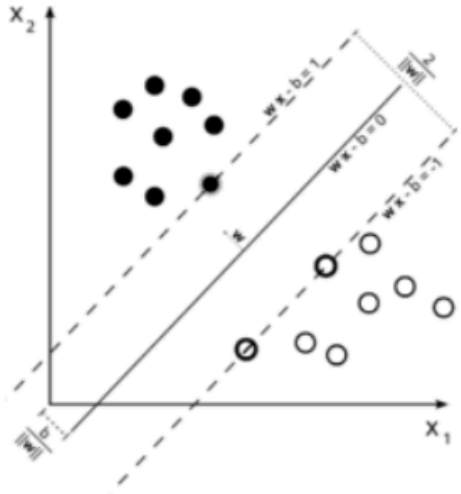
Deep Learning

- La mayoría de los sistemas de aprendizaje tienen arquitecturas poco profundas (superficiales)
- Durante decadas los investigadores de redes neuronales quisieron entrenar redes multi-capas profundas con poco éxito
- El verdadero cambio vino hasta 2006 con un artículo de Hinton y sus colaboradores de la U. de Toronto.
- Poco después se desarrollaron muchos otros esquemas con la misma idea general: guiar el aprendizaje por niveles usando aprendizaje no supervisado en cada nivel

Shallow vs. Deep

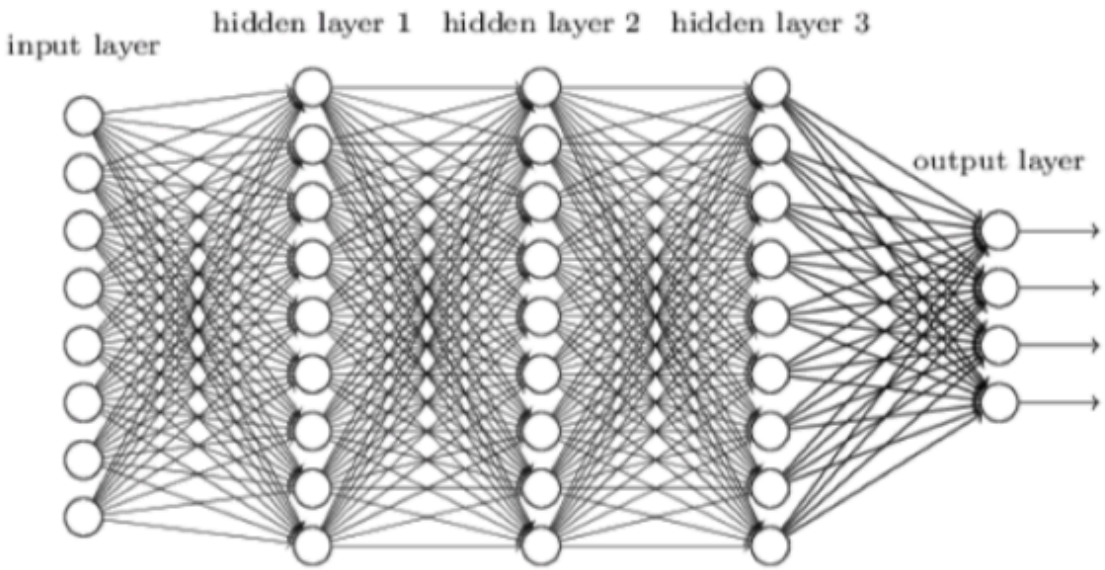


$$f(\mathrm{x}) = w\phi(x) + b$$



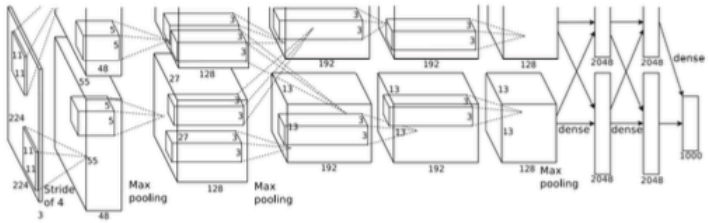
$$f(\mathrm{x}) = \sum_{i}^{N} \alpha_i y_i k(\mathrm{x}_i, \mathrm{x}) + b$$

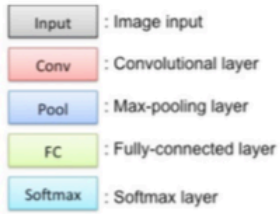
Shallow vs. Deep



$$f(\mathrm{x}) = W_3\phi_3(W_2\phi_2(W_1\phi_1(X) + b) + b_2) + b_3$$

Shallow vs. Deep





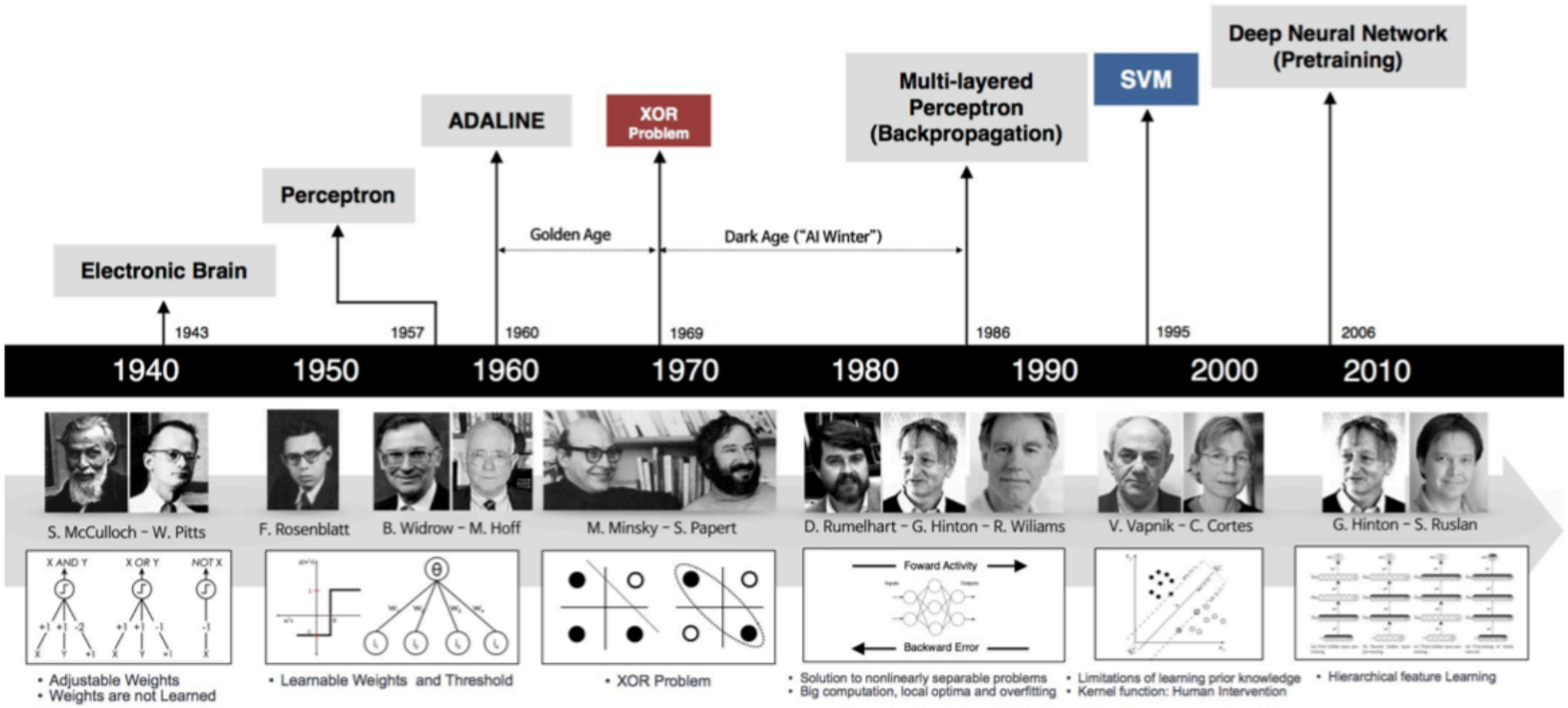
Deep Learning

- Uno de los argumentos principales es que existen ciertas funciones que no pueden ser representadas de manera eficiente por arquitecturas poco profundas
- El éxito de los SVM y *Kernel Machines* radica en cambiar la representación de entrada a una de mayor dimensionalidad que permita construir mejores clasificadores (pero se quedan en un solo nivel)
- Investigaciones en anatomía del cerebro sugieren que el sistema visual y el generador de voz tienen varias capas de neuronas (entre 5 y 10)

Deep Learning

- Se puede entrenar usando algunas de las múltiples variantes de *backpropagation* (gradiente conjugado, *steepest descent*, etc.)
- El problema de *backpropagation* con muchas capas es que al propagar los errores a las primeras capas se vuelven muy pequeños y por lo tanto poco efectivos
- Se han desarrollado varios esquemas para evitarlo

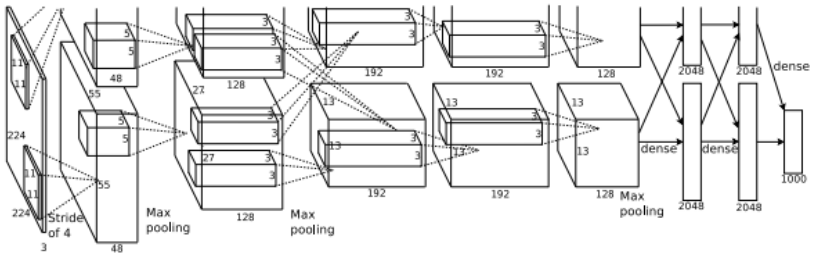
Historia



http://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

Algunos Resultados

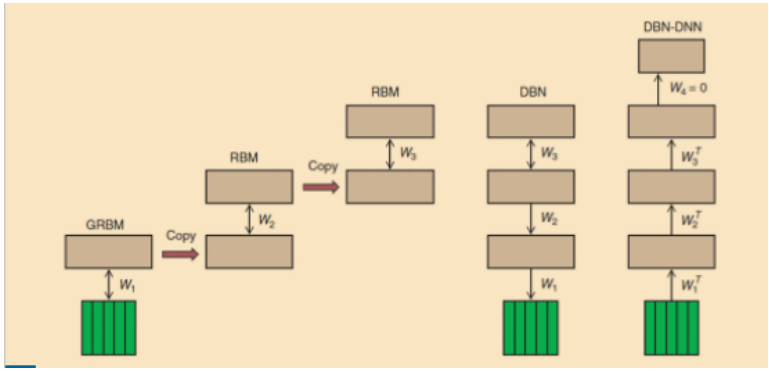
- Uno de los principales resultados fue en clasificación de imágenes
- En 2012 Krizhevsky et al. (*Imagent classification with deep convolutional neural networks*) entrenaron una red para clasificar 1,000 conceptos de ImageNet usando alrededor de 1.2 millones de imágenes
- Trucos: ReLU, múltiples GPUs, data augmentation, dropout, ..



Model	Top-1	Top-5
Sparse coding [2]	*47.1%*	*28.2%*
SIFT + FVs [24]	*45.7%*	*25.7%*
CNN	**37.5%**	**17.0%**

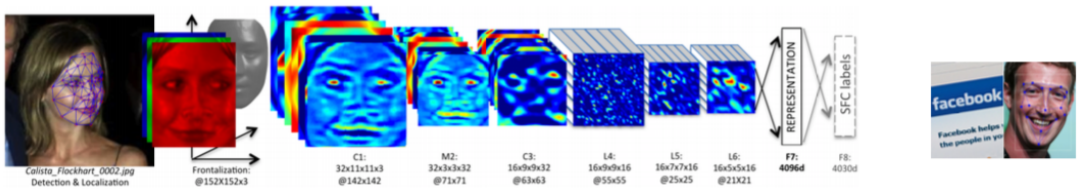
Algunos Resultados

- Hinton et al. 2012 (*Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups*) publicaron un artículo con resultados mucho mejores que el estado del estado
- Las principales compañías de comunicaciones cambiaron al uso de Restricted Boltzman Machines para reconocimiento de voz
- Ideas: Pre-entrenar RBMs + fine tuning + HMM

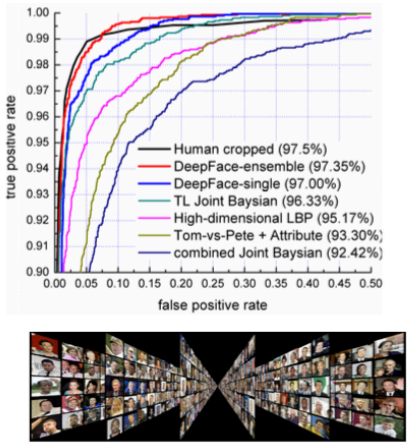


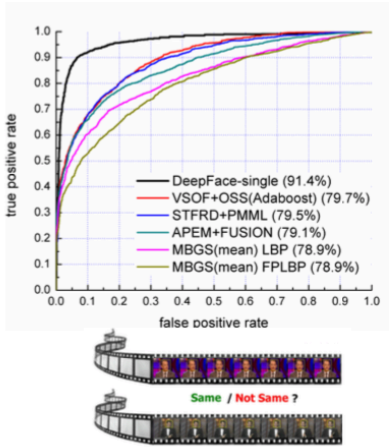
Algunos Resultados

- En 2014 se anunció una red neuronal entrenada con 4.44 millones de imágenes



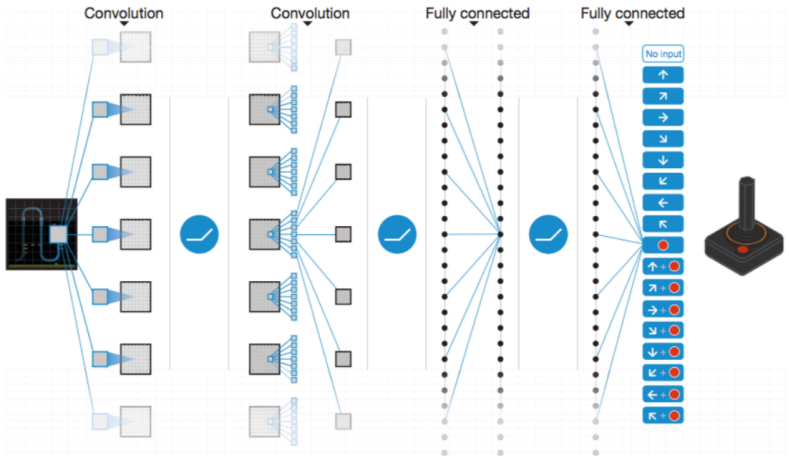
- Obtuvo 97.35% de accuracy vs. 97.5% de humanos y un 91.4% en la base de datos de caras de youtube

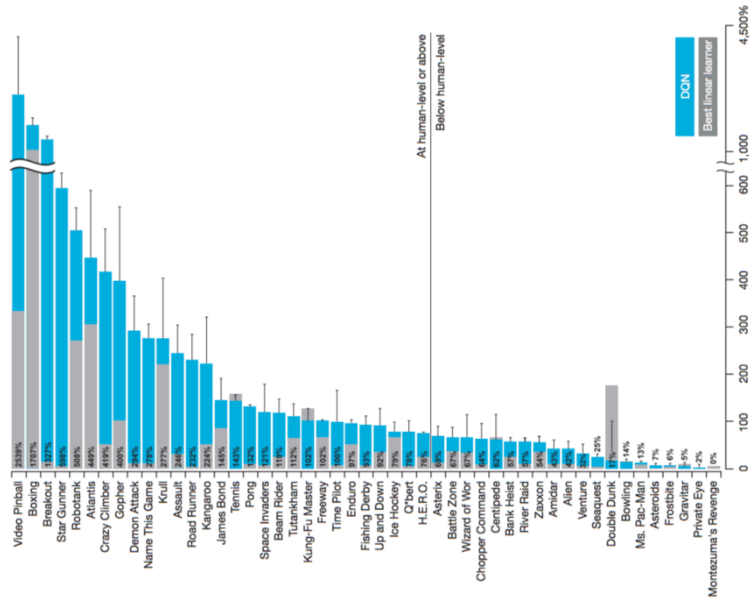




Algunos Resultados

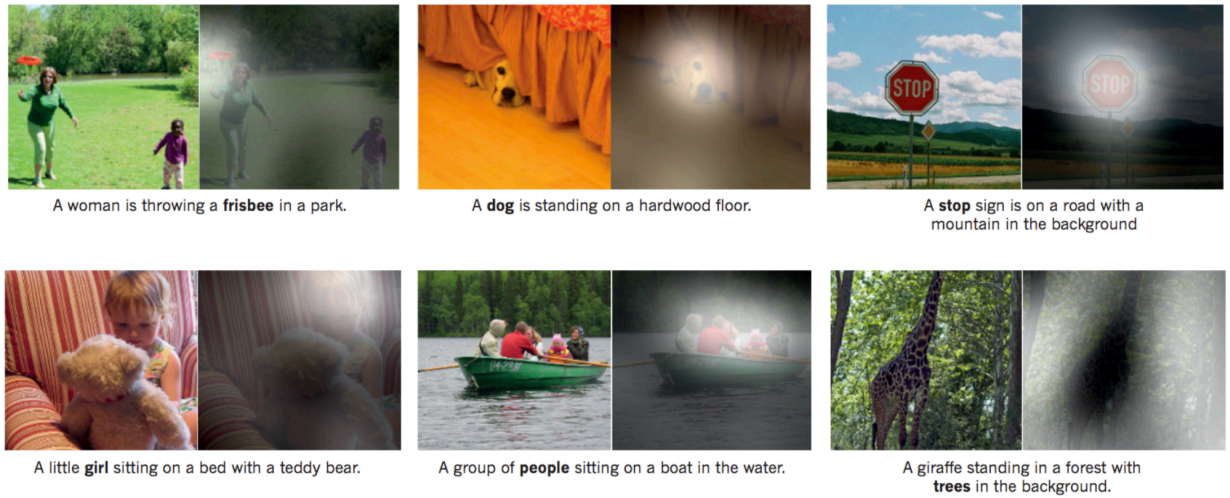
- En 2015, Deepmind (Mnih et al.) publicaron su algoritmo de aprendizaje por refuerzo profundo (DQN) para aprender a jugar los videojuegos de Atari
- Mejoró todos los algoritmos anteriores y fue superior en muchos de los juegos que los expertos humanos





Algunos Resultados

- Anotación automática de imágenes



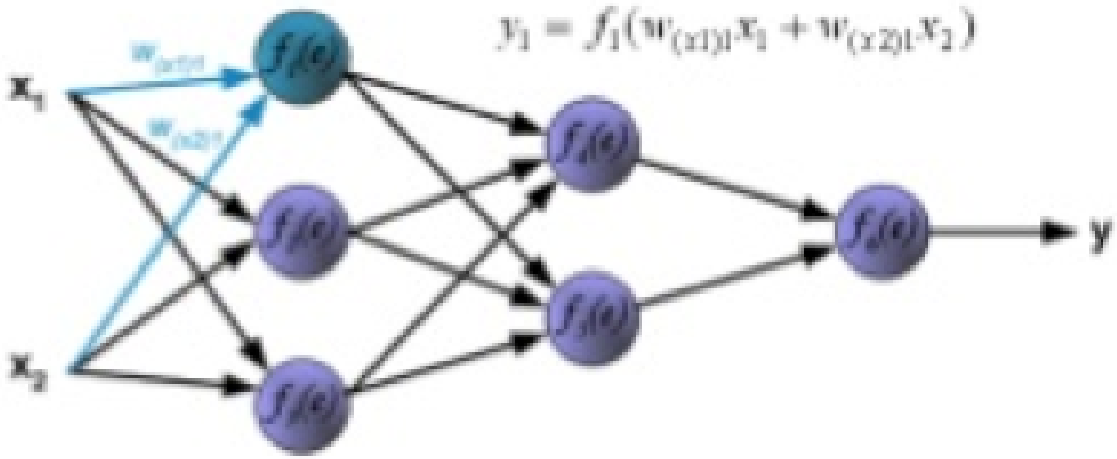
https://pdollar.wordpress.com/2015/01/21/image-captioning/

- AlphaGo Zero
- Generación automática de texto
- Reemplazo de cuerpos en videos
- ...

