Ingeniería del Conocimiento CIF-8458

Redes Neuronales Artificiales

Carlos Valle Vidal

Segundo Semestre 2023

Motivación

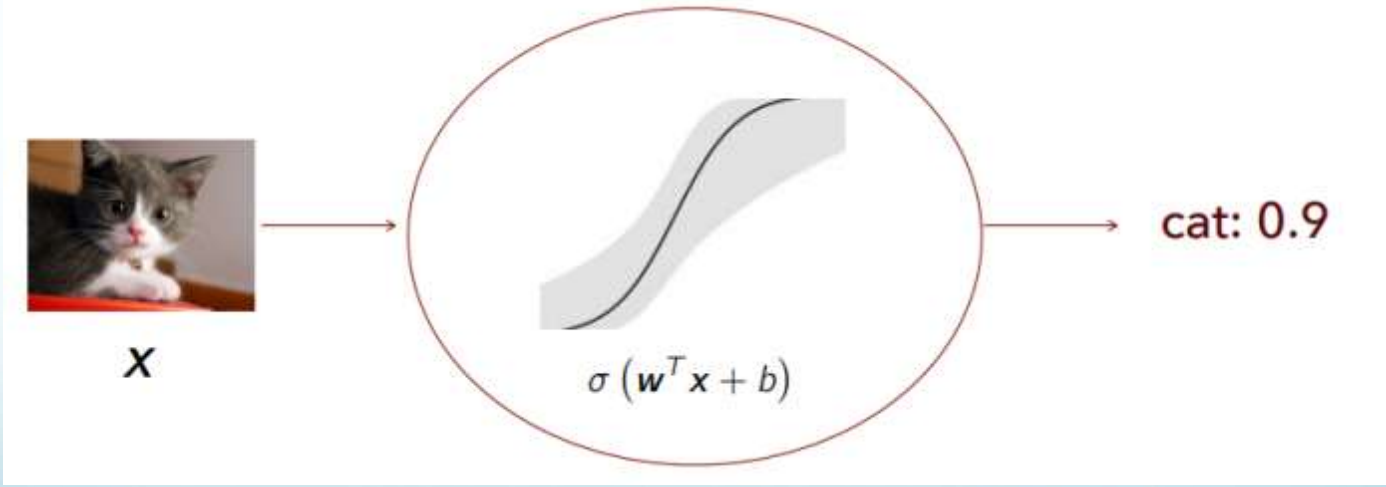
- Las redes Neuronales Artificiales (ANN) fueron inspiradas por científicos que intentan responder preguntas como:
- ¿Qué hace que el cerebro humano sea una máquina tan formidable en el procesamiento del pensamiento cognitivo?
- ¿Cuál es la naturaleza de esta cosa llamada inteligencia?
- ¿Cómo resuelven los problemas los humanos?
- Hay muchas teorías y modelos diferentes sobre cómo la mente trabaja.

Conexionismo

- Una de esas teorías, llamada conexionismo, usa análogos de neuronas y sus conexiones junto con los conceptos de funciones de activación y la capacidad de modificar esas conexiones para crear algoritmos de aprendizaje.
- Ahora las ANN se tratan de forma más abstracta, como una red de elementos que contienen información no lineal y que se encuentran interconectados.
- Problemas de reconocimiento de voz, reconocimiento de caracteres manuscritos, el reconocimiento facial son aplicaciones importantes de los ANN.

¿Por qué redes neuronales?

- Motivación esencial: extracción automática de características relevantes para resolver una tarea de aprendizaje.



¿Por qué redes neuronales? (2)

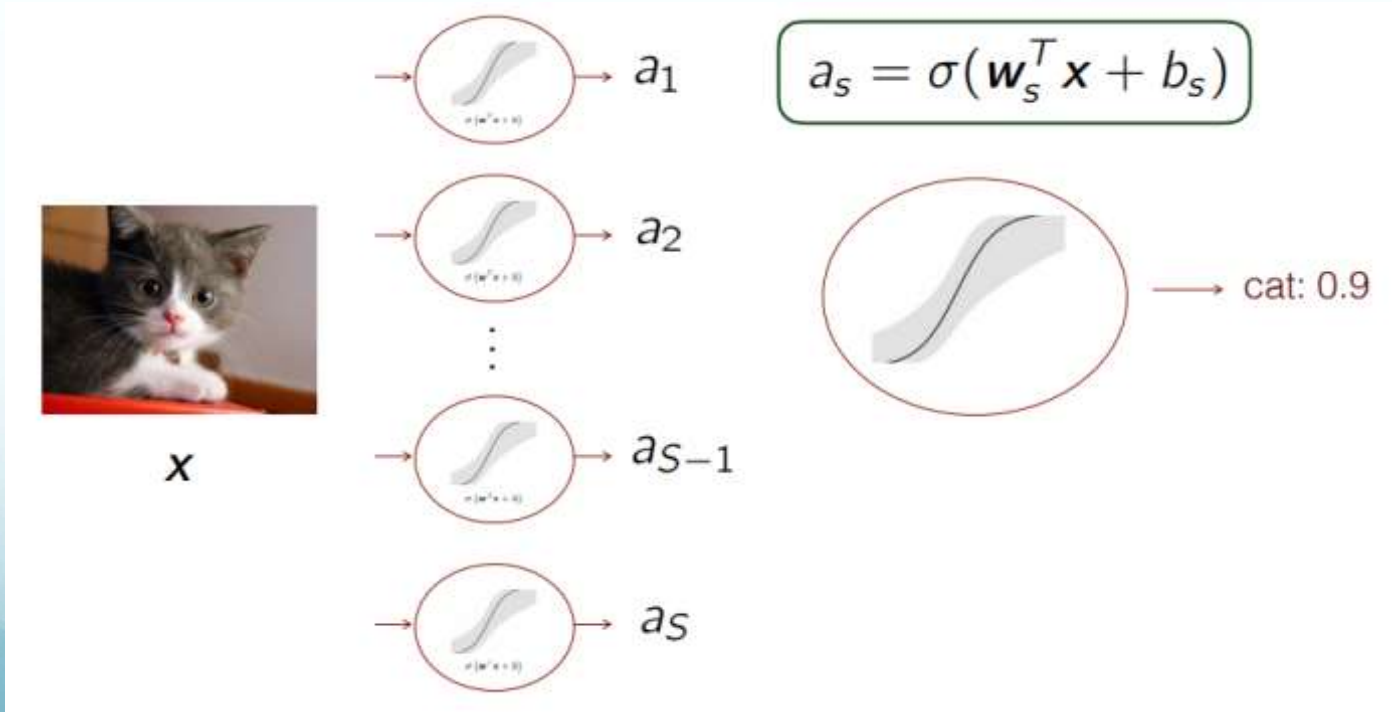
- Recordemos que un modelo de regresión logística $f(x)$ viene dado por:

$$f\ (x)\ = \sigma(w^T x\ +\ b)\ = \sigma\left(\sum_{i=1}^{I} w^{(i)} x^{(i)} + b\right)$$

- Por lo tanto, podemos decir que $f(x)$ depende de $x^{(1)}, x^{(2)}, \ldots, x^{(I)}$.
- En la práctica, no sabemos cuáles son los atributos relevantes para resolver el problema usando métodos lineales.

¿Por qué redes neuronales? (3)

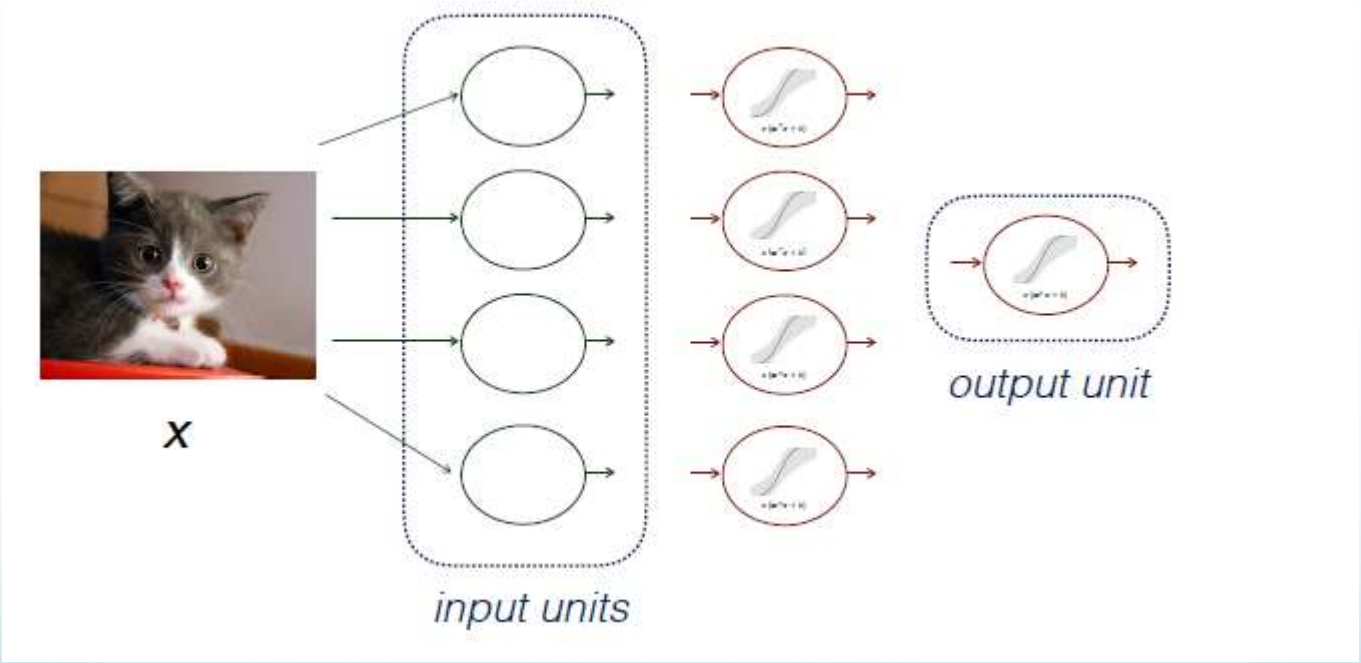
- Idea: La determinación de cada atributo (representación) es en sí mismo un problema de aprendizaje.



Neuronas y capas

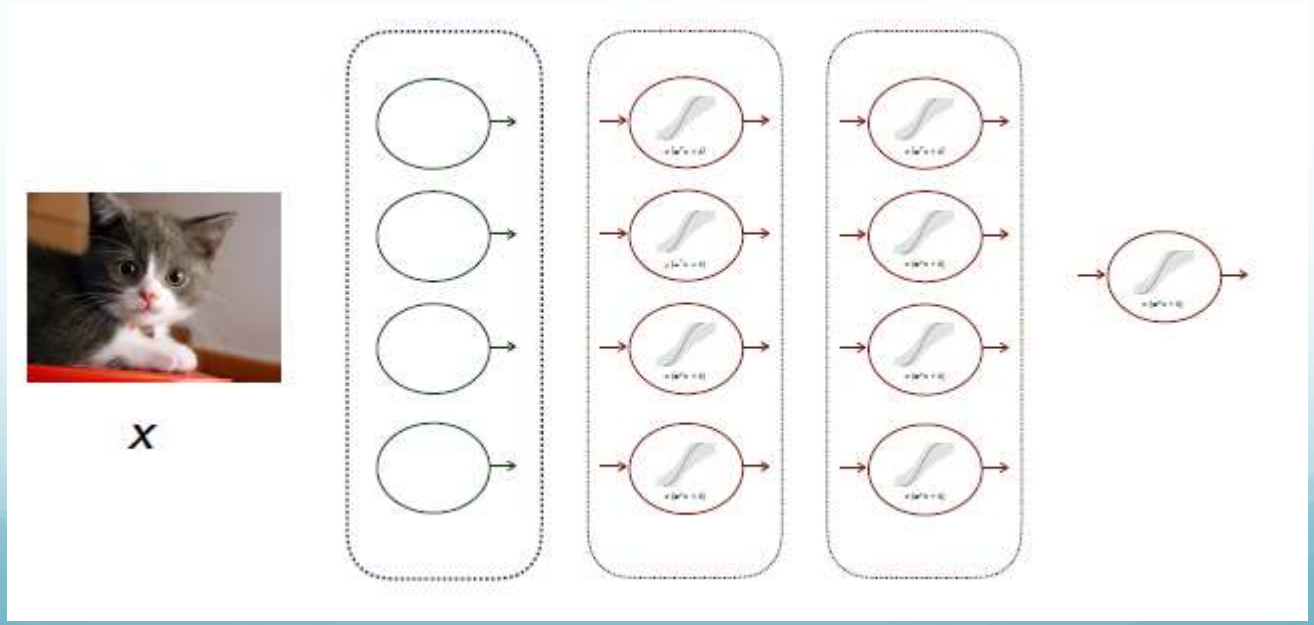
- Una **neurona** es cada una de las unidades de cálculo utilizadas en el modelo (para aprender un atributo).
- Cada característica original alimenta una unidad especializada (neurona) llamada **neurona de entrada**.
- La unidad que produce la salida final se llama la **neurona de salida**.
- Las **neuronas ocultas** se ubican entre la entrada y la salida. Ellas aprenden la representación subyacente.