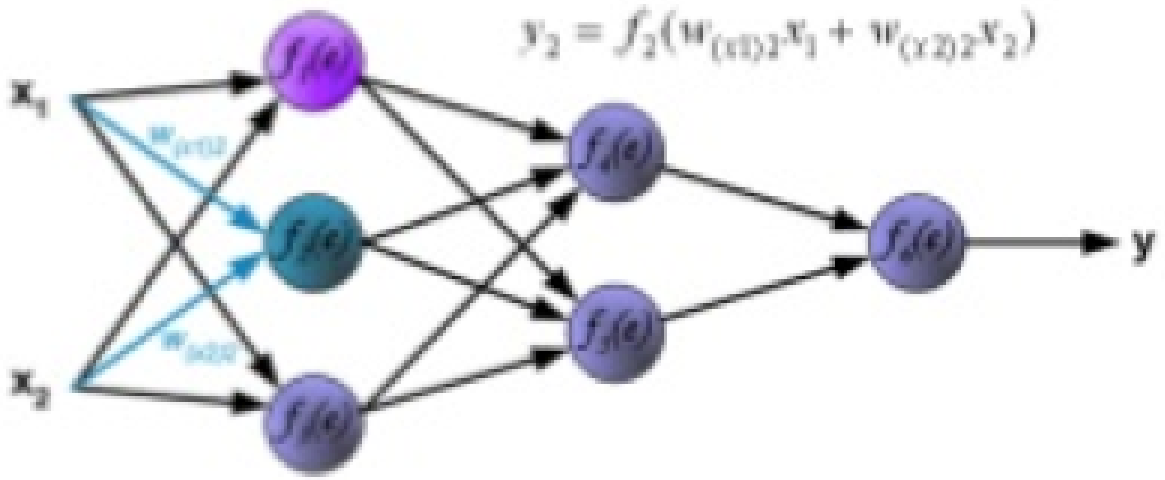
Deep Learning

- El corazón de la mayoría de las redes neuranales es el algoritmo de retropropagación (*backpropagation*)
- Vamos a hacer un repaso rápido

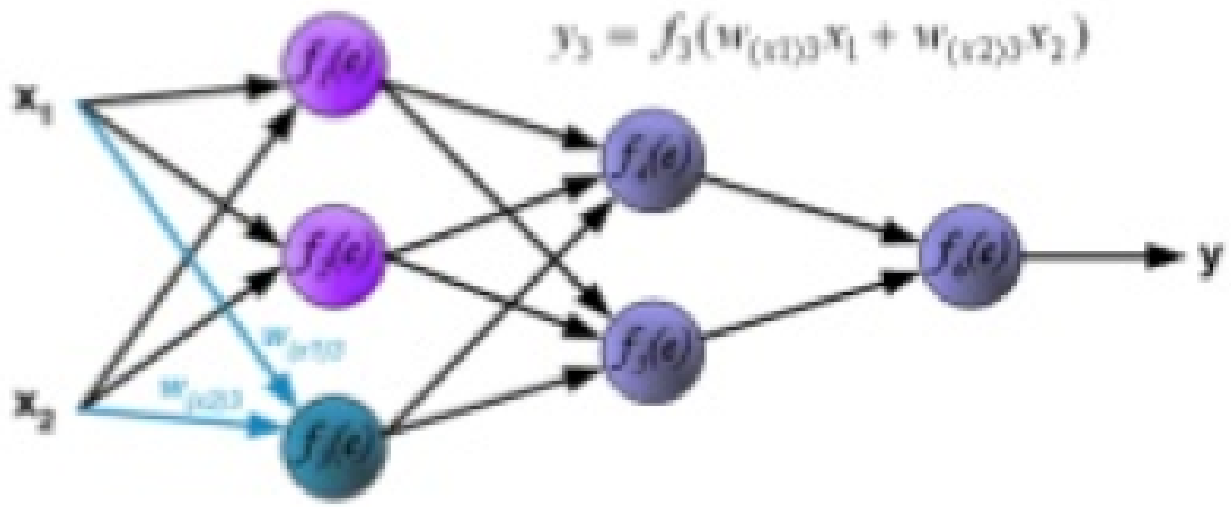
Repaso RN-Backpropagation



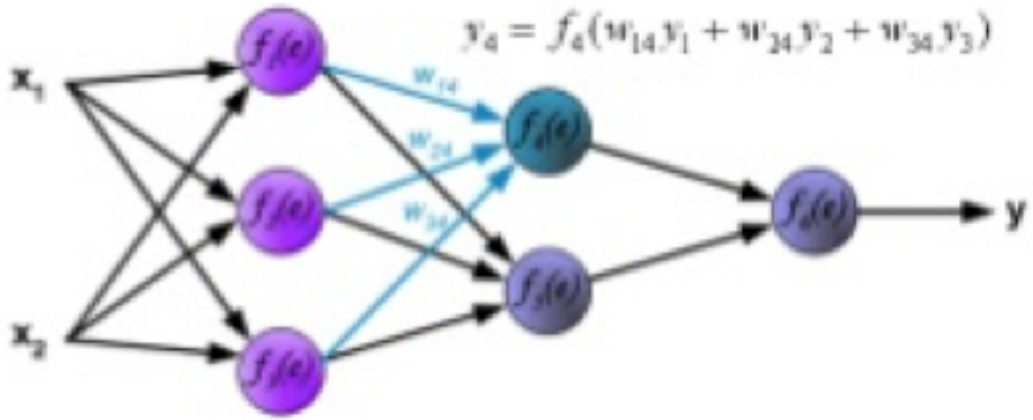
Repaso RN-Backpropagation



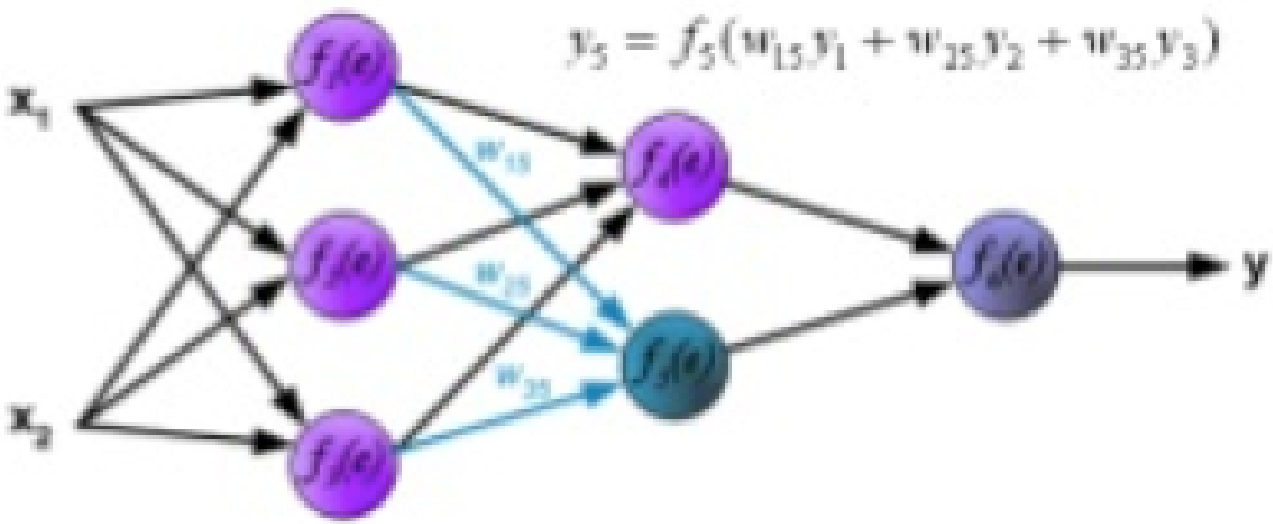
Repaso RN-Backpropagation



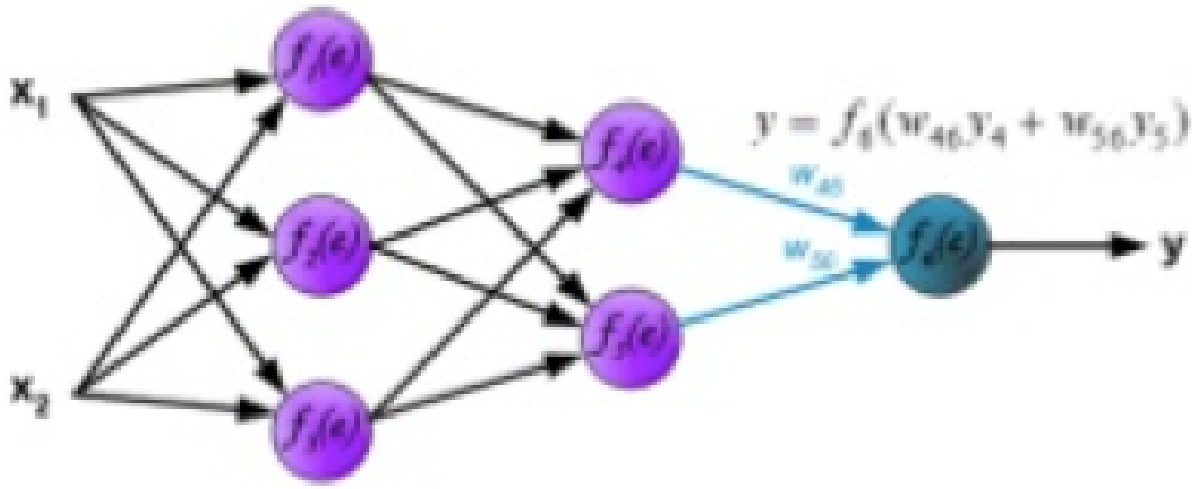
Repaso RN-Backpropagation



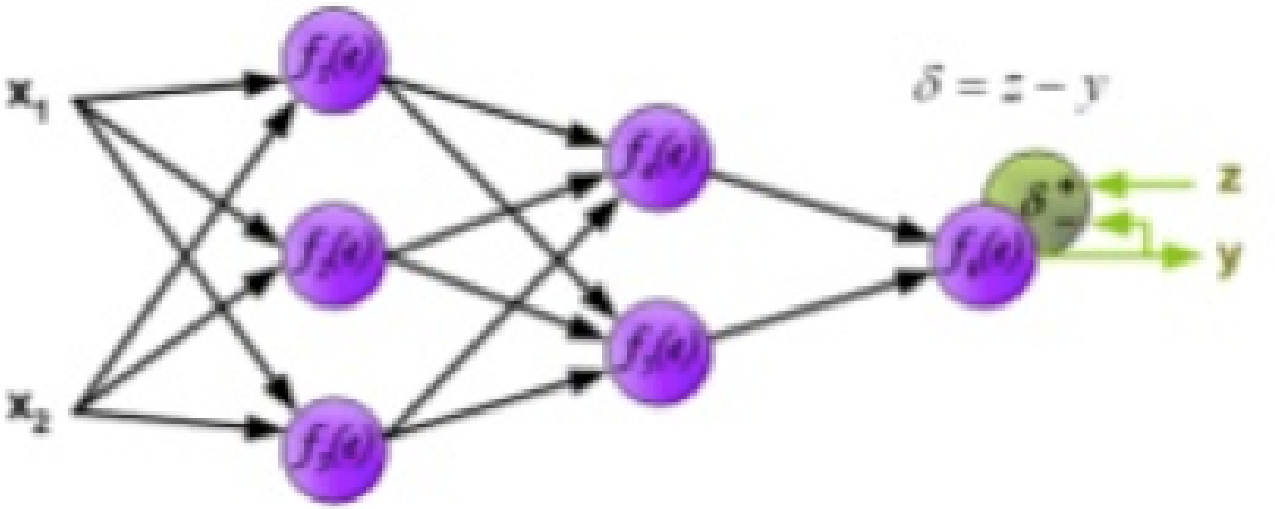
Repaso RN-Backpropagation



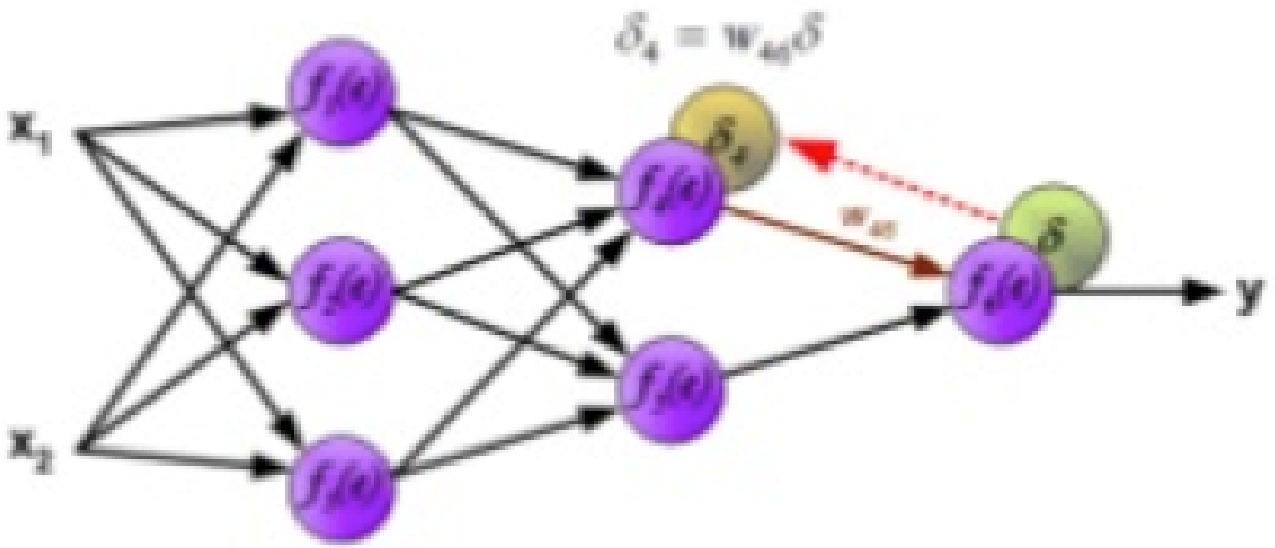
Repaso RN-Backpropagation



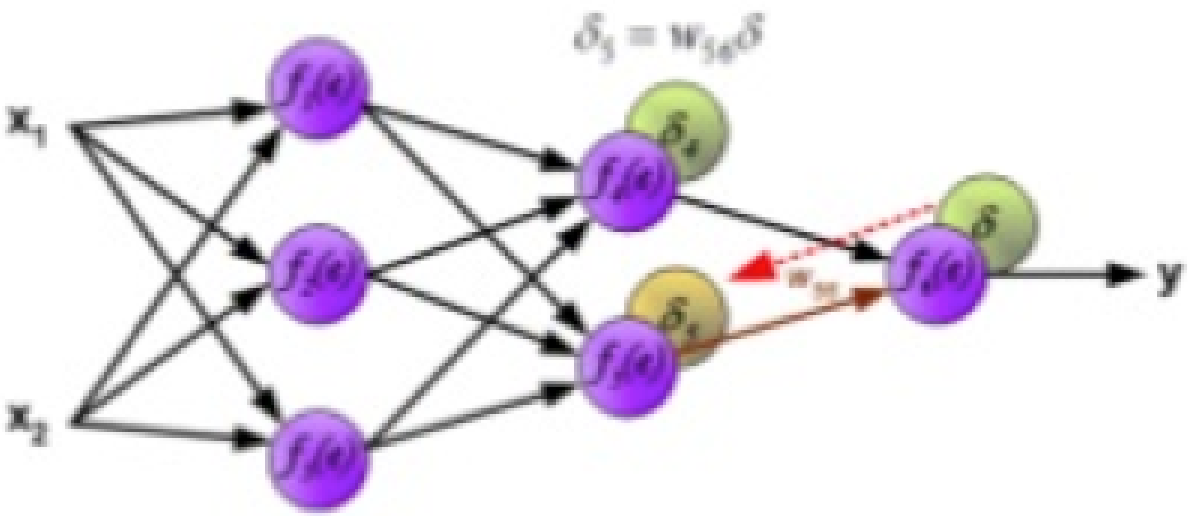
Repaso RN-Backpropagation



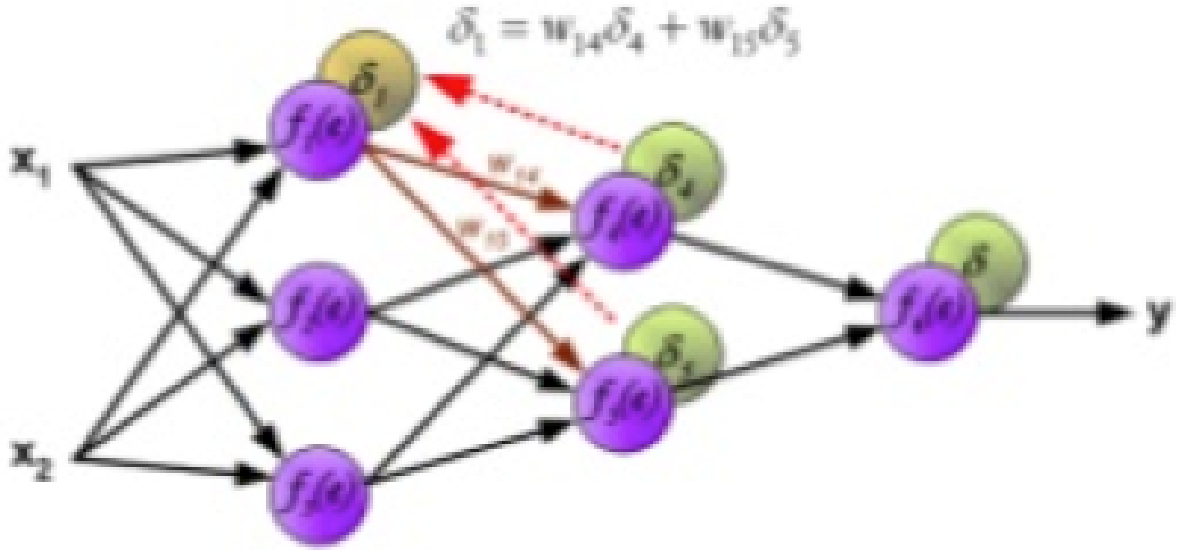
Repaso RN-Backpropagation



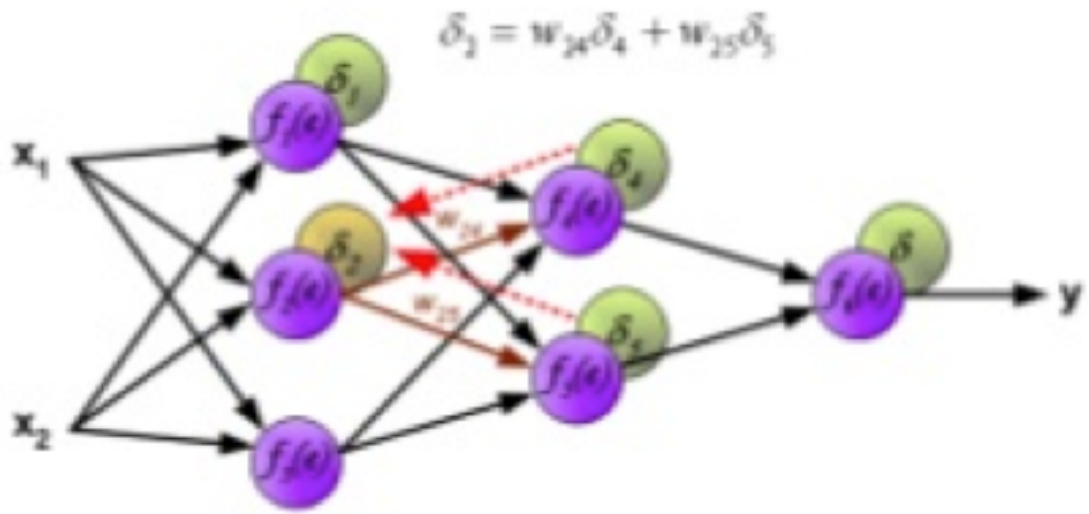
Repaso RN-Backpropagation



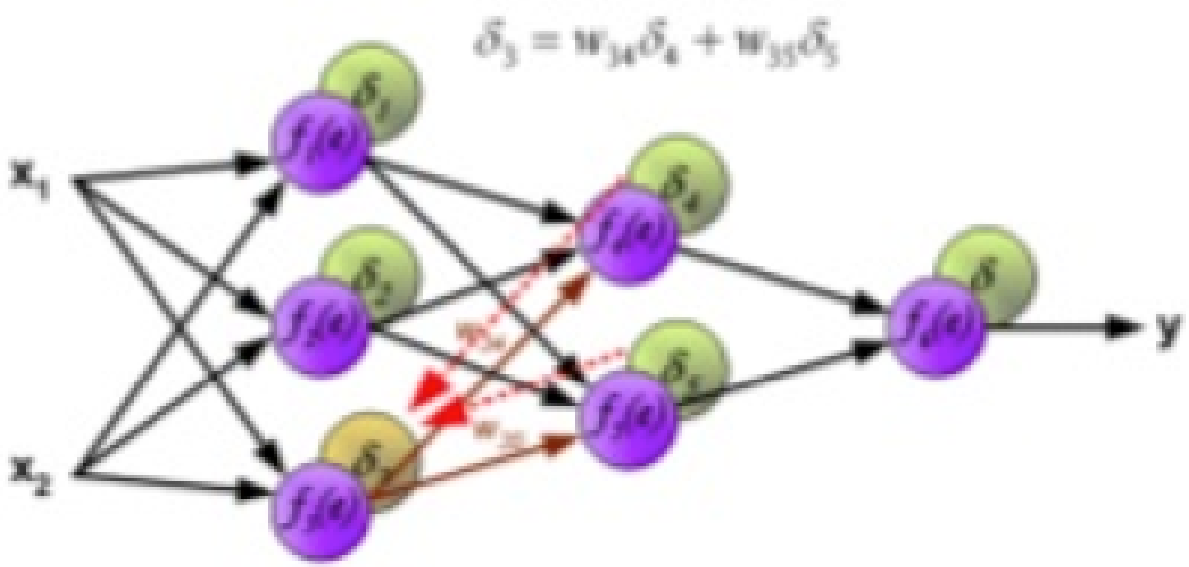
Repaso RN-Backpropagation



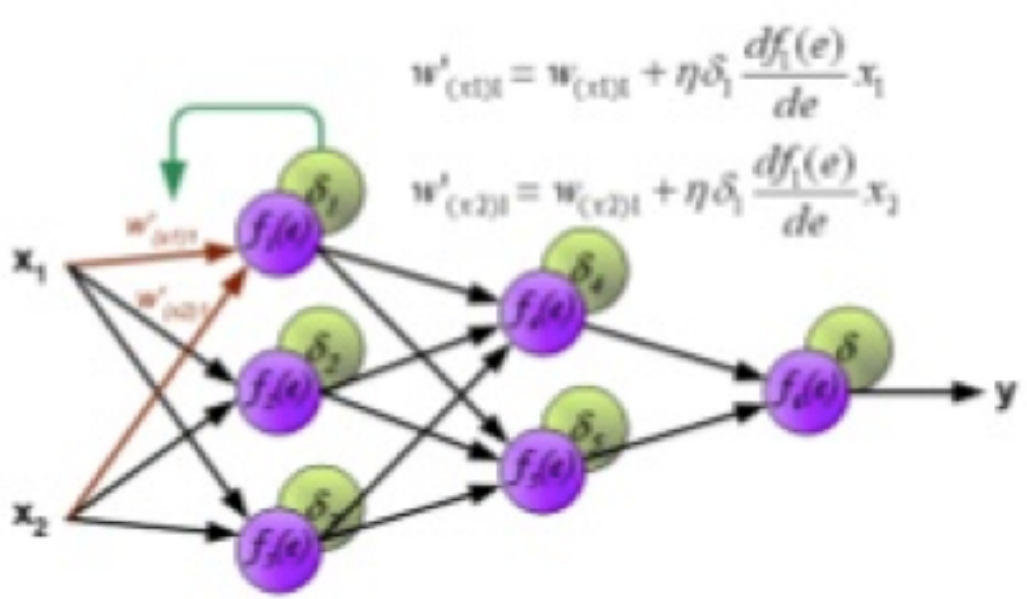
Repaso RN-Backpropagation



Repaso RN-Backpropagation



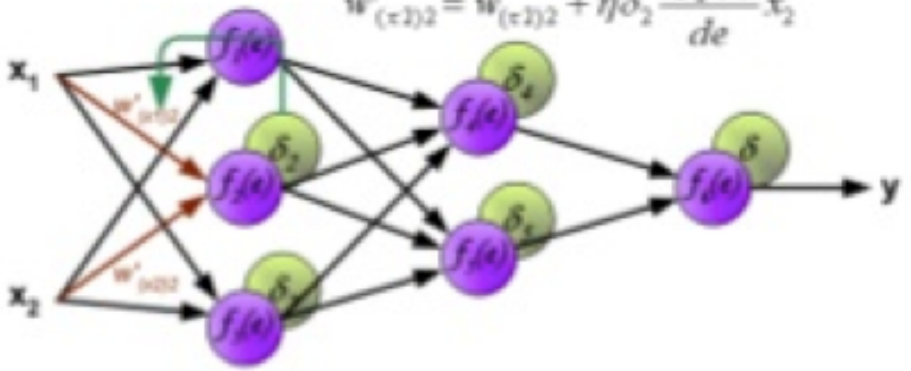
Repaso RN-Backpropagation



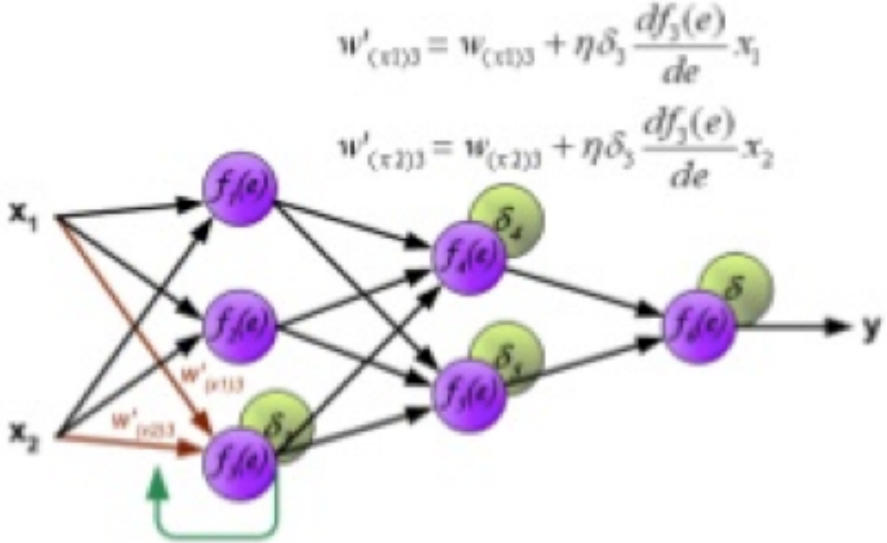
Repaso RN-Backpropagation

$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 \frac{df_2(e)}{de} x_2$$



Repaso RN-Backpropagation

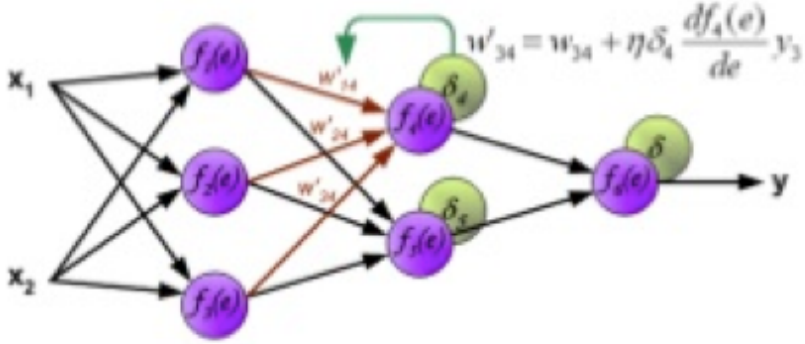


Repaso RN-Backpropagation

$$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$



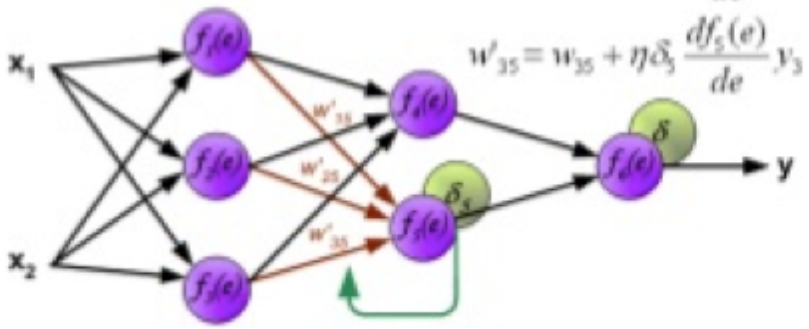
Repaso RN-Backpropagation

Contact Sheet

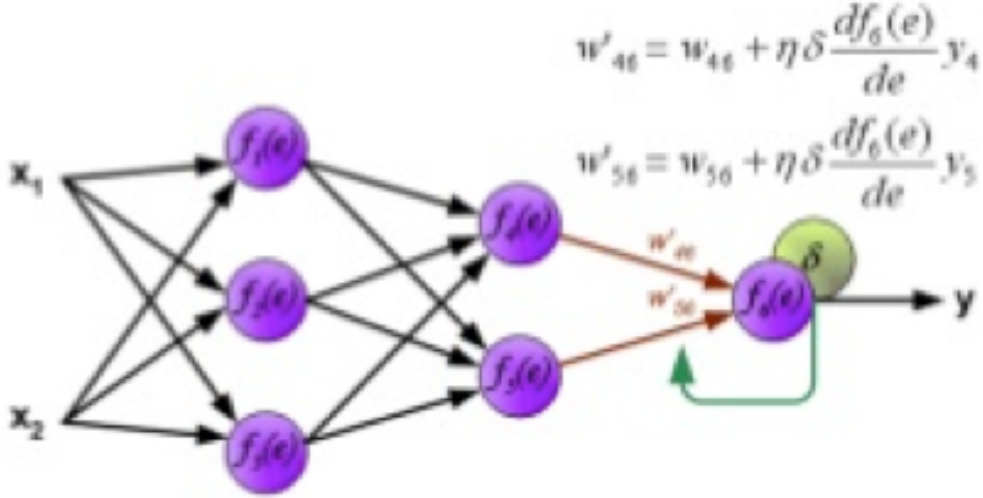
$$w'_{15} = w_{15} + \eta\delta_5 \frac{df_5(e)}{de} y_1$$

$$w'_{25} = w_{25} + \eta\delta_5 \frac{df_5(e)}{de} y_2$$

$$w'_{35} = w_{35} + \eta\delta_5 \frac{df_5(e)}{de} y_3$$



Repaso RN-Backpropagation



Backpropagation

- Cada unidad recibe señales de sus ligas de entradas y calcúla un nuevo nivel de activación que manda a través de sus ligas de salidas.
- La computación se hace en función de los valores recibidos y de los pesos.
- Se divide en dos:
  1. Un componente lineal, llamado la función de entrada (in_i), que calcula la suma de los valores de entrada.
  2. Un componente no lineal, llamado función de activación (g), que transforma la suma pesada en una valor final que sirve como su valor de activación (a_i).

Backpropagation

- La suma pesada es simplemente las entradas de activación por sus pesos correspondientes:

$$in_i = \sum_j w_{j,i} a_j = \mathbf{w_i} \cdot \mathbf{a_i}$$

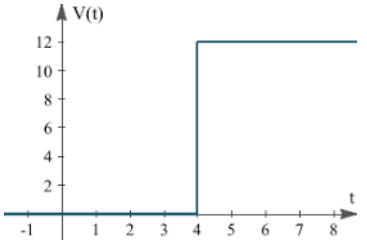
$\mathbf{w_i}$: vector de los pesos que llegan a la unidad i
$\mathbf{a_i}$: vector de los valores de activación de las entradas a la unidad i

- El nuevo valor de activación se realiza aplicando una función de activación g:

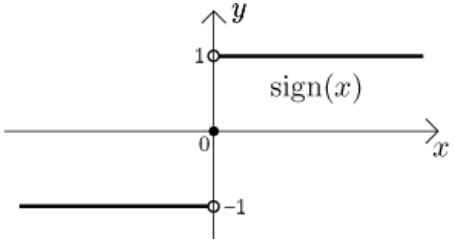
$$a_i \leftarrow g(in_i) = g(\sum_j w_{j,i} a_j)$$

Funciones de Activación

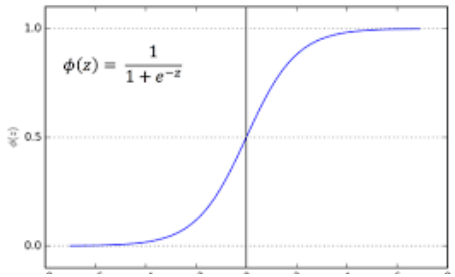
$$escalón_t(x) = \begin{cases} 1, & \text{si } x \geq t \\ 0, & \text{si } x < t \end{cases}$$



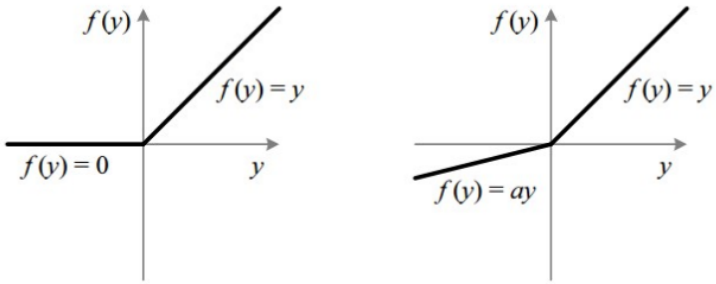
$$signo(x) = \begin{cases} +1, & \text{si } x \geq 0 \\ -1, & \text{si } x < 0 \end{cases}$$



$$sigmoide(x) = \frac{1}{1 + e^{-x}}$$

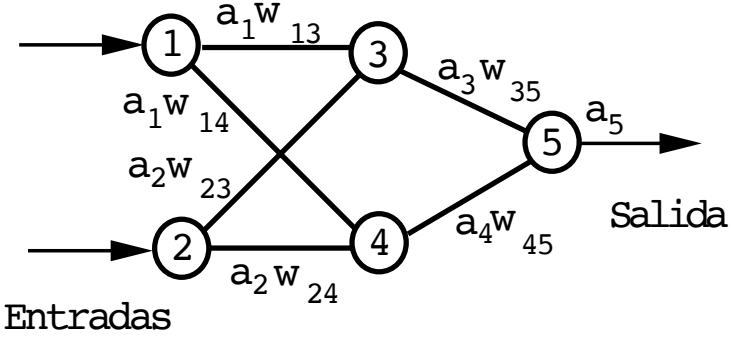


$$ReLU(x) = max(0, x)$$



Backpropagation

- Con una estructura fija y función de activación g fija, las funciones representables por una red *feed-forward* están restringidas por una estructura específica parametrizada.
- Por ejemplo:



$$a_5 = g(w_{3,5}a_3 + w_{4,5}a_4)$$

$$= g(w_{3,5}g(w_{1,3}a_1 + w_{2,3}a_2) + w_{4,5}g(w_{1,4}a_1 + w_{2,4}a_2))$$

Como g es una función no lineal, la red representa una función nolineal compleja.