Neuronas y capas (2)



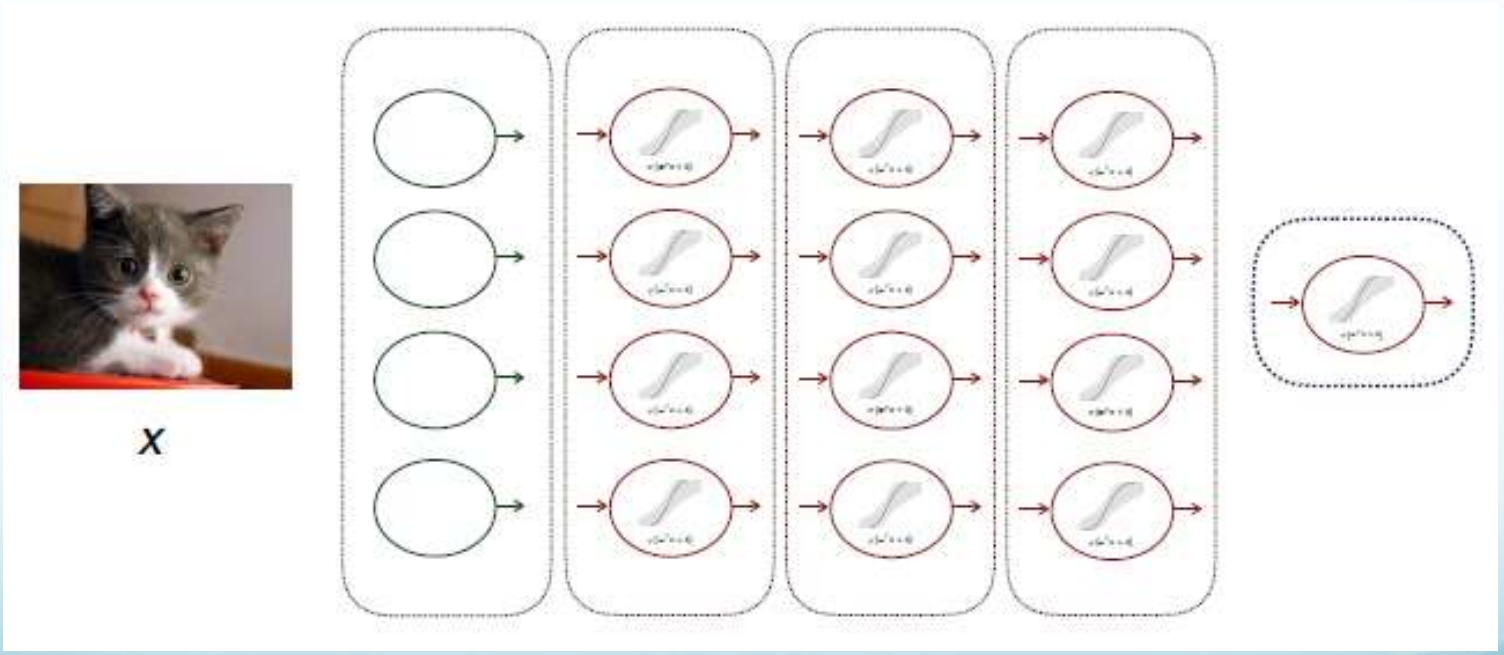
Neuronas y capas (3)

- Ampliando esta idea, es posible que tengamos que agregar más niveles de procesamiento, con el objetivo de aprender los atributos necesarios para aprender los atributos necesarios para aprender la salida.



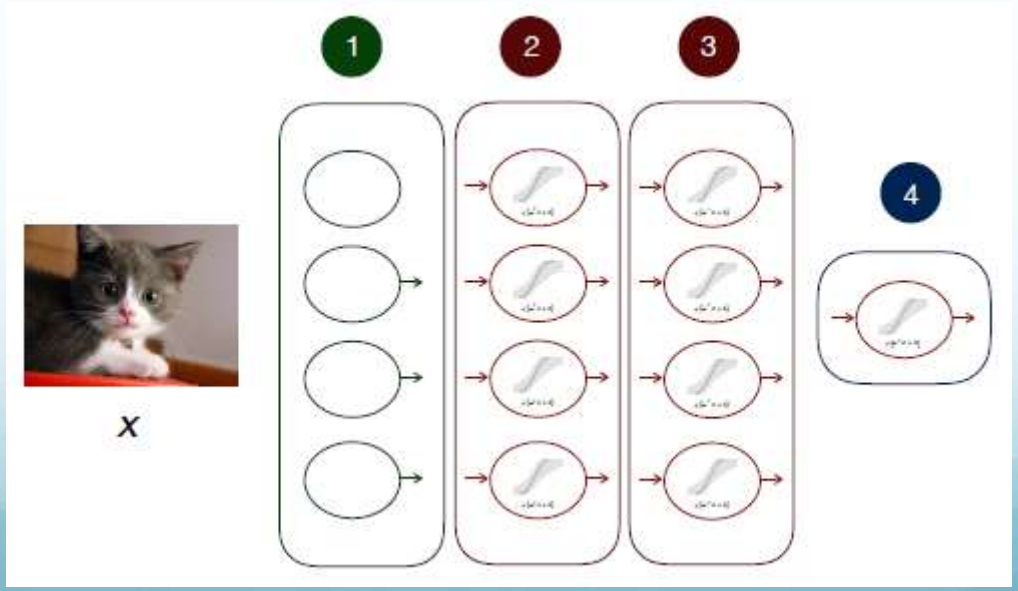
Neuronas y capas (4)

- Y así sucesivamente,



Neuronas y capas (5)

- Enumeramos las capas correlativamente desde la capa de entrada (que se enumera como 1) hasta la capa de salida.
- Por ejemplo en la imagen, la capa de entrada (1), la capa de salida (4) y las capas ocultas (2,3).



Neuronas y capas (6)

- Denotaremos por a_s^ℓ a la salida del nodo s de la capa ℓ.
- Sea a^ℓ el vector formado por las salidas de todas las neuronas de la capa ℓ, es decir

$$a^\ell = (a_1^\ell, a_2^\ell, \ldots, a_{S_\ell}^\ell)$$

- Donde S_ℓ es el número de neuronas de la capa ℓ.
- Sea L el número de capas, por lo tanto, a^1 es la capa de entrada y a^L es la capa de salida.

Neuronas y capas (7)

- En problemas de regresión necesitamos un solo nodo en la capa de salida.
- En problemas de clasificación binaria, también nos basta con un nodo en la capa de salida para modelar la probabilidad de que el ejemplo sea de la clase 1.
- Esto es, el nodo de salida modela:

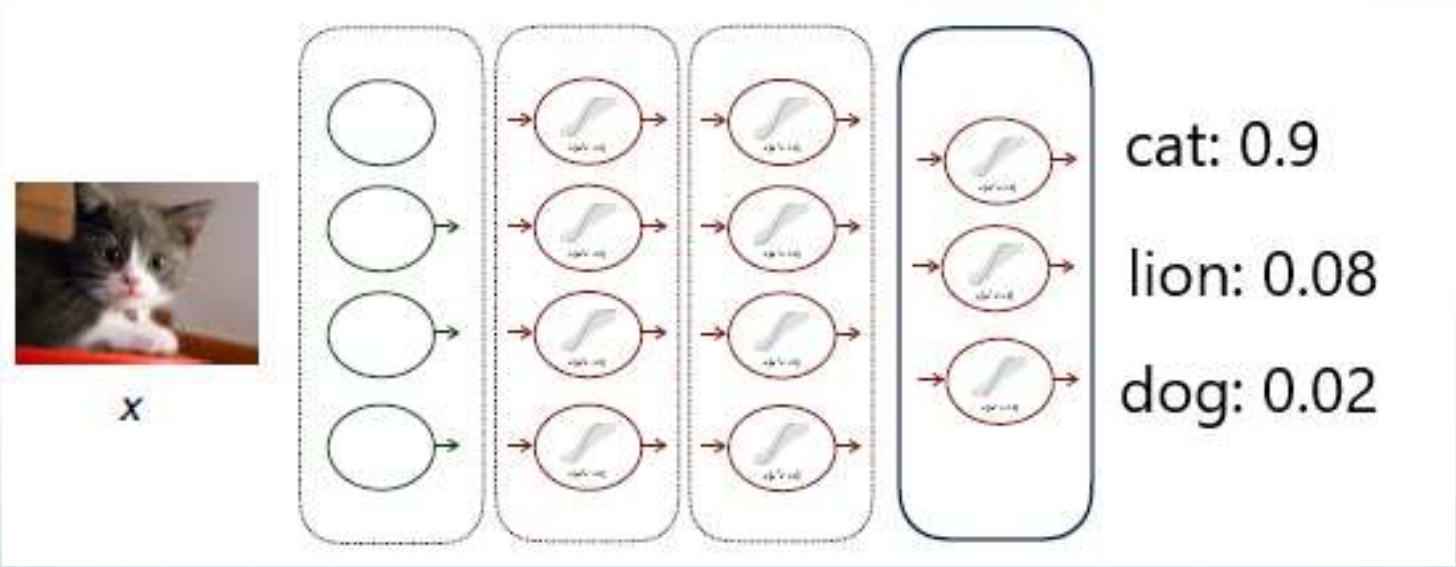
$$a_1^{(L)} = P(y_m = 1|x_m)$$

Neuronas y capas (8)

- En problemas de clasificación con más de dos clases
- Necesitamos que cada neurona de salida modele la probabilidad de que el ejemplo pertenezca a dicha clase
- $a_s^{(L)} = P(y_m = s|x_m)$
- Por lo tanto si tenemos K clases necesitaremos K neuronas de salida.
- Para poder medir correctamente el error de la red, el target debemos escribirlo como *one-hot-vector* de largo K que contendrá un 1 en la casilla de la clase a la que corresponde y un 0 en el resto de las casillas.
- Por ejemplo, si $K = 5$, codificaremos $y_2 = 3$ como $y_2 = (0,0,1,0,0)$. Es decir, ponemos un uno en la tercera componente.

Neuronas y capas (9)

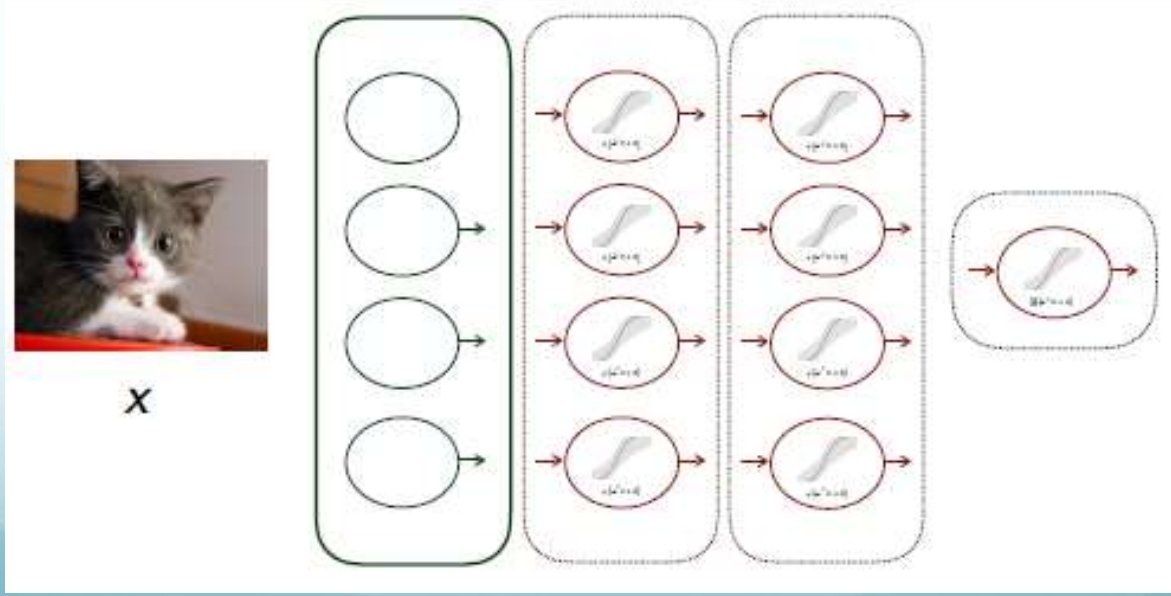
- Para esto necesitaremos agregar una capa especial que asegure que las probabilidades sumen 1.



Capas = Transformaciones

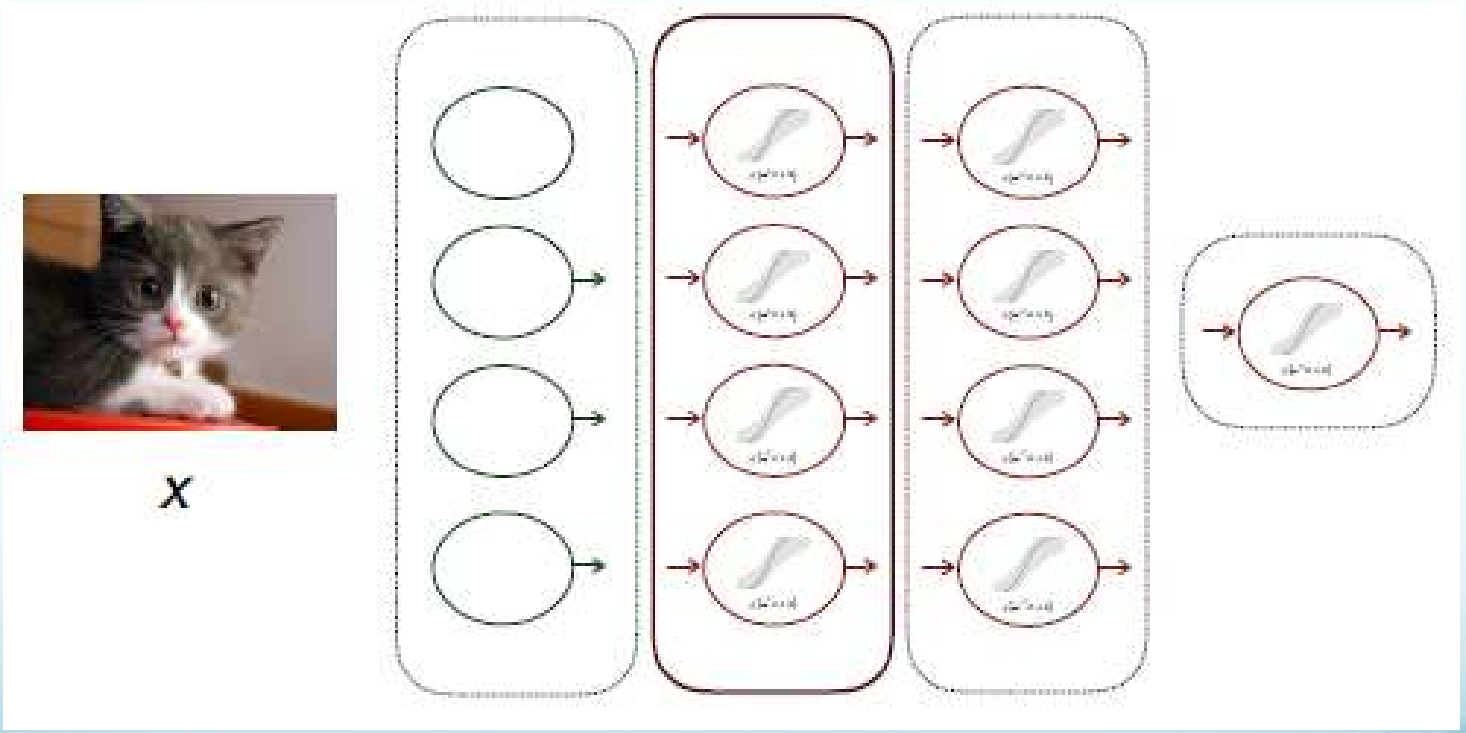
- Matemáticamente, cada nivel calcula una transformación de la representación obtenida en el nivel (o capa) anterior.