Algoritmos de Red Neuronal

Función aprendizaje-red-neuronal (ejemplos)
red $\leftarrow$ una red con pesos asignados aleatoriamente
repeat
para cada $e \in$ *ejemplos* do
    $o \leftarrow$ salida de la red neuronal(red,e)
    $t \leftarrow$ valor observado de e
    Actualiza los pesos en la red con base en e, o y t
    end
until todos los ejemplos son predichos correctamente o se alcance un criterio de paro
regresa red

Backpropagation

- El gradiente descendiente trata de encontrar los pesos que mejor se ajustan a los ejemplos
- El error lo podemos expresar por diferencias de error al cuadrado de la siguiente forma:

$$E(W) = \frac{1}{2}\sum_i (t_i - o_i)^2$$

- Lo que queremos es determinar el vector de pesos que minimice el error E. Esto se logra alterando los pesos en la dirección que produce el máximo descenso en la superficie del error.
- La dirección de cambio se obtiene mediante el gradiente. El gradiente nos especifica la dirección que produce el máximo incremento, por lo que el mayor descenso es el negativo de la dirección

Backpropagation

- La regla de actualización de pesos es entonces:

$$W \leftarrow W + \Delta W$$

$$\Delta W = -\alpha \nabla E$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ &= \sum_{d \in D} (t_d - o_d)(-x_{i,d}) \end{aligned}$$

Por lo que:

$$\Delta w_i = \alpha \sum_{d \in D} (t_d - o_d) x_{i,d}$$

Deep Learning

- En la práctica, se tiende a usar un gradiente descendiente incremental. Esto es, en lugar de procesar el error sobre todos los datos, se hace sobre uno solo.
- En este caso, la regla de actualización es:

$$\Delta w_i = \alpha(t - o)x_i$$

- La cual es también conocida como la regla delta, LMS (*least-mean-square*), Adeline ó Widrow–Hoff.
- Rosenblatt la propuso en 1960 y probó que usando esta regla, se convergía a los pesos correctos, mientras la función fuera linealmente separable.
- Aprender en una red multicapas es muy parecido a un perceptrón, el truco es cómo dividir la culpa del error entre los pesos contribuyentes.

Backpropagation - Notación

- x_{ij} = la i-ésima entrada al nodo j
- w_{ij} = el peso asociado a la i-ésima entrada del nodo j
- $net_j = \sum_i w_{ij} x_{ij}$ (suma pesada de entradas al nodo j)
- o_j = la salida del nodo j
- t_j = la salida esperada del nodo j
- σ = función sigmoide (o algún otra)
- sal = el conjunto de nodos de salida
- α = razón de aprendizaje.
- $sal(j)$ = conjunto de nodos cuyas entradas directas incluyen la salida del nodo j

Algortimo de Backpropagation

1. Propaga las entradas a través de la red y calcula la salida
2. Propaga el error hacia atrás:
   1. para cada unidad de salida k, calcula su error δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

   2. Para cada unidad oculta h, calcula su error δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in sal(h)} w_{hk}\delta_k$$

   3. Actualiza los pesos w_{ij}:

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \qquad \text{donde} \qquad \Delta w_{ij} = \alpha \delta_j x_{ij}$$

Desarrollo

- Lo que queremos calcular es la actualización de los pesos w_{ij} sumándole Δw_{ij}

$$\Delta w_{ij} = \alpha \frac{\partial E_d}{\partial w_{ij}}$$

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ij} = \delta_j x_{ij}$$

Desarrollo - Capa de salida

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in sal}(t_k - o_k)^2$$

La derivada es cero en todos los casos, excepto cuando $k = j$, por lo que:

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = -(t_j - o_j)$$

Como $o_j = \sigma(net_j)$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

Desarrollo - Capa de salida

Que en el caso de la sigmoide, la derivada es:

$$= \sigma(net_j)(1 - \sigma(net_j)) = o_j(1 - o_j)$$

Por lo que:

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)o_j(1 - o_j)$$

y finalmente:

$$\Delta w_{ij} = -\alpha \frac{\partial E_d}{\partial w_{ij}} = \alpha(t_j - o_j)o_j(1 - o_j)x_{ij}$$

Desarrollo - Capa oculta

- Ahora en la regla de actualización del peso w_{ij} se deben de considerar las formas indirectas en las que pudo contribuir al error (de alguna forma estamos distribuyendo el error), por lo que consideramos todos los nodos a los cuales les llega la salida del nodo oculto j.
- Vamos a denotar: $\delta_i = -\frac{\partial E_d}{\partial net_i}$

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in sal(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$\delta_j = \sum_{k \in sal(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$\delta_j = \sum_{k \in sal(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

Desarrollo - Capa oculta

- $\frac{\partial net_k}{\partial o_j}$ es diferente de cero, sólo cuando tenemos el término $w_{jk} \cdot x_{jk}$ (donde $x_{jk} = o_j$) en la sumatoria, por lo que:

$$\delta_j = \sum_{k \in sal(j)} -\delta_k w_{jk} \frac{\partial o_j}{\partial net_j}$$

De nuevo para la sigmoide:

$$\delta_j = \sum_{k \in sal(j)} -\delta_k w_{jk} o_j (1 - o_j)$$

$$\delta_j = o_j (1 - o_j) \sum_{k \in sal(j)} -\delta_k w_{jk}$$

Lo que corresponde a la fórmula del inciso 2(b). Finalmente:

$$\Delta w_{ij} = \alpha \delta_j x_{ij}$$

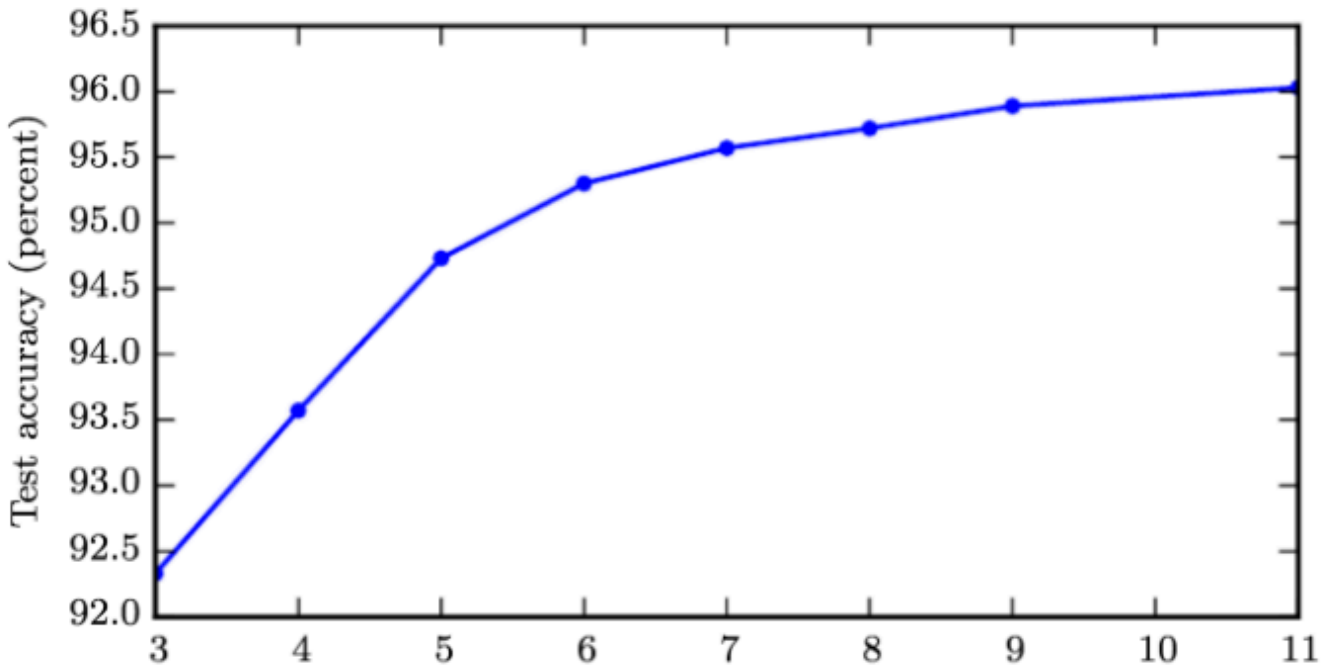
Deep Learning

- El tamaño de las capas le dan la profundidad al modelo
- Las NN son aproximadores de funciones
- Sus ventajas es que pueden encontrar representaciones intermedias o atributos automáticamente, librando al usuario de eso

Deep Learning

- Las redes neuronales son aproximaciones de funciones universales, lo que quiere decir es que no importa que función sea, se puede tener una red nueronal suficientemente grande para representarla
- No se garantiza, sin embargo, que el algoritmo de aprendizaje la pueda aprender
- El usar redes profundas puede ayudar a encontrar esas representaciones

Profundidad vs. Desempeño



Deep Learning

- Las ideas principales sobre las redes neuronales modernas no han cambiado mucho desde los 80's.
- Los resultados recientes de DL se pueden atribuir a: (i) grandes bases de datos y (ii) redes neoronales más grandes
- Con algunos cambios:
  - Cambar el error cuadrático medio por las funciones de pérdida basadas en entropía cruzada en problemas de clasificación
  - Cambiar la función sigmoide por ReLU

Deep Learning

- Hay varios aspectos que se tienen que elegir:
  - cómo optimizar
  - la función de costo
  - la función de activación
  - la arquitectura
- La no linearidad de las redes neuronales hacen que las funciones de costo sean no convexas
- Por esto se entrenan de forma iterativa usando gradiente descendente
- El gradiente descendente estocástico no tiene garantías de convergencia y depende de los valores de inicialización

Función de Evaluación

- En general nuestro modelo paramétrico define una distribución:

$$P(\mathbf{y}|\mathbf{x};\Theta)$$

- Lo que significa que la función de costo es el negativo del logaritmo de la máxima verosimilitud (con un término de regularización)

Funciones de Pérdida

- Como no podemos calcular los pesos perfectos en una red neuronal (demasiados parámetros), se plantea como un problema de optimización y se usa, generalmente, SGD buscando mover los pesos para reducir el error
- La función de pérdida regresa un sólo valor que permite comparar soluciones candidatas y capturar las propiedades del problema
- Existen diferentes funciones de pérdida, pero se quiere que se puedan mapear a un espacio de alta dimensionalidad suave

Funciones de Pérdida

- La estimación de máxima verosimilitud (MLE) busca optimizar los valores de los parámetros maximizando una función de verosimilitud derivada de los datos de entrenamiento
- Lo que hace es que estima qué tan cerca está la distribución de las predicciones del modelo con respecto a la distribución de los datos
- Uno de los beneficios de este estimador es que al aumentar el número de datos, el estimador de los parámetros mejora (consistencia)
- El error entre distribuciones de probabilidad se mide usando entropía cruzada

Función de Evaluación

- Por ejemplo, si estamos en un problema de regresión y el modelo lo suponemos como una Gaussiana:

$$p_{modelo}(y|x) = \mathcal{N}(y; f(x, \Theta), I)$$

- Entonces la función de costo o de pérdida es el error cuadrático medio:

$$J(\Theta) = \frac{1}{2} E_{x,y \sim \hat{p}_{data}} ||\mathbf{y} - f(x; \Theta)||^2 + const$$

- Un tema recurrente en las redes neuronales es que el gradiente de la función de costo debe de ser suficientemente grande
- Las funciones que se saturan (*saturate*) hacen que el gradiente se pueda volver muy pequeño

Entropía Cruzada

- La entropía cruzada es:

$$H(P, Q) = -\mathcal{E}_{x \sim P}(logQ(x))$$

- Esto se puede expresar en términos de la divergencia de Kullback-Leibler (KL):

$$H(P, Q) = H(P) + D_{KL}(P||Q)$$

donde, $H(P)$ es la entropía: $-\mathcal{E}_{x \sim P} logP(X)$ que en el caso discreto es: $H(P) = -\sum_X P(X) logP(X)$

Entropía Cruzada

$$D_{KL}(P||Q) = \mathcal{E}_{x\sim P}\left[log\frac{P(X)}{Q(X)}\right] = \mathcal{E}_{x\sim P}\left[logP(X) - logQ(X)\right]$$

- Para distribuciones de probabilidad discretas:

$$H(P,Q) = -\sum_{X} P(X)logQ(X)$$

- Minimizar la entropía cruzada con respecto a Q es equivalente a minimizar la divergencia KL, en donde P es la distribución real (tomada de los datos) y Q es la distribución estimada (tomada del modelo)

Deep Learning

La selección de la función de pérdidad depende de la función de activación de la capa de salida:

- En el caso de problemas de regresión es común usar el error cuadrático medio (MSE) - que se deriva de minimizar la verosimilitud y suponer una distribución Gausiana en el ruido de los datos
- Para problemas de clasificación se usa la entropía cruzada que tiende a aprender más rápido que usando MSE

Función de Evaluación

- En las unidades intermedias, normalmente se usa ReLU la cual no es diferenciable cuando el valor = 0 (aunque en general no se espera que los valores alcancen 0) i.e., no pueden aprender si su valor de activación vale cero.
- Hay varios remedios (*absolute value rectification, leaky ReLU, parametric ReLU* ó *PReLU, Maxout units*)
- Antes de ReLU lo que se usaba era la función sigmoide o la tangente hiperbólica, por sus propiedades al momento de derivarlas
- El problema es que estas funciones se saturan (comprimen todos los posibles valores a un pequeño rango [0..1] o [-1..1] y es difícil aprender.

Regularización

- Un aspecto fundamental en todos los algoritmos de aprendizaje es cómo hacer que el algoritmo se comporte bien, no sólo en los ejemplos de entrenamiento, sino también en ejemplos nuevos.
- En general se tienen varias estrategias para reducir el error de prueba, las cuales en general, se conocen como regularización.
- En general, se hace añadiendo restricciones en los valores de los parámetros o añadiendo términos adicionales en la función objetivo.
- Estos mecanismos muchas veces incorporan conocimiento *a priori* o preferencias hacia modelos más simples.

Regularización

- En DL, la mayoría se basan en estimadores de regularización que aumentan el sesgo, pero reducen la varianza.
- Muchos añaden un parámetro de penalización:

$$\tilde{J}(\Theta; X, y) = J(\Theta; X, y) + \alpha\Omega(\Theta)$$

- Uno de los más comunes es usando la norma L^2 conocido como decaimiento de pesos, el cual mueve los pesos hacia el origen (0).

Regularización

- También se conoce como Ridge regression o Tikhonov regularization

$$\tilde{J}(\Theta; X, y) = J(\Theta; X, y) + \frac{\alpha}{2} w^T w$$

- El gradiente queda entonces:

$$\Delta_w \tilde{J}(\Theta; X, y) = \alpha w + \Delta_w J(\Theta; X, y)$$

- Y la actualización de pesos en un solo paso queda:

$$w \leftarrow w - \epsilon(\alpha w + \Delta_w J(\Theta; X, y))$$

$$w \leftarrow (1 - \epsilon\alpha) w - \epsilon \Delta_w J(\Theta; X, y)$$

- Lo que reduce el tamaño de los pesos por un factor constante en cada paso

Regularización

- Otra opción es usar la regularización L^1

$$\Omega(\Theta) = ||w||_1 = \sum_i |w_i|$$

- Por lo que la función objetivo queda:

$$\tilde{J}(\Theta; X, y) = J(\Theta; X, y) + \alpha||w||_1$$

- y el gradiente queda como:

$$\Delta_w \tilde{J}(\Theta; X, y) = \alpha signo(w) + \Delta_w J(\Theta; X, y)$$

Dataset Augmentation

- Otra forma de mejorar la clasificación es aumentando el número de datos
- Esto es importante en DL en donde se tiene una gran cantidad de parámetros a optimizar, por lo que se requiere tener una gran cantidad de datos para no sobreajustar
- Lo que se hace es hacer transformaciones sobre los datos originales
- En imágenes es más fácil (traslandando un poco los pixeles, añadiendo ruido, rotando las imágenes, ...)
- Es importante, que al comparar algoritmos, si se generan datos, que se haga lo mismo en todos los algoritmos

Regularización

- Otra forma en que se ha usado ruido para efectos de regularización, es añadiendo ruido en los pesos
- Otro punto importante es decidir cuando parar, para no sobre-ajustar
- Para esto, lo común es tener un conjunto de validación (se divide el conjunto de entrenamiento en dos)
- Una forma de incluir conocimiento previo es si sabemos que ciertos parámetros deben de tener algún tipo de relación
- Uno de los más populares es forzar que los parámetros sean los mismos
- Esto es lo que se usa en las redes neuronales convolucionales

Dropout

- Quita temporalmente algunas unidades de la red multiplicando sus salidas por cero
- Técnica común al usar mini-batch. Cada vez que se usa un mini-batch usamos una mascara binaria para eliminar algunos nodos
- Normalmente, las unidades de entrada se incluyen con una probabilidad de 0.8 y las unidades ocultas con 0.5

Adversarial Training

- La idea es buscar ejemplos que la red clasifique mal
- Se probó que se puede “engañar” fácilmente a las redes haciendo pequeños cambios en los ejemplos
- Si se hace *adversarial training* se puede reducir el error
- La suposición es que clases diferentes deberían de estar en espacios diferentes, por lo que perturbaciones pequeñas no nos deberían de cambiar el valor de las clases.

Optimización y Aprendizaje

- En ML normalmente se minimiza el riesgo empírico (*empirical risk*) que sin embargo, es susceptible de sobreajustar
- Por lo que a veces se usa el negativo del logaritmo de la verosimilitud no la función directamente
- Otra diferencia es que en optimización se para en el óptimo, en ML muchas veces se detiene el proceso bajo otro criterio (*early stopping*)

Batch y Minibatch

- Calcular el gradiente sobre todos los ejemplos es muy costoso, en la práctica lo que se hace es que se calcula haciendo un muestreo aleatorio y sacando el promedio, pero sólo de esos ejemplos (*minibatch*)
- Los algoritmos que usan todos los ejemplos (*batch*) a veces se les llama de gradiente determinísta
- Los algoritmos que usan un solo ejemplo a la vez, se les llama estocásticos o en-línea
- Actualmente la mayoría toma un subconjunto de ejemplos y obtiene el promedio, a estos se les llama *minibatch* o minibatch estocásticos

Minibatch

- Batches grandes dan un mejor estimado del gradiente pero proporcionalmente no mejoran tanto
- Los batches deben de ser seleccionados aleatoriamente, con muestreos independientes, y minibatches consecutivos también independientes

Algoritmos de Optimización

- Los problemas generales de gradiente tienen que ver con mínimos locales, *plateaus*, riscos, dependencias lejanas, ... y existen varios algoritmos
- Stochastic Gradient Descent (SGD): Es el más usado. El parámetro clave es la razón de aprendizaje.
- En general, se decrece gradualmente con el tiempo, una regla común es:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

$$\alpha = \frac{k}{\tau}$$

- Después de la iteración τ se deja ϵ constante

Stochastic Gradient Descent

τ se selecciona para que se hagan cientos de pasadas sobre el conjunto de datos y ϵ_τ como el 1% del valor de ϵ_0

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

Algoritmo con Momento

- El algoritmo con momento introduce una variable v que juega el papel de velocidad (qué tan rápido se mueven los parámetros):

$$v \leftarrow \alpha v - \epsilon \nabla_{\Theta} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \Theta), y^{(i)}) \right)$$

$$\Theta \leftarrow \Theta + v$$

Algoritmo con Momento

El paso de actualización aumenta cuando gradientes sucesivos apuntan en la misma dirección. Valores comunes de α son 0.5, 0.9 y 0.99.

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

Nesterov Momentum

- Es una variante del algoritmo de momento, en donde se añade una corrección

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Inicialización

- Los valores iniciales pueden determinar si el algoritmo converge o no y la verdad no se tiene claro qué puede ser una buena inicialización
- En general se hace de forma aleatoria
- Una heurística que funciona en general es inicializar todos los parámetros como:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

donde m son el número de entradas y n es el número de salidas

AdaGrad

- El AdaGrad adapta la razón de aprendizaje escalandolo inversamente proporcional a la raíz cuadrada de la suma del los valores al cuadrado históricos del gradiente
- Los parámetros con las derivadas parciales grandes tienen un decrecimiento rápido en su razón de aprendizaje, mientras que los parámetros que tiene valores bajos en sus derivadas parciales tienen un decrecimiento pequeño en su razón de aprendizaje
- Tiene el riesgo de decrecer prematuramente la razón de aprendizaje

AdaGrad

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSProp

- La única diferencia es que cambia la acumulación del grandiente en una media móvil pesada

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

RMSProp con Nesterov Momentum

Lo mismo pero con momento

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.

Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

Adam (adaptive moments)

- En al algoritmo Adam el momento se incorpora directamente como un estimado del momento de primer orden
- También incluye un sesgo en las correcciones a los estimados de los momentos de primer y segundo orden

Adam (adaptive moments)

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Métodos de Segundo Orden

- La diferencia de estos métodos es que en lugar de usar la primera derivada, utilizan la segunda derivada
- El método de Newton es un esquema de optimización basado en usar la expansión de serie de Taylor de segundo orden

$$J(\Theta) \approx J(\Theta_0)+(\Theta-\Theta_0)^T\nabla_\Theta J(\Theta_0)+\frac{1}{2}(\Theta-\Theta_0)^T H(\Theta-\Theta_0)$$

donde H es el Hessiano de J con respecto a Θ evaluado en Θ_0

Métodos de Segundo Orden

- Resolviendo se obtiene la regla de actualización:

$$\Theta^* = \Theta_0 - H^{-1}\nabla_\Theta J(\Theta_0)$$

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of m examples
  while stopping criterion not met **do**
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Métodos de Segundo Orden

- El problema obvio es calcular el inverso del Hessiano en cada iteración de entrenamiento
- El método del gradiente conjugado evita el cálculo del inverso del Hessiano
- Otro esquema es Broyden-Fletcher-Goldfarb-Shanno (BFGS), un método quasi-Newton que hace una aproximación de la inversa del Hessiano con otra matriz
- Este último se puede mejorar reduciendo su uso de memoria (Limited Memory BFGS)

Batch Normalization

- Uno de los problemas con redes muy produndas es que involucran la composición de varias funciones (o capas).
- La actualización puede tener resultados inesperados porque muchas funciones compuestas se cambian al mismo tiempo bajo la suposición que las otras funciones se mantienen constantes
- Sea H el minibatch de activaciones de la capa a normalizar organizadas en una matriz donde la activación de cada ejemplo es una fila.
- H se reemplaza por:

$$H' = \frac{H - \mu}{\sigma}$$

$$\mu = \frac{1}{m}\sum_i H_i; \sigma = \sqrt{\delta + \frac{1}{m}\sum_i (H - \mu)_i^2}$$

Otras Técnicas

- *Greedy*: Partir el problema en partes, resolver cada parte por separado y posiblemente hacer al final un *fine tuning*
- *FitNets*: Aprender una red poco profunda, pero ancha y que esta sirva como “maestro” de otra red más profunda y delgada
- *Curriculum Learning*: Aprender primero conceptos sencillos y después los conceptos más complicados, que dependen de los sencillos

Deep Learning

Existen 3 clases de arquitecturas de *deep learning*:

- Arquitecturas generativas: en general se hace un entrenamiento no supervisado como pre-entrenamiento, donde se aprende de manera "greedy" capa por capa en forma "bottom-up". En esta categoría están los modelos basados en energía, que incluyen los *auto-encoders* y las *máquinas de Boltzmann restrictivas* (RBM)
- Arquitecturas discriminativas: *Conditional Random Fields* profundos y las redes neuronales convolucionales (*Convolutional Neural Network*) o CNN
- Arquitecturas Híbridas: Usan los dos esquemas

Redes de Convolución

- Convolución es un proceso en donde se mezclan dos fuentes de información
- Cuando se aplica una convolución en imágenes se hace en 2 dimensiones.
- Fuentes:
  - La imagen de entrada (una matriz de 3 dimensiones - espacio de color, 1 por cada color con valores enteros entre 0 y 255)
  - Un kernel de convolución, que es una matriz de números cuyo tamaño y valores dice cómo mezclar la información

Redes de Convolución

- La salida es una imagen alterada llamada mapa de atributos (*feature map*) en DL
- Existe un mapa de atributos por cada color
- Una forma de aplicar convolución es tomar un pedazo (*patch*) de la imagen del tamaño del kernel
- Hacer una multiplicación de cada elemento en orden de la imagen con el kernel
- La suma de las multiplicaciones nos da el resultado de un pixel

Redes de Convolución

- Después se recorre el *patch* un pixel en otra dirección y se repite el proceso hasta tener todos los pixeles de la imagen.
- Se normaliza la imagen por el tamaño del kernel para asegurarse que la intensidad de la imagen no cambie
- Convolución se usa en imágenes para filtar información indeseable
- Por eso, convolución también se llama filtrado y a los kernels también les dicen filtros
- E.g., de filtro de sobel para obtener bordes

Redes de Convolución

- Tomar una entrada, transformarla y usar la imagen transformada como entrada a un algoritmo se conoce como ingeniería de atributos (*feature engineering*)
- La ingeniería de atributos es difícil y se tiene que partir de cero para nuevas imágenes
- Las redes de convolución se usan para encontrar los valores del kernel adecuados para realizar una tarea
- En lugar de fijar valores numéricos al kernel, se asignan parámetros que se aprenden con datos

Redes de Convolución

- Conforme se entrena la red de convolución, el kernel filtra mejor la imagen para conseguir la información relevante
- Este proceso también se llama aprendizaje de atributos (*feature learning*) y es uno de los atractivos de usar DL
- Normalmente no se aprende un solo kernel sino una jerarquía de múltiples kernels al mismo tiempo
- Por ejemplo, si aplicamos un kernel de $32 \times 16 \times 16$ a una imagen de 256×256 nos produce 32 mapas de atributos de 241×241 (tamaño de la imagen - tamaño kernel + 1)

Redes Convolucionales

- Las redes convolucionales o CNNs son redes neuronales que usan convolución, en lugar de multiplicación de matrices en al menos uno de sus capas
- La convolución en una operación de dos funciones

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

normalmente se denota con un "*"

$$s(t) = (x * w)(t)$$

- En el caso discreto se tiene:

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

Redes Convolucionales

- En ML la entrada (x) es un arreglo multidimensional de datos y el kernel (w) es un arreglo multidimensional de parámetros
- En general, se hace convolución sobre más de una dimensión o eje a la vez:

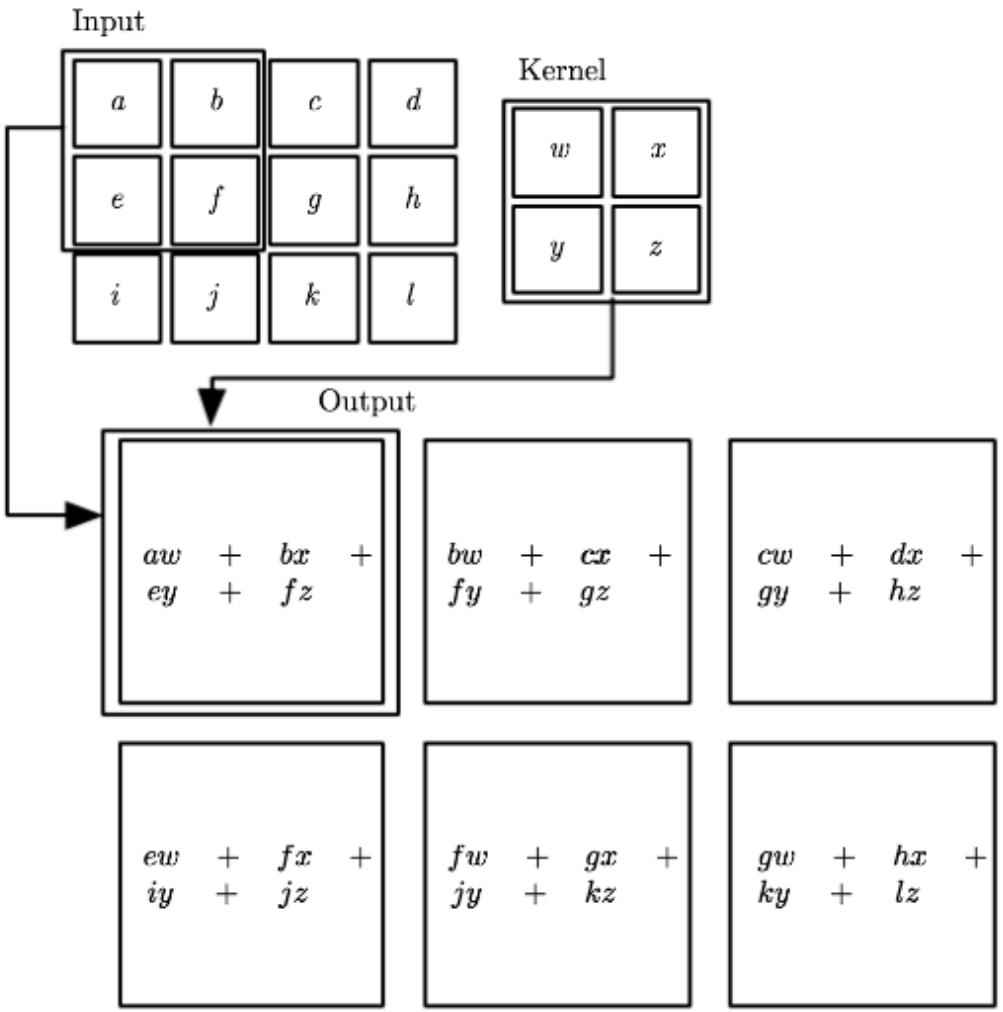
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

- La convolución es conmutativa y está relacionada con la correlación cruzada que es la que mormalmente se implementa en los algoritmos de CNN:

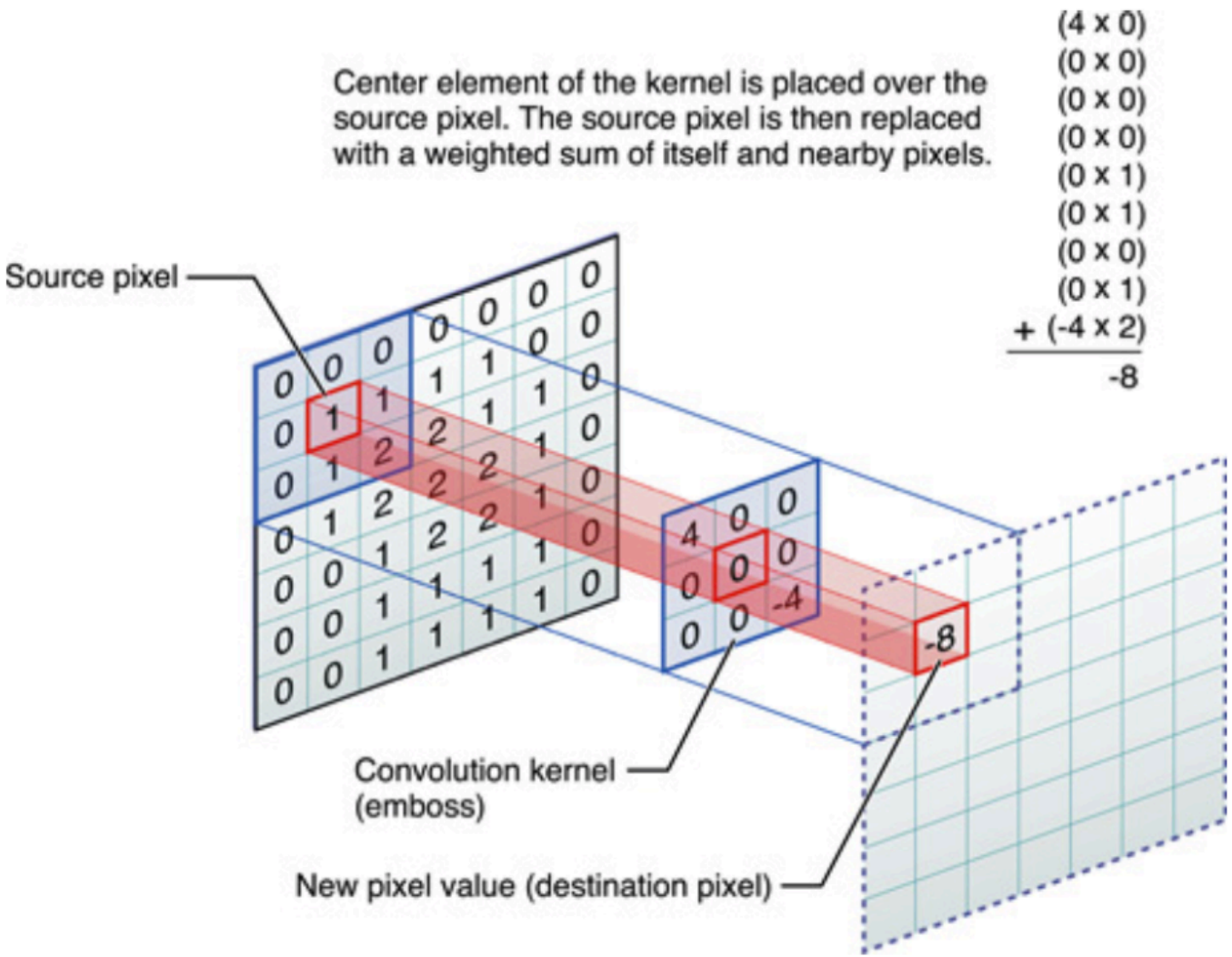
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

- La convolución se puede ver como una multiplicación de matrices en donde la matriz kernel es mucho más pequeña que la entrada (imagen)

Redes Convolucionales



Redes Convolucionales



Redes Convolucionales

Original

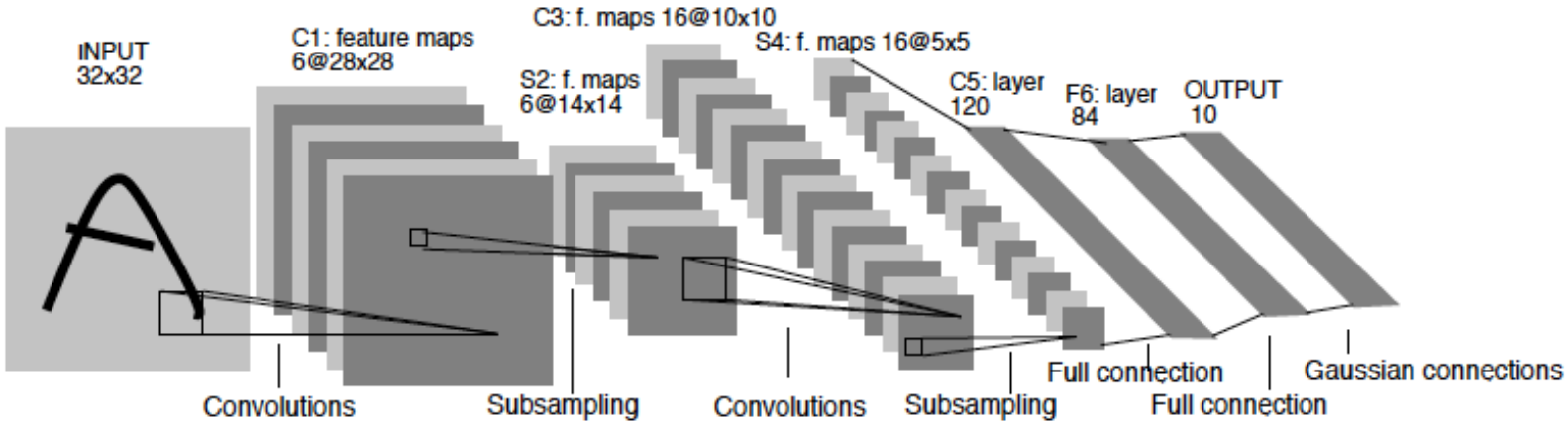
Emboss

-2	-2	0
-2	6	0
0	0	0

Redes de Convolución

- Estos mapas sirven de entrada para el siguiente kernel.
- Una vez aprendida la jerarquía de kernels se pasa a una red neuronal normal que los combina para clasificar la imagen de entrada en clases.