$$a^{(1)} = x = (x^{(1)}, \ldots, x^{(I)})$$



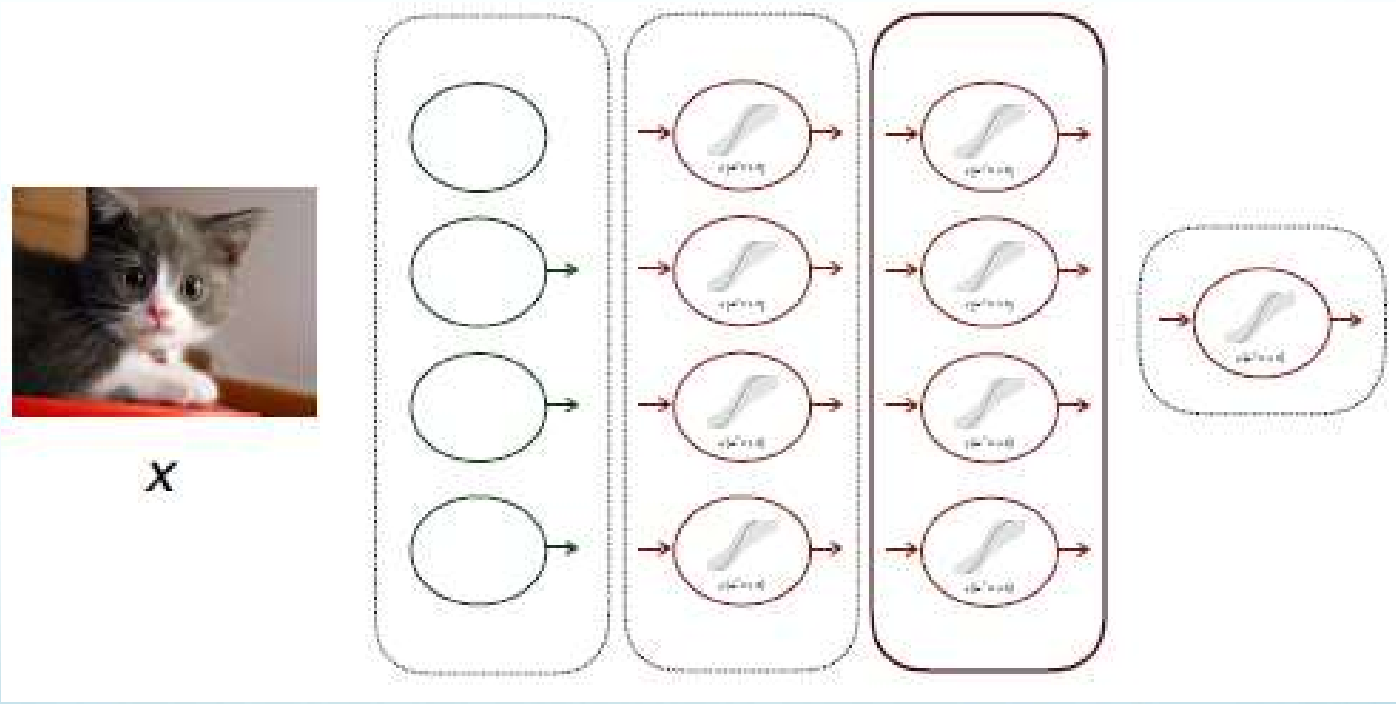
Capas = Transformaciones (2)

- $a^{(2)} = H^{(1)}\left(a^{(1)}\right)$



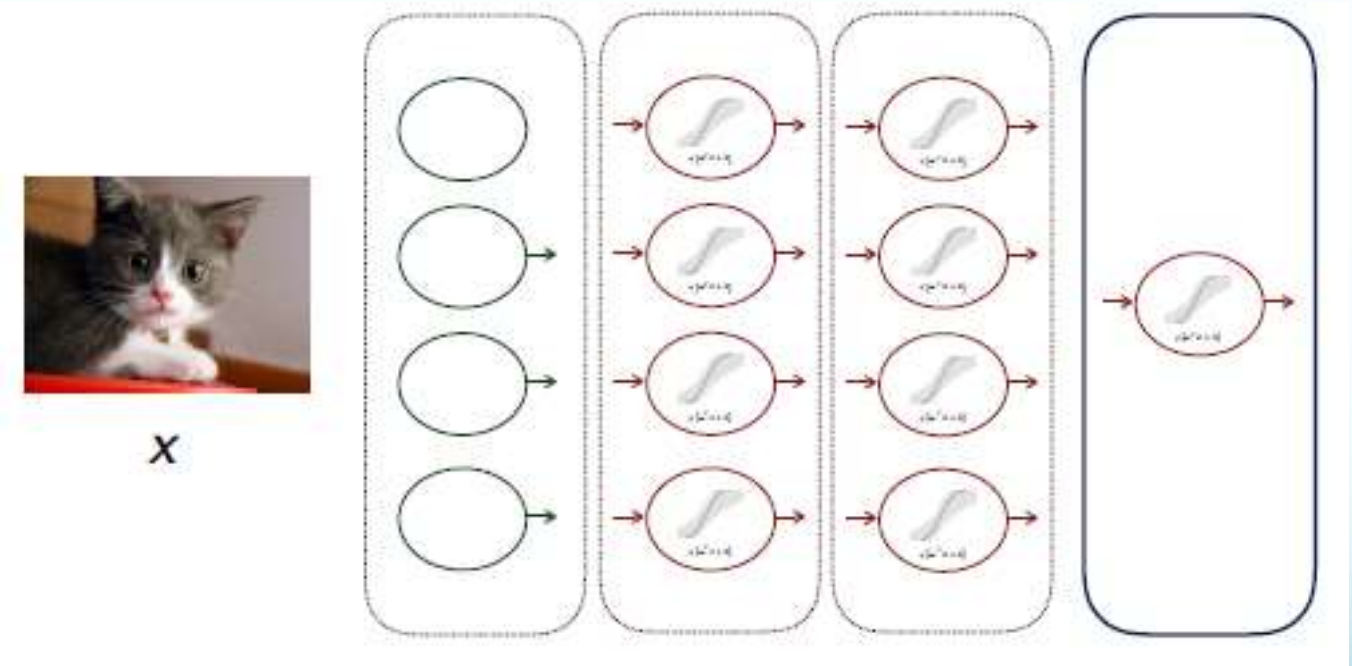
Capas = Transformaciones (3)

- $a^{(3)} = H^{(2)}\left(a^{(2)}\right)$



Capas = Transformaciones (4)