Ejemplo: LeNet-5

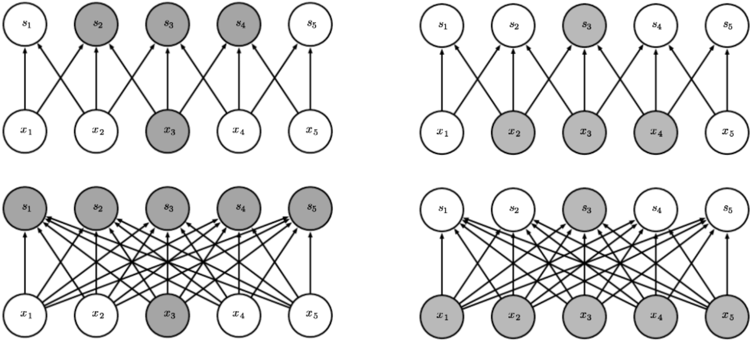


Redes Convolucionales

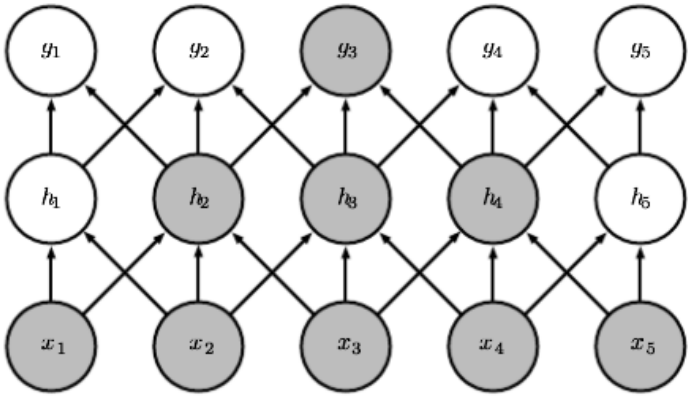
Algunos de las caracterísitcas de las CNNS son:

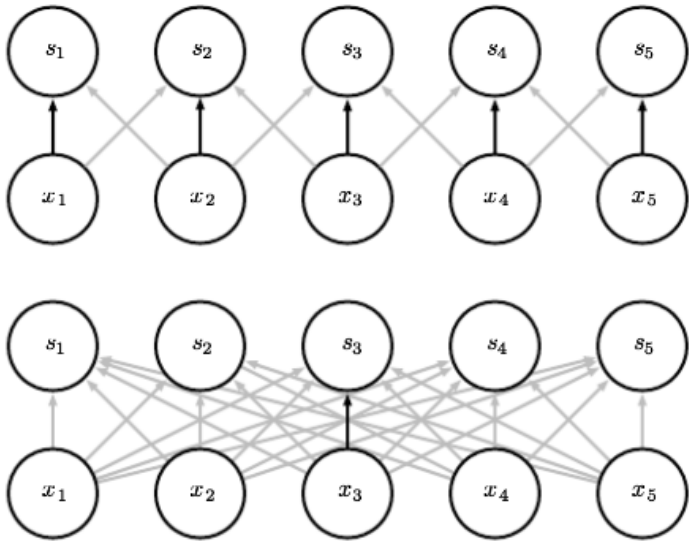
- Se usan varios kernels (se aprenden parámetros de cada uno)
- Cada kernel se aplica varias veces en las imágenes (deslizandolos por toda la imagen)
- Lo que tienen las CNNs es que comparten parámetros, los pesos de cada kernel son iguales en todas las entradas
- Lo que hace la red es aprender los parámetros del kernel

Redes Convolucionales



Redes Convolucionales





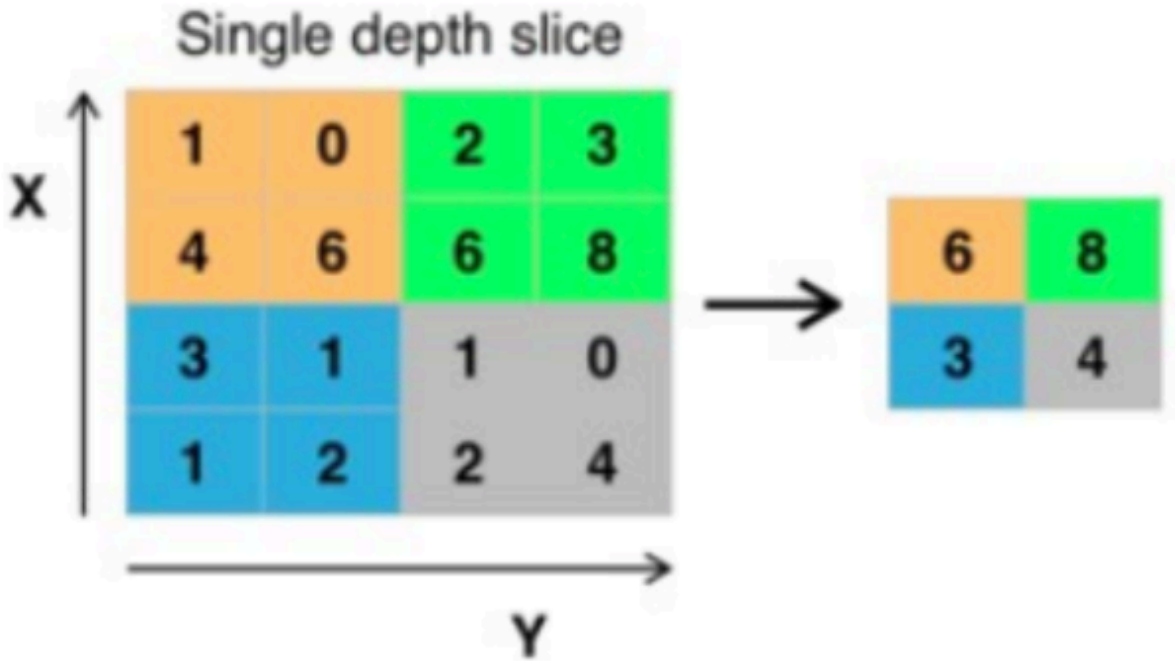
Pooling

Una capa de convolución tradicional tiene 3 pasos:

- Aplicar la convolución en paralelo
- Pasar la salida por una función de activación (e.g., ReLU)
- Pooling

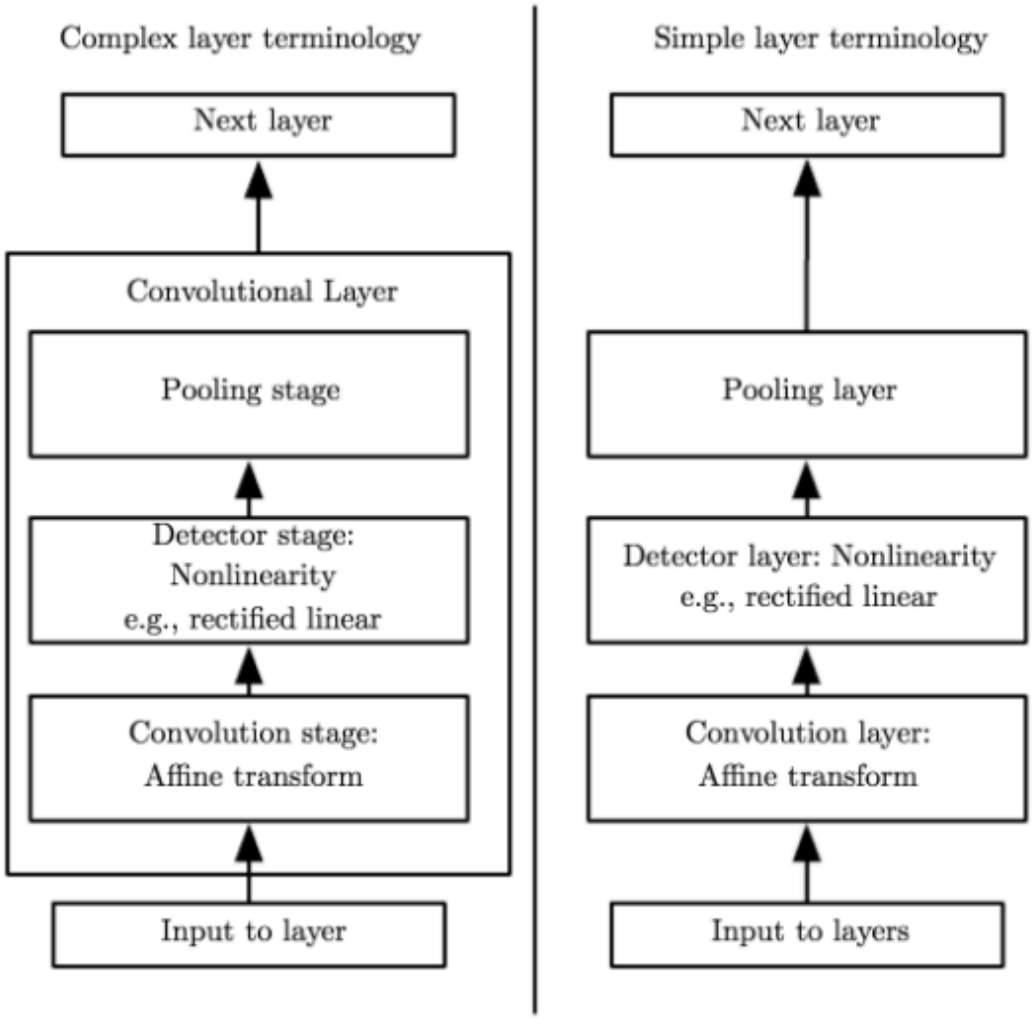
Lo que hace pooling es remplazar parte de la salida de la red por un estadístico, por ejemplo, max-pooling (regresa el máxima de los valores de un rectángulo)

Pooling



Además de max, existen otras varianes como sum, avg, min

Pooling



Pooling

- Pooling crea un fuerte sesgo al suponer que las entradas son invariantes a pequeñas traslaciones
- Los pesos de una unidad de una cada oculta son iguales a los pesos de sus vecinos, sólo trasladados en el espacio
- Esto puede crear un sub-ajuste (*underfitting*)

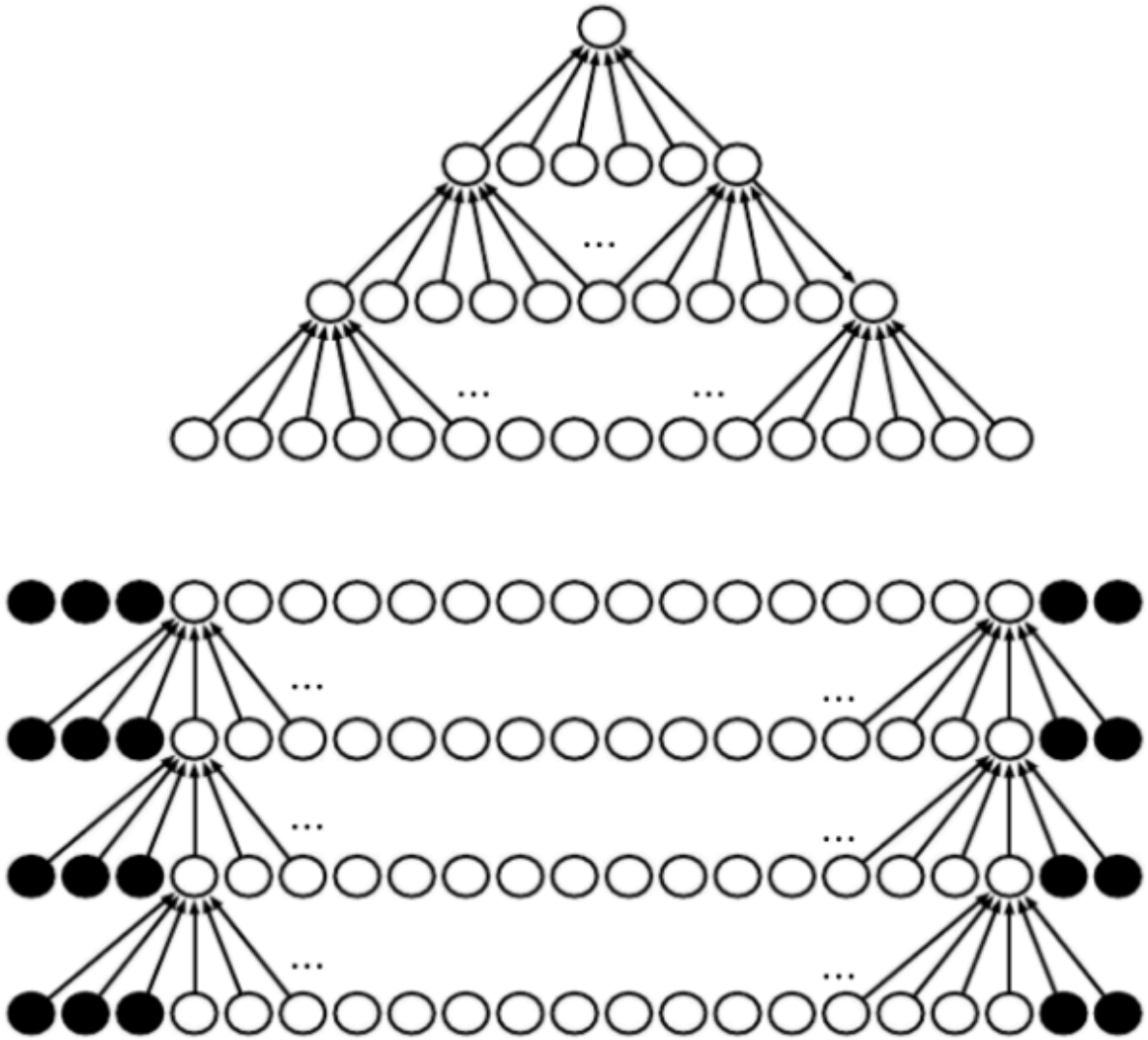
Redes Convolucionaes

- Al usar imágenes, normalmente, más que matrices, usamos tensores, ya que se tienen 3 canales (RGB) y varios ejemplos (4D)
- Un parámetro importante al usar los kernels es el *stride*, que significa saltarse algunos lugares al aplicar el kernel
- Esto simplifica el costo computacional y se puede ver como un submuestreo (*downsampling*) de la salida.

¿Cómo tratar las orillas?

- Si queremos que se aplique el kernel sobre todos los datos definidos (*valid*), entonces la imagen se reduce en forma proporcional al tamaño del kernel
- Podemos llenar los huecos que existen en las orillas por ceros para mantener el mismo tamaño, pero puede afectar los pesos que se aprenden de los kernels

¿Cómo tratar las orillas?



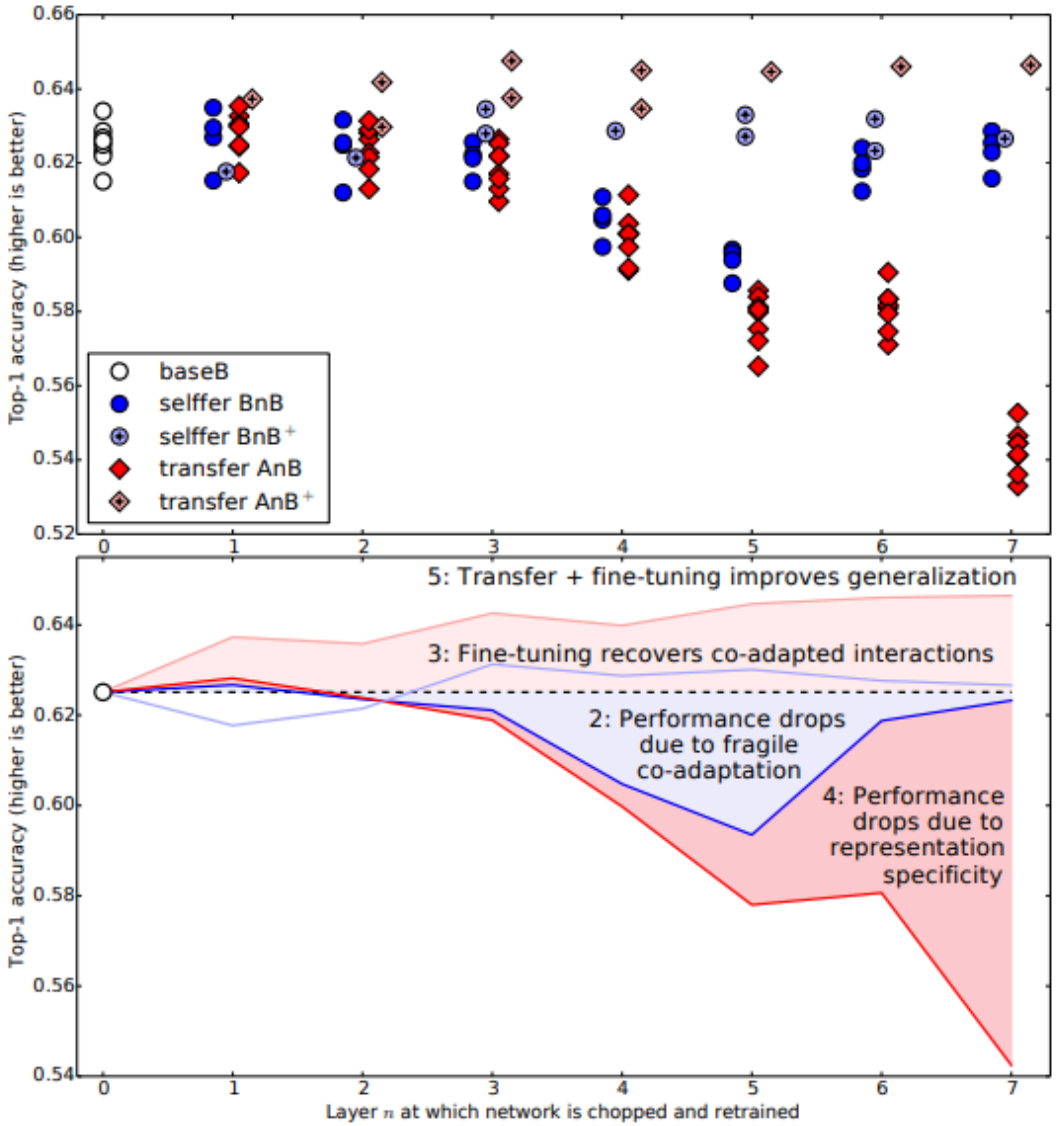
Redes Convolucionales

- Para el aprendizaje uno necesita calcular el gradiente con respecto al kernel, dado el gradiente con respecto a las salidas (ver Goodfellow 2010)
- Para las capas de pooling (e.g., max pooling) se pasa el gradiente sólo a los puntos usados (i.g., máximos) con los que se hizo el cálculo del error
- Para esto se requiere multiplicar por la traspuesta de la matriz definida por la convolución para facilitar el cálculo para propagar el error ($\delta w = w^T \delta$)

Transfer Learning

- Dados los requerimientos computacionales para entrenar una red, es común re-utilizar una red pre-entrenada con millones de imágenes:
  1. Uso directo: Se utilizan las salidas de una capa intermedia como atributos de un clasificador (extractor de atributos)
  2. Modelo a la medida: Se re-utilizan algunas capas y se realiza *fine tuning* sobre las capas de interés

Transfer Learning



Deep Autoencoders

Deep Autoencoder: Tiene al menos 3 capas en una red neuronal

- Capa de entrada
- Capa oculta
- Capa de salida

Input = decoder(encoder(Input))