- $a^{(4)} = H^{(3)}\left(a^{(3)}\right)$

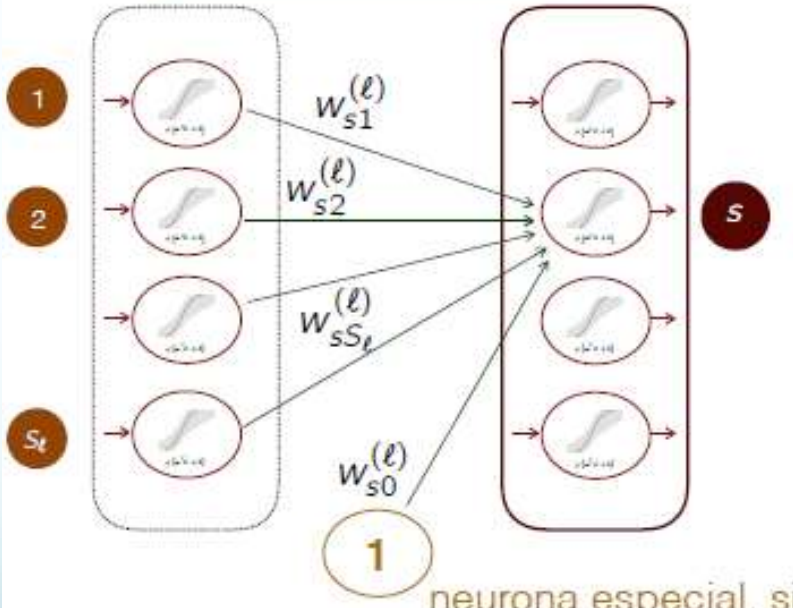


Transformaciones: Combinaciones lineales + combinaciones no lineales

- Cada neurona combina linealmente los atributos generados en el nivel anterior y luego transforma la señal total calculada utilizando una función no lineal llamada **función de activación**.
- Para combinar linealmente los atributos necesitamos asociarles a cada uno un peso, y agregar el coeficiente libre de cada capa que llamaremos **sesgo** o **bias**.
- Llamaremos w_{st}^{ℓ} al peso que conecta el nodo t de la capa ℓ con el nodo s de la capa $\ell + 1$.

Transformaciones: Combinaciones lineales + combinaciones no lineales (2)

$$a^{(\ell+1)} = H^{(\ell)}(a^{(\ell)})$$



$$\boldsymbol{w}_s^{(\ell)} = (w_{s1}^{(\ell)}, w_{s2}^{(\ell)}, \ldots, w_{sS_\ell}^{(\ell)})^\top \in \mathbb{R}^{S_\ell}$$

$$a_s^{(\ell+1)} = \sigma(\boldsymbol{w}_s^{(\ell)\top} \boldsymbol{a}^{(\ell)} + b_s^{(\ell)})$$
$$= \sigma\left(\sum_{t=1}^{S_\ell} w_{st}^{(\ell)} a_t^{(\ell)} + w_{s0}^{(\ell)}\right)$$

Transformaciones: Combinaciones lineales + combinaciones no lineales (3)

- Como podemos ver, la salida del nodo s es la transformación no lineal de la combinación lineal de todas las salidas de los nodos de la capa anterior (más el sesgo).

Evaluando un ejemplo (Forward Pass)

- Entrada: Vector x (con I atributos).

1. $a^{(1)} = x$
2. for $\ell = 1$ to L do
   1. $a_s^{(\ell+1)} = \sigma\left(\sum_{t=1}^{S_t} w_{st}^{(\ell)} a_t^{(\ell)} + w_{s0}^{(\ell)}\right)$
3. end for

- La información se propaga desde los nodos de entrada a los nodos de salida.

¿Cuántas neuronas?¿Cuántas capas?

- Teorema de Aproximación Universal, Cybenko (1989):
  - Para toda tarea de aprendizaje podemos aprender una función continua f usando una red neuronal de 3 capas con error menor o igual que $\varepsilon, \forall x$.

$$|f(x) - f_{ANN}(x)| \leq \varepsilon$$

- En problemas reales, los valores óptimos dependen en gran medida del problema, específicamente depende de los atributos y el tamaño del conjunto de datos.

Parámetros vs Hiperparámetros

- Parámetros del modelo: su valor se determina entrenando al modelo, es decir, del error observado en los ejemplos de entrenamiento.
- Ejemplos:
  - Pesos de la red.
  - Sesgos de la red.

Parámetros vs Hiperparámetros (2)

- Hiper-parámetros del modelo: En general, su valor no puede ser determinado a partir de los datos de entrenamiento.
- Se pueden estimar usando técnicas de validación.
- Ejemplos:
  - Número de capas.
  - Número de neuronas en cada capa.

Deep versus Shallow

- Antes de 2006 el modelo preferido tenía 1 capa oculta.
- Hoy en día, el reciclaje de atributos es la idea más popular.
  - Cada neurona de un nivel puede utilizar todos los atributos obtenidos por la capa anterior.
  - Cada atributo generado por una neurona en una capa puede ser utilizado por todas las neuronas en la siguiente capa.
  - Cuanto más profunda es la red, mayor es el posible reciclaje de atributos, es decir, es posible obtener representaciones más compactas.
  - Capas como niveles de abstracción para resolver un problema: un mayor número de capas de procesamiento permiten construir atributos de mayor complejidad a partir de atributos más simples.

Funciones de Activación

- Cada neurona en la red aprende una transformación (combinación lineal + no linealidad)

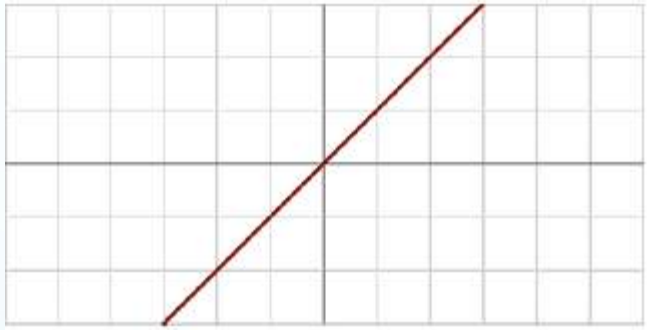
$$a_s^{(\ell+1)} = \sigma\left(\sum_{t=1}^{S_t} w_{st}^{(\ell)} a_t^{(\ell)} + w_{s0}^{(\ell)}\right)$$

- Donde σ es la función de activación.
- Es posible modelar / elegir esta última con diferentes criterios.

Función de Activación Lineal

- Esta función no modifica la entrada:

$$\sigma(\varepsilon) = \varepsilon$$



- Esta función es comúnmente usada en la capa de salida para problemas de regresión.

Función de Activación Sigmoidal

- Esta función sigmoidal o logística transforma la entrada en una salida en el rango 0,1.