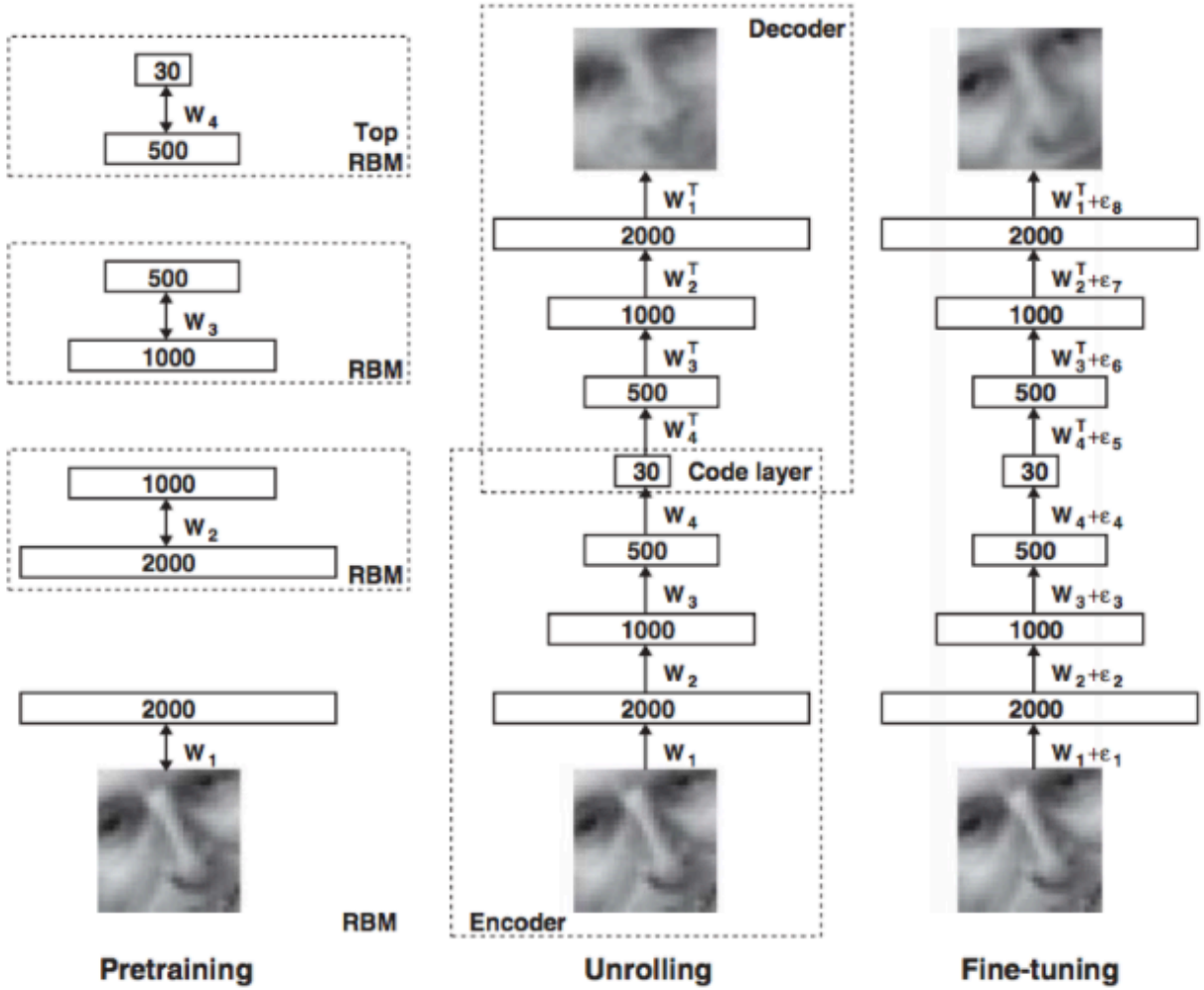
Deep Autoencoder

Es el más fácil de entender, la idea es la siguiente:

- Usa una capa de entrada, una capa oculta con un menor (aunque puede ser mayor) número de nodos (*undercomplete AE*) y entrena para obtener una capa de salida igual a la entrada
- Usa lo que se aprendió en la capa oculta (abstracción) como la nueva capa entrada para el siguiente nivel
- Continúa con los niveles determinados por el usuario
- Al final, usa la última capa oculta como entrada a un clasificador

Deep Autoencoder

Deep Autoencoder fue lo que usó Hinton en su artículo de Science



Máquinas de Boltzmann Restringidas (RBM)

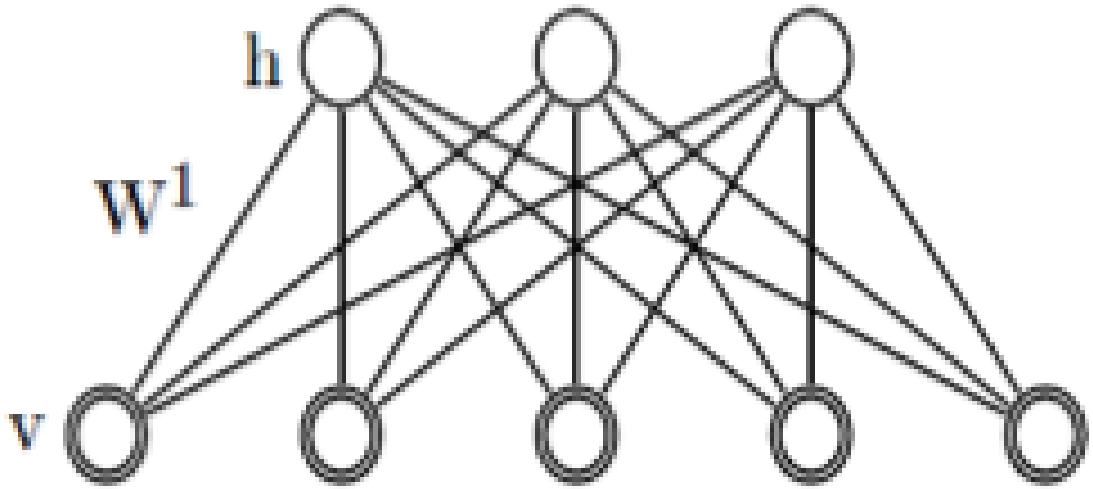
- Una RBM es un tipo de campo aleatorio de Markov que tiene una capa de unidades oculta (típicamente Bernoulli) y una capa de unidades observables (típicamente Bernoulli o Gaussiana).
- Se puede representar como un grafo bipartita en donde todos los nodos visibles están conectados a todos los nodos ocultas, y no existen conecciones entre nodos de la misma capa.

RBM

- Se construye una RBM con la capa de entrada con ruido gaussiano y una capa oculta binaria.
- Depués se aprende en la siguiente capa, tomando como capa de entrada la capa oculta binaria anterior, un RBM binario-binario.
- Esto continua hasta la última capa en donde se ponen las salidas y se entrena todo con *backpropagation*

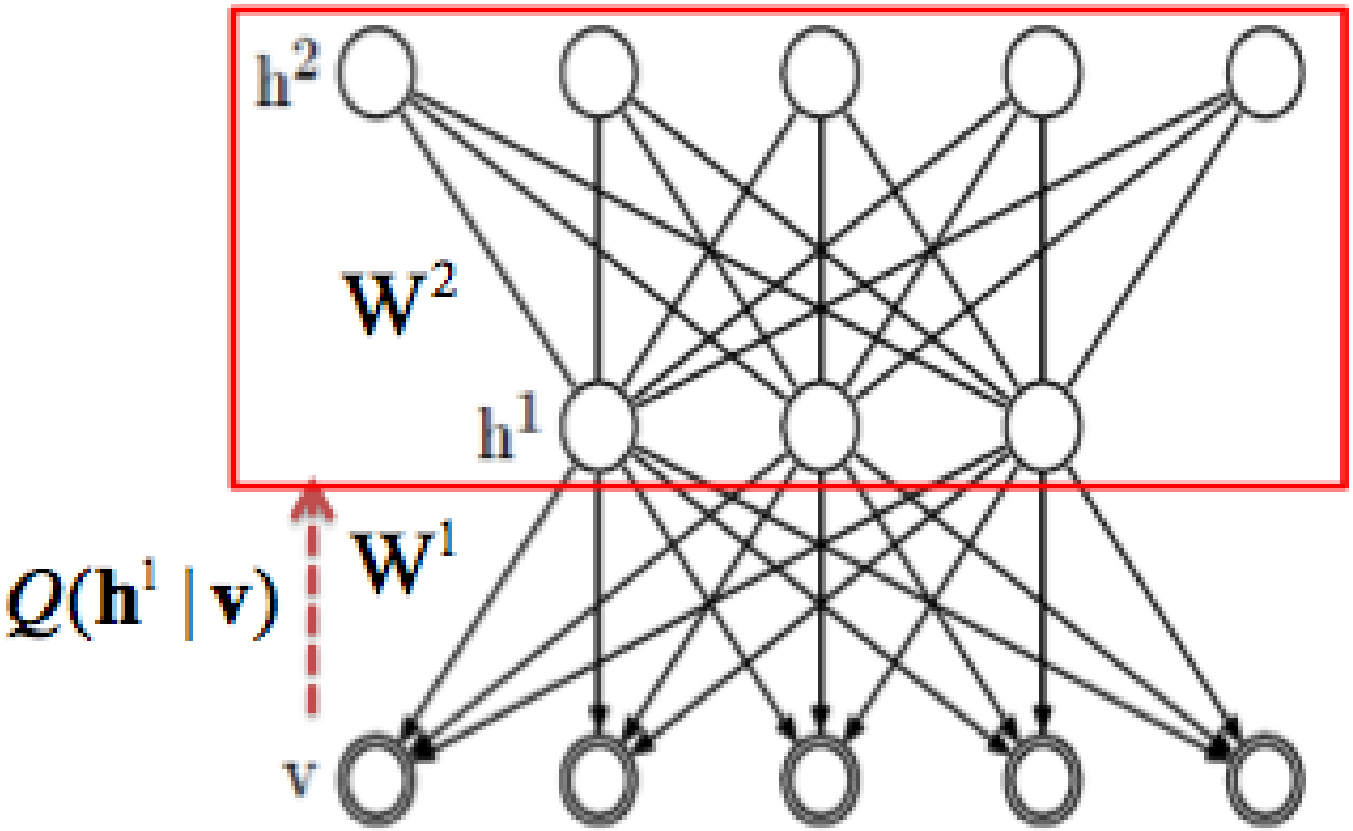
RBM

Se construye y entrena una RBM con una capa de entrada (visible) y una capa oculta



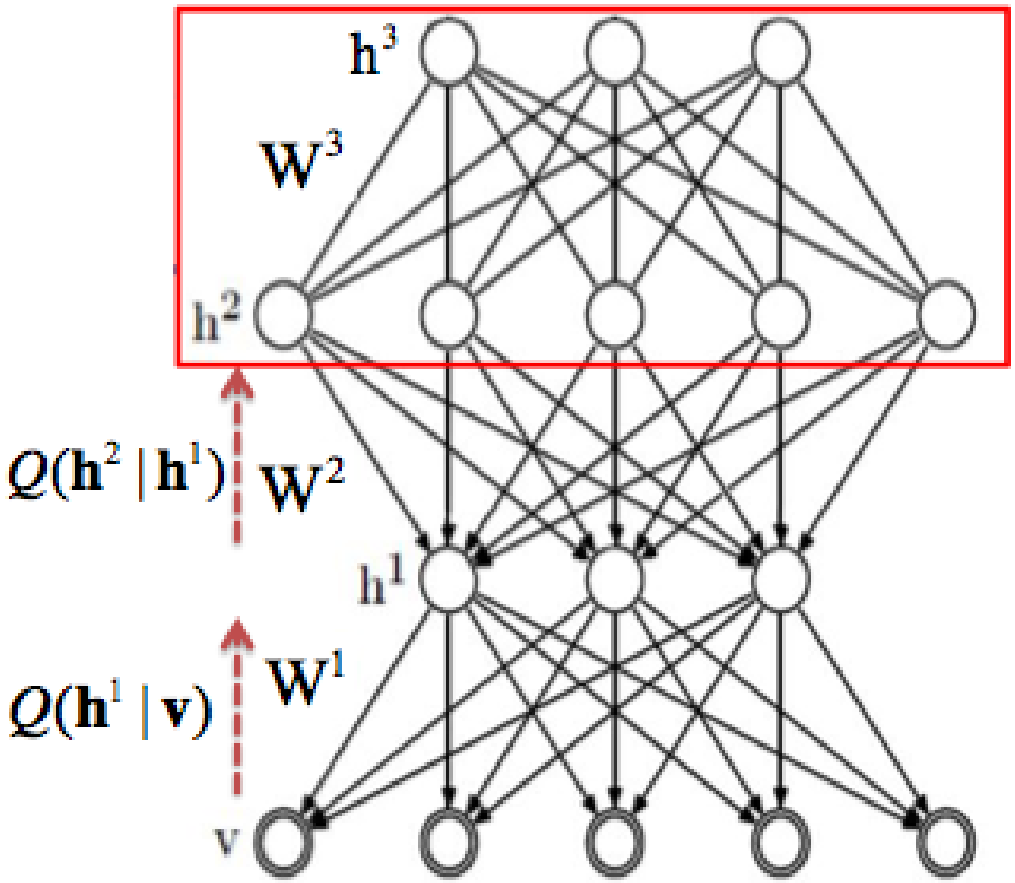
RBM

Se apila otra capa oculta, se fijan los pesos de la primera capa, se muestrean los valores de la primera capa oculta usando el modelo y los valores de entrada y se entrenan los pesos de la nueva capa.

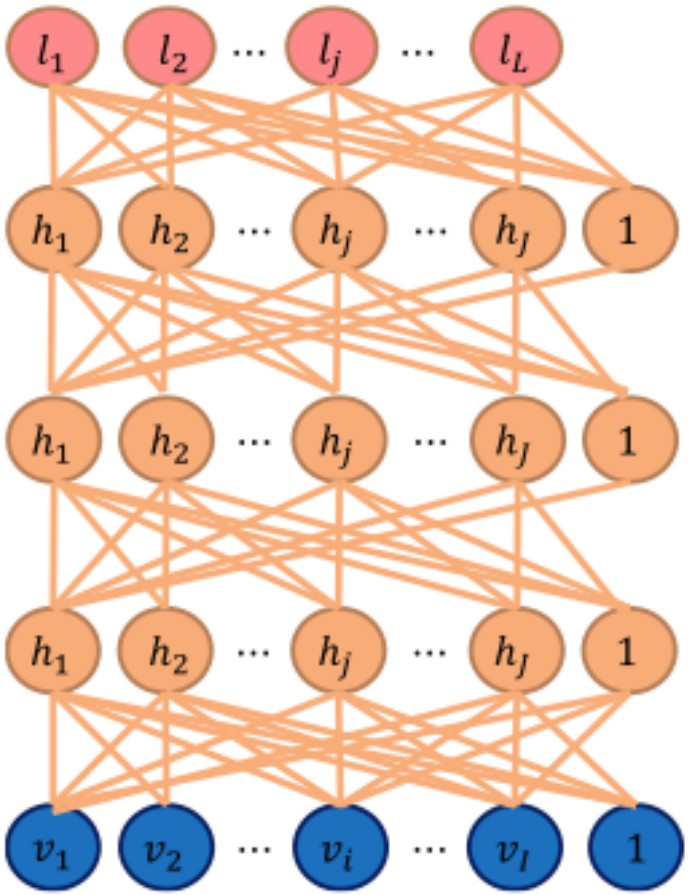


RBM

Se apila otra capa y se entrena igual tomando muestras de la segunda capa oculta, dados los valores de la primera capa oculta.



RBM



RBM

- La distribución conjunta de las unidades visibles (**v**) y las unidades ocultas (**h**) dados los parámetros del modelo (θ), $p(\mathbf{v}, \mathbf{h}; \theta)$ se define en términos de una función de energía $E(\mathbf{v}, \mathbf{h}; \theta)$:

$$p(\mathbf{v}, \mathbf{h}; \theta) = \frac{exp(-E(\mathbf{v}, \mathbf{h}; \theta))}{Z}$$

donde Z es un factor de normalización o partición:

$$Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} exp(-E(\mathbf{v}, \mathbf{h}; \theta))$$

RBM

- La probabilidad marginal del vector visible se obtiene sumando sobre todos los nodos ocultos:

$$p(\mathbf{v};\theta) = \frac{\sum_{\mathbf{h}} exp(-E(\mathbf{v},\mathbf{h};\theta))}{Z}$$

- Para una RBM Bernoulli (visible) - Bernoulli (oculta), la función de energía se define como:

$$E(\mathbf{v},\mathbf{h};\theta) = -\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij} v_i h_j - \sum_{i=1}^{I} b_i v_i - \sum_{j=1}^{J} a_j h_j$$

donde w_{ij} representa la interacción simétrica entre las unidades visibles (v_i) y las ocultas (h_j), b_i y a_j son sesgos e I y J son el número de unidades visibles y ocultas.

RBM

- Las probabilidades condicionales se pueden calcular como:

$$p(h_j = 1|\mathbf{v};\theta) = \sigma\left(\sum_{i=1}^{I} w_{ij} v_i + a_j\right)$$

$$p(v_i = 1|\mathbf{h};\theta) = \sigma\left(\sum_{j=1}^{J} w_{ij} h_i + b_i\right)$$

donde $\sigma(x) = 1/(1 + exp(x))$

RBM

- De la misma forma para una RBM Gaussiana (visible) - Bernoulli (oculta) la energía es:

$$E(\mathbf{v}, \mathbf{h}; \theta) = -\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij} v_i h_j - \frac{1}{2}\sum_{i=1}^{I}(v_i - b_i)^2 - \sum_{j=1}^{J} a_j h_j$$

- Las probabilidades condicionales son:

$$p(h_j = 1|\mathbf{v}; \theta) = \sigma\left(\sum_{i=1}^{I} w_{ij} v_i + a_j\right)$$

$$p(v_i = 1|\mathbf{h}; \theta) = \mathcal{N}\left(\sum_{j=1}^{J} w_{ij} h_i + b_i, 1\right)$$

donde v_i sigue una distribución normal con media $\sum_{j=1}^{J} w_{ij} h_i + b_i$ y varianza uno.

RBM

- Esto último normalmente se hace para procesar variables contínuas de entrada que se convierten entonces a binarias y que posteriormente se procesan como binarias-binarias en capas superiores.
- Para aprender los modelos se tienen que ajustar los pesos w_{ij}, lo cual se puede hacer tomando el gradiente del logaritmo de la verosimilitud:

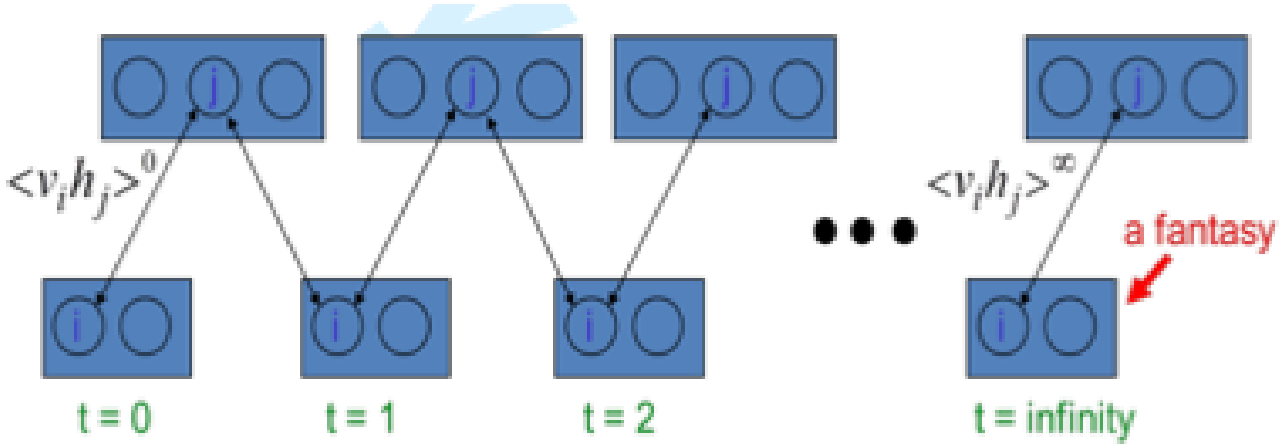
$$\Delta w_{ij} = E_{datos}(v_i h_j) - E_{modelo}(v_i h_j)$$

Donde $E_{datos}(v_i h_j)$ es valor esperado en el conjunto de entrenamiento y $E_{modelo}(v_i h_j)$ definido por el modelo.