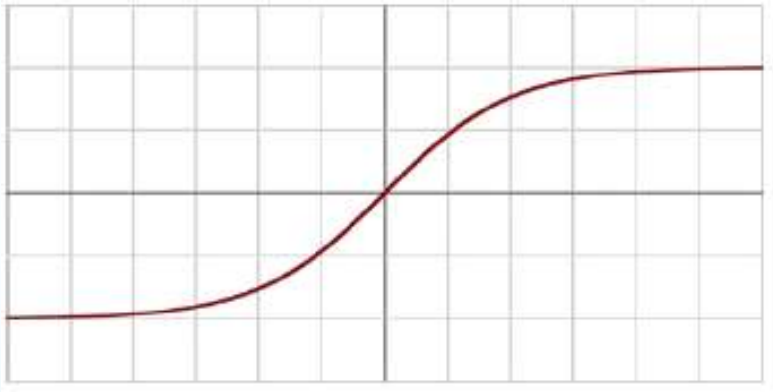
$$\sigma(\varepsilon) = \frac{1}{1 + e^{-\varepsilon}}$$

- Como ya hemos visto esta función modela $P(y = 1|x)$
- Por lo que es muy usada en la neurona de salida en un problema de clasificación binaria.

Función de Activación tanh

- La función tangente hiperbólica tiene la forma:

$$\sigma(\varepsilon) = \frac{e^{-2\varepsilon} - 1}{e^{-2\varepsilon} + 1}$$

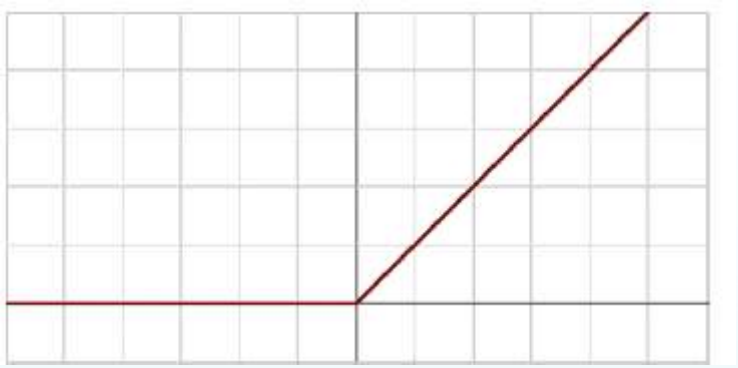


- Esta función transforma la entrada en una salida en el rango -1,1.

Función de Activación ReLu

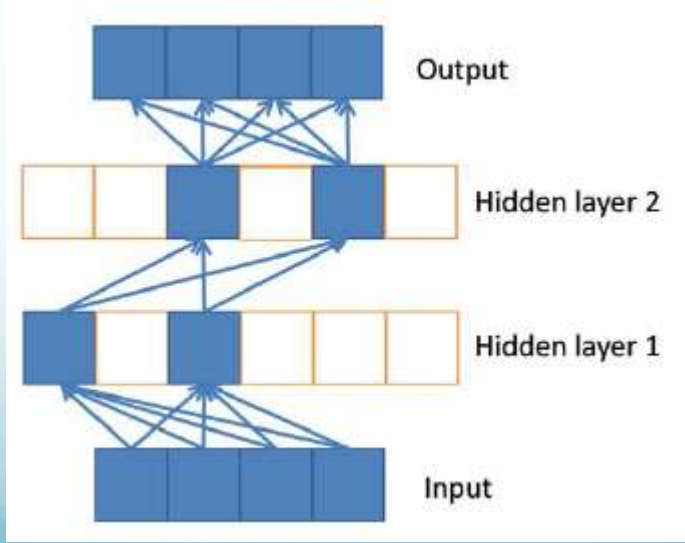
- Esta función es un rectificador lineal:

$$\sigma(\varepsilon) = \begin{cases} \varepsilon & \varepsilon \geq 0 \\ 0 & \varepsilon < 0 \end{cases}$$



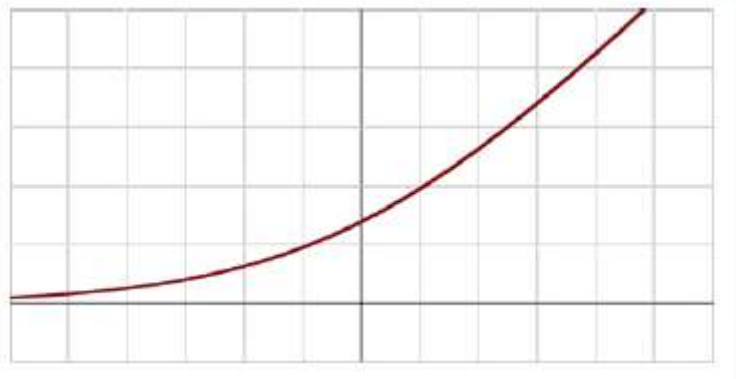
Función de Activación ReLu

- Cuando se procesa una instancia x, se activa solo un subconjunto de las unidades de la red.
- Y la respuesta es lineal en el subconjunto activo neuronas.
- Podemos ver esto como un árbol de clasificación con un exponencialmente grande número de hojas y predictores lineales sobre ellas.



Función de Activación Softplus

- $\sigma(\varepsilon) = \log(1 + e^{-\varepsilon})$



Funciones de Activación

- Hornik en 1991 establece que para mantener la propiedad de aproximación universal de las redes neuronales, las funciones de activación deben ser:
  - No constantes
  - Acotadas
  - Continuas
- Kurt Hornik. Approximation Capabilities of Multilayer Feedforward Networks. Neural Networks, Vol. 4, pp. 251-257. 1991
- En la práctica, la función de activación puede variar de una capa a otra.

Funciones de Activación (2)

- Investigaciones recientes establecen que es posible utilizar ciertas funciones no acotadas y aún mantener la propiedad de aproximación universal.
- S. Sonuda, N. Murata. Neural network with unbounded activation functions is universal approximator, 2015.
- En particular, la propiedad se mantiene para los 2 más populares funciones de transferencia: ReLu y Softplus.

Capa Softmax

- Como ya discutimos, en problemas de clasificación con múltiples clases, necesitamos que cada neurona modele la probabilidad de que el ejemplo pertenezca una clase, y por lo tanto, necesitamos que las salidas de todas las neuronas de la capa de salida sumen 1.
- Para hacerlo aplicaremos la función softmax, que es equivalente a aplicar la función logística y luego normalizar.
- Por esto, la función softmax puede ser vista como una capa aparte que toma las salidas y las normaliza para que sumen uno.

$$a_s^{(L-1)} = \sigma\left(\sum_{t=1}^{S_{L-2}} w_{st}^{(L-2)} a_t^{(L-2)} + w_{s0}^{(L-2)}\right), s = 1,2,\dots,S_{L-1}$$

$$a_s^{(L)} = \frac{a_s^{(L-1)}}{\sum_{t=1}^{S_{L-1}} a_t^{(L-1)}}, s = 1,2,\dots,S_{L-1} = S_L$$

¿Cómo ajustar los pesos?

- Necesitamos un método de optimización que nos permita determinar los mejores valores para los pesos y sesgos que forman parte de la red neuronal.
- Una vez más, un criterio en quedarnos con los pesos que minimicen el error de la red sobre todo el conjunto de entrenamiento:
- $min_W R_{emp}(f_W) = min_W \frac{1}{M} \sum_{m=1}^{M} \ell(f_{ANN}(x_m), y_m)$

¿Cómo escoger ℓ?