RBM

- Este último no se puede calcular por lo que se hacen aproximaciones.
- Una de ellas es hacer muestreos (*Gibbs sampling*) siguiente los siguientes pasos:
  1. Inicializa $\mathbf{v}_0$ con los datos
  2. Muestrea $\mathbf{h}_0 \sim p(\mathbf{h}|\mathbf{v}_0)$
  3. Muestrea $\mathbf{v}_1 \sim p(\mathbf{v}|\mathbf{h}_0)$
  4. Muestrea $\mathbf{h}_1 \sim p(\mathbf{h}|\mathbf{v}_1)$
- Esto continua. $(\mathbf{v}_1, \mathbf{h}_1)$ es un estimado de $E_{modelo}(v_i h_j) = (\mathbf{v}_\infty, \mathbf{h}_\infty)$

RBM



Contrastive Divergence

1. El proceso de MCMC es lento y se han hecho aproximaciones
2. La más conocida se llama *Contrastive Divergence*
3. Se hacen dos aproximaciones: (i) reemplazar el promedio sobre todas las entradas (2^o término del log del error) por una sola muestra y (ii) correr el MCMC solo por k pasos

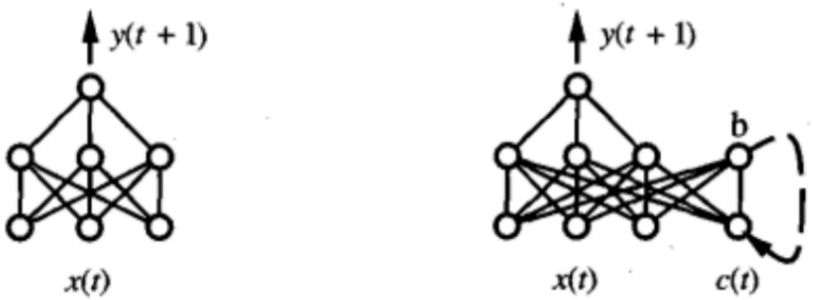
Otros Enfoques

Existen otros enfoques:

- LSTM
- Multi-task learning
- Multi-stream models
- Residual DNNs
- ...

LSTM

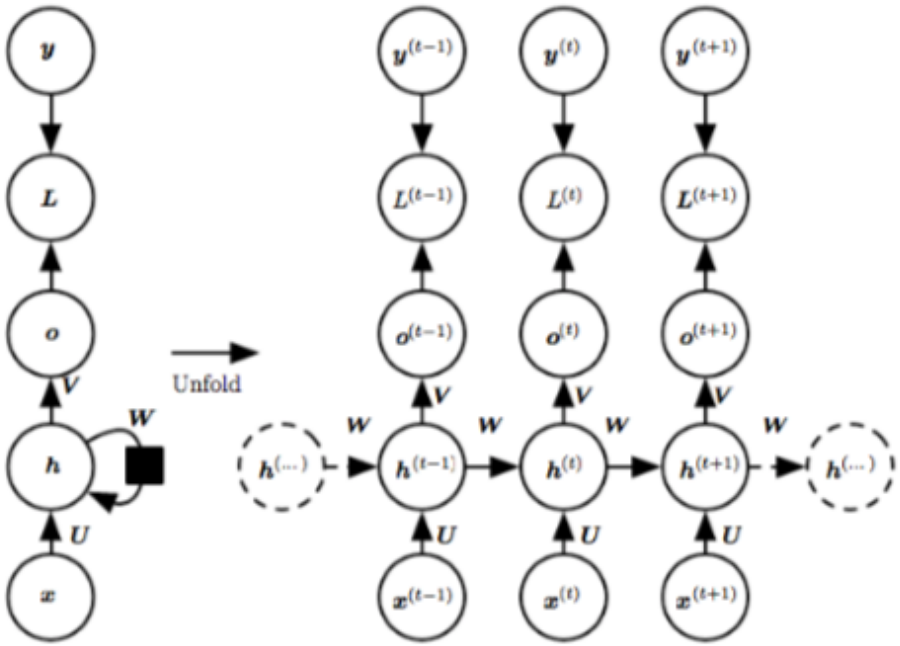
- Las redes recurrentes pueden recibir como entrada información de una de sus salidas



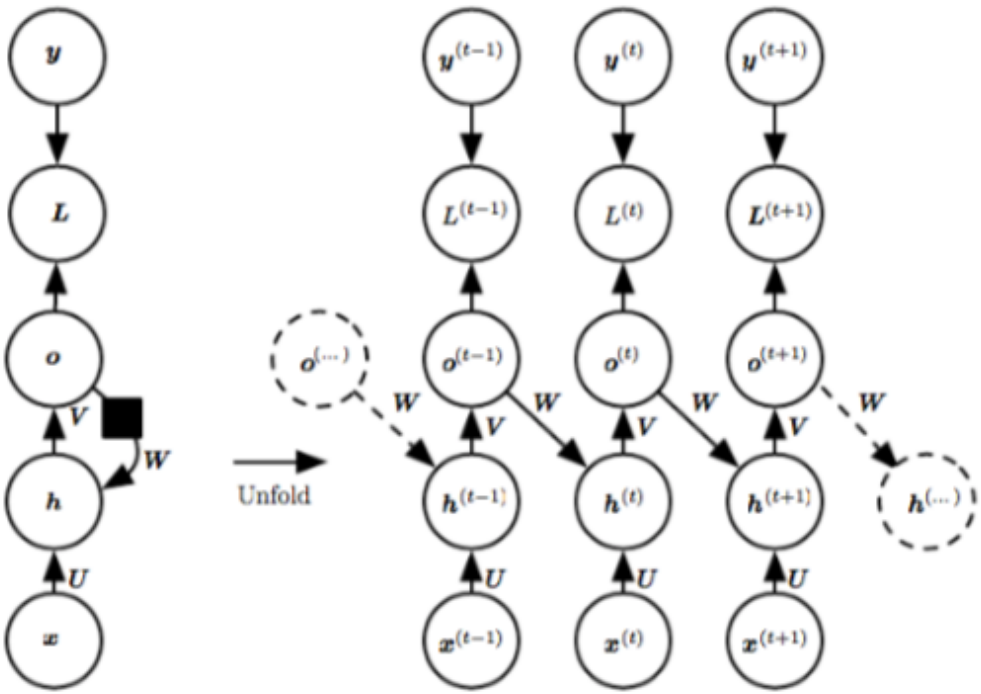
(a) Feedforward network (b) Recurrent network

- Existen de diferentes tipos y en general se desenrrollan (*unfolding*)

RNN-1

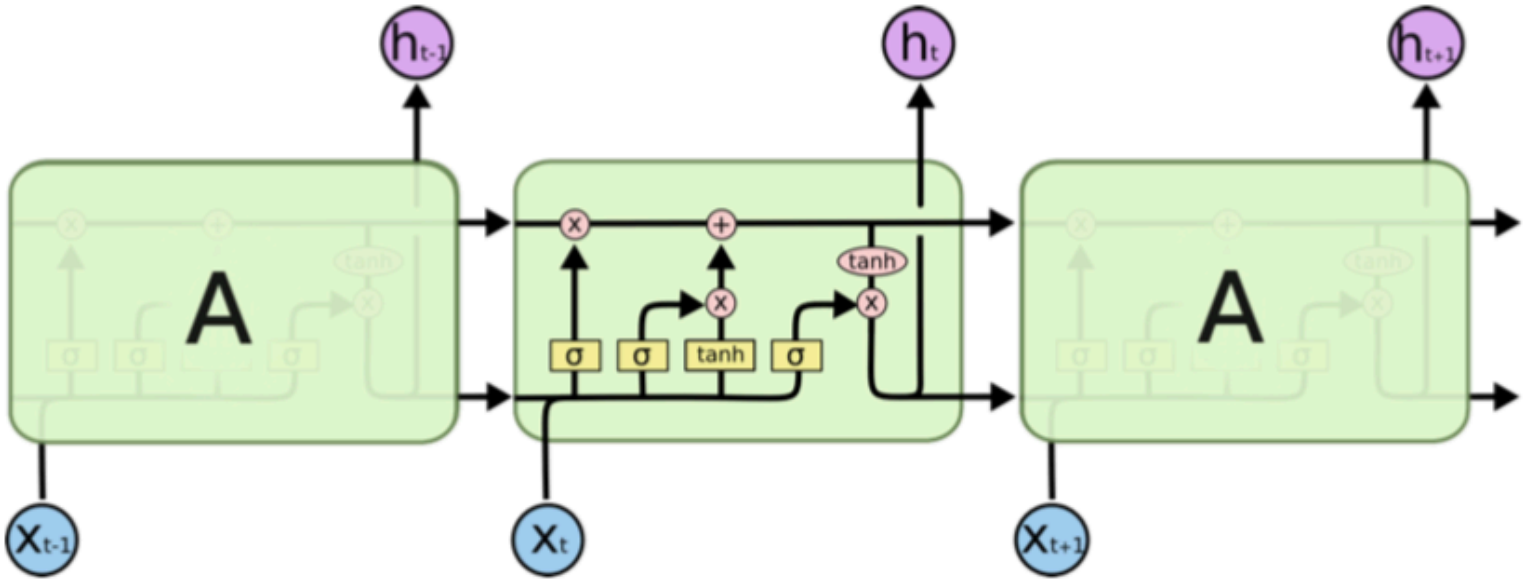


RNN-2

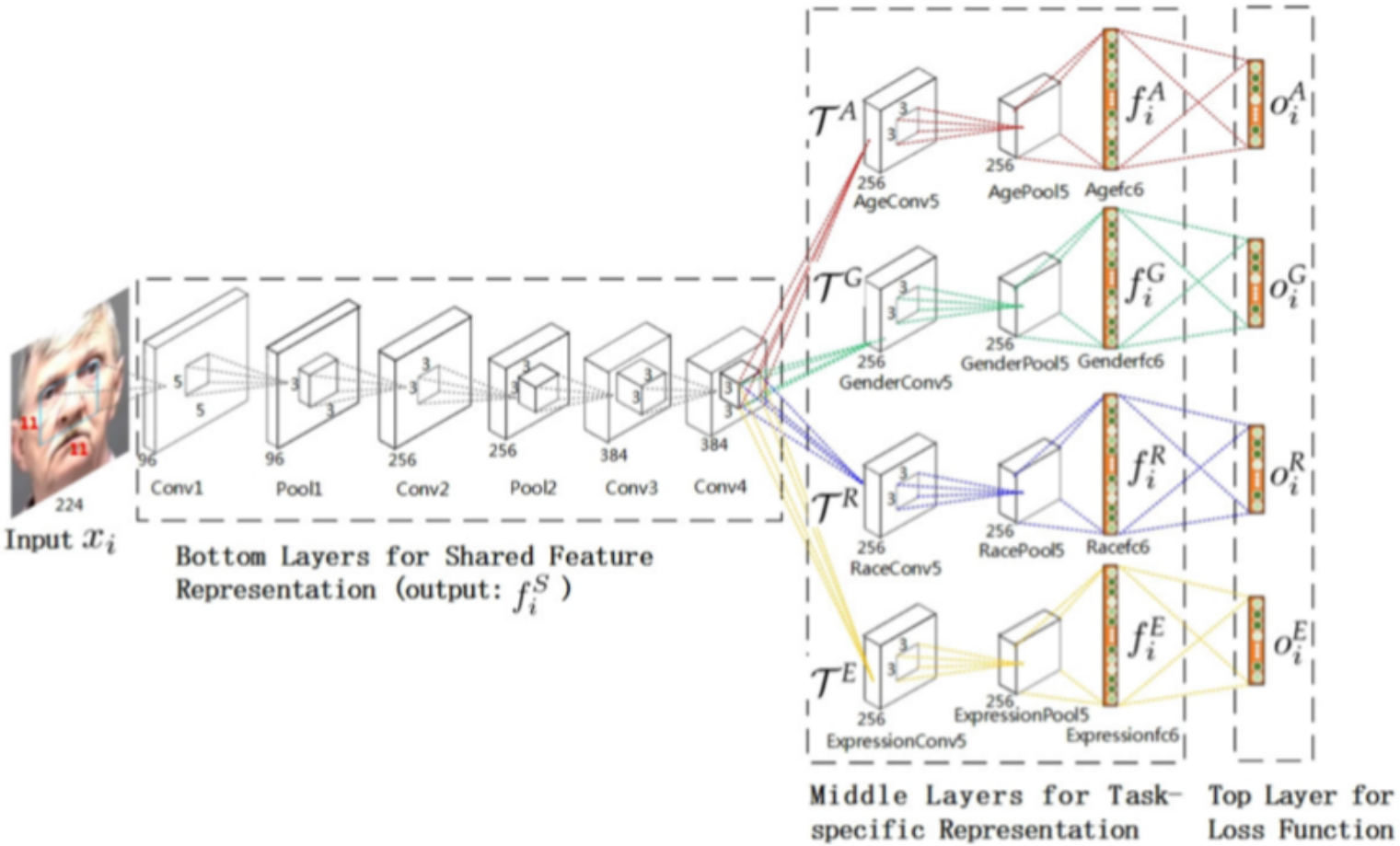


Long short-term memory (LSTM)

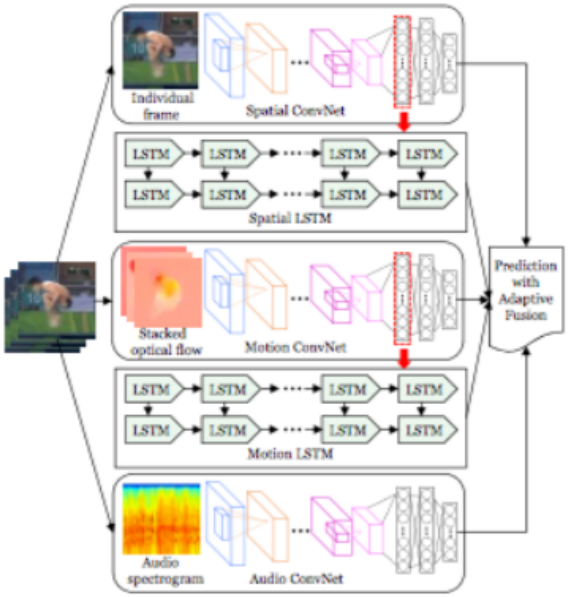
Se han usado para procesamiento de voz, traducción automática, procesamiento de lenguaje natural, etc.

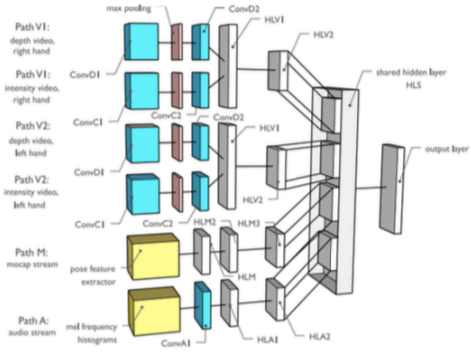


Multi-Task Learning

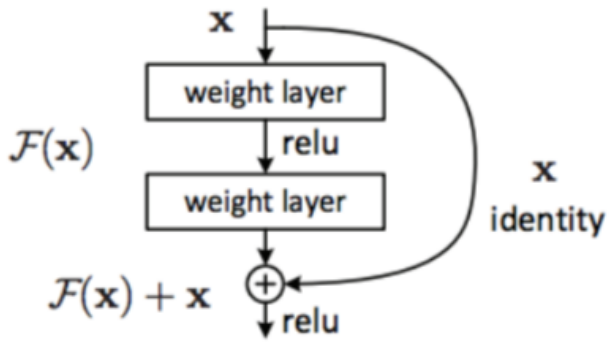


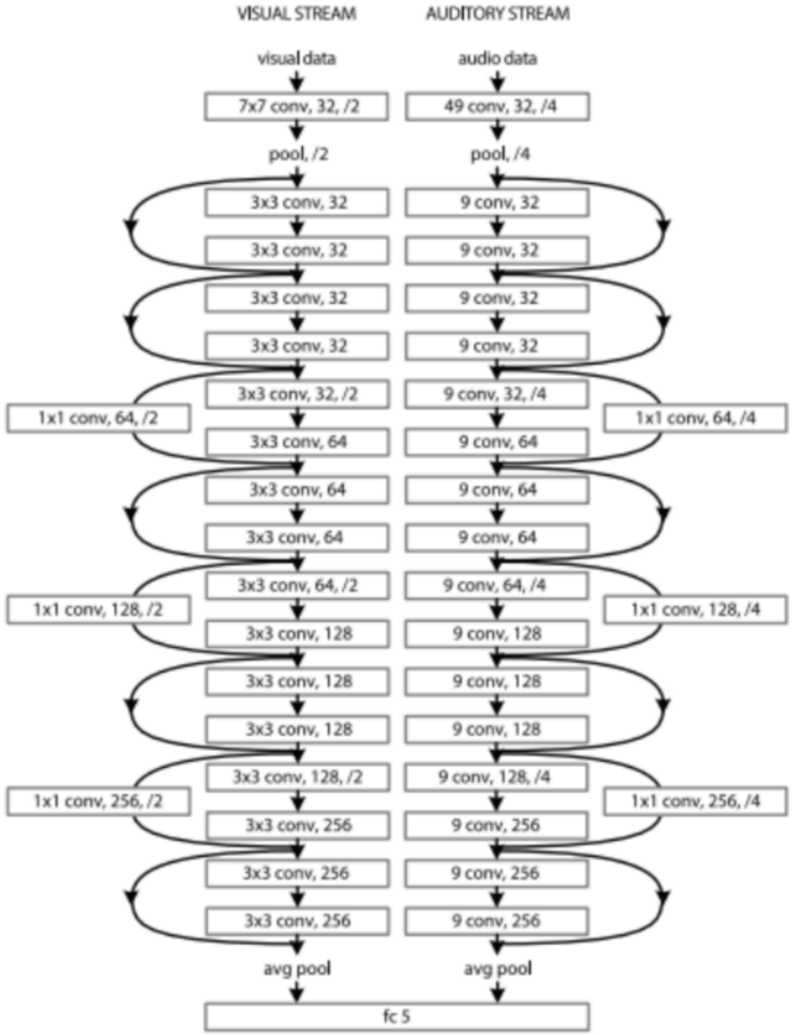
Multi-Stream Models





Residual Networks





Deep Learning

- Existen varios métodos de Deep Learning, pero no hay ninguno que domine a los demás.
- Se requieren algoritmos efectivos y escalables paralelos para poder procesar grandes cantidades de datos.
- Se requiere experiencia para poder definir los parámetros, número de capas, razón de aprendizaje, número de nodos por capa, etc.
- Se requiere más teoría
- Se desea que sean interpretables