- Recordemos que si tenemos un problema de clasificación binaria y usamos una salida sigmoidal, la salida de la red es interpretable directamente como:

$$f(x) = p(y = c_1|x) = p(y = 1|x)$$

- Como es un problema de dos clases, el modelo probabilístico subyacente es:

$$\begin{aligned} p(y|x) &= p(y = 1|x)^y + p(y = -1|x)^{1-y} \\ &= p(y = 1|x)^y + (1 - p(y = 1|x))^{1-y} \end{aligned}$$

- La log-verosimilitud del modelo probabilístico es:

$$\mathrm{L(S)} = \ln \prod_m p(y_m|x_m) = \sum_m \ln p(y_m|x_m)$$

$$= \frac{1}{M}\left(\sum_m (y_m \ln f(x_m) + (1 - y_m) \ln(1 - f(x_m)))\right)$$

¿Cómo escoger ℓ? (2)

- Maximizar $L(S)$ es equivalente a minimizar $-L(S)$.
- Por lo tanto, para problemas de clasificación binaria, la función de pérdida a minimizar será:
- $\mathrm{E}(f) = -\sum_m (y_m \ln f(x_m) + (1 - y_m) \ln(1 - f(x_m)))$
- Esta se conoce como entropía cruzada o cross-entropy

¿Cómo escoger ℓ? (3)

- Para problemas de clasificación con múltiples clases, la entropía cruzada se extiende como:
- $\mathrm{E}(f) = -\sum_m \sum_k (y_{mk} \ln \hat{y}_{mk} + (1 - y_{mk}) \ln(1 - \hat{y}_{mk}))$
- Donde $\hat{y}_{mk}$ es el valor de la neurona k al evaluar x_m en la red neuronal (forward-pass).
- Es decir,

$$\hat{y}_{mk} = f(x_m)_k$$

- Y y_{mk} es la casilla k del one-hot vector que representa y_m.

¿Cómo escoger ℓ? (4)

- Para problemas de regresión usamos el error cuadrático:

$$\ell(f(x_m), y_m) = (y_m - f(x_m))^2$$

Backpropagation

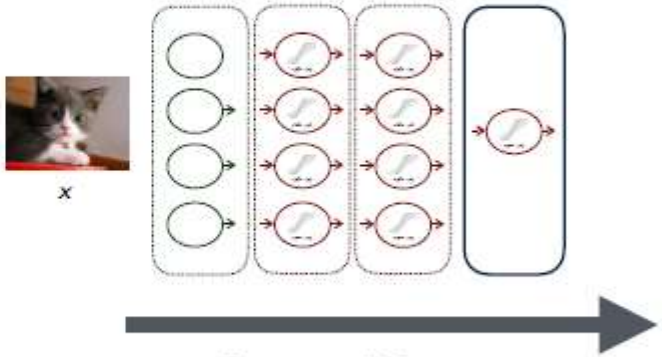
Algorithm Back-propagation algorithm

1: Initialize the network weights
2: **while** stop criteria not met **do**
3: **for** each example (x_m, y_m) **do**
4: Compute forward pass(x_m, y_m)
5: Compute the error $E = E(\mathbf{x}_m, y_m)$
6: Compute backward pass(E)
7: **end for**
8: **end while**

Forward pass

Forward Pass

$a^{(1)} = x$;
for $\ell = 1, \ldots, L-1$ **do**
  $a^{(\ell+1)} = \sigma(W^{(\ell)\top} a^{(\ell)} + w_0^{(\ell)})$;
end
return $a^{(L)}$



Midiendo el error

- En problemas de regresión:

$$E = E(\mathbf{x}_m, y_m) = \frac{1}{2}\sum_{k=1}^{K}(a_k^{(L)} - y_k)^2$$

$$= E(\mathbf{x}_m, y_m) = \frac{1}{2}\sum_{k=1}^{K}(f_{ANN}(\mathbf{x})_k - y_k)^2$$

- Así

$$\frac{\partial E}{\partial a_S^{(L)}} = (a_s^{(L)} - y_s) = (f_{ANN}(\mathbf{x})_S - y_S)$$

Midiendo el error (2)

- En problemas de clasificación:

$$\frac{\partial E}{\partial a_S^{(L)}} = \frac{(y_s - a_s^{(L)})}{a_s^{(L)}(1 - a_s^{(L)})}$$

- Así

$$\frac{\partial E}{\partial a_S^{(L)}} = \frac{(y_s - a_s^{(L)})}{a_s^{(L)}(1 - a_s^{(L)})}$$

¿Cómo implementar backward pass?

- Usando gradiente descendente, para cada peso o bias:

$$w \leftarrow w - \eta \frac{\delta E}{\delta w}$$

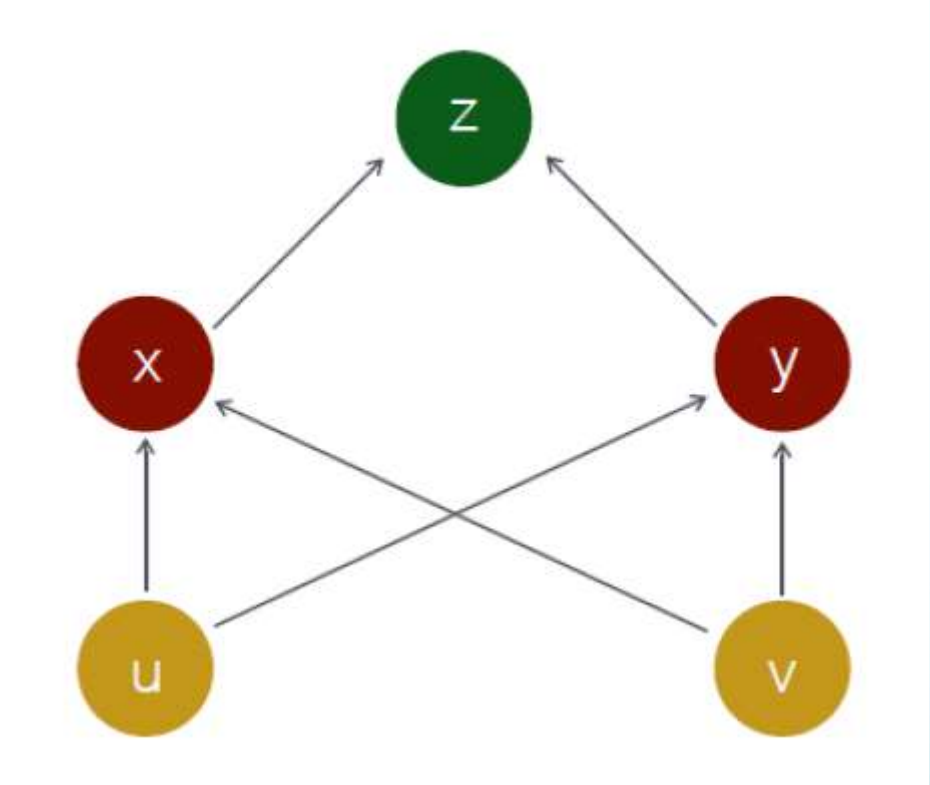
- Así, para cada capa ℓ:

$$w_{st}^{(\ell)} \leftarrow w_{st}^{(\ell)} - \eta \frac{\delta E}{\delta w_{st}^{(\ell)}}$$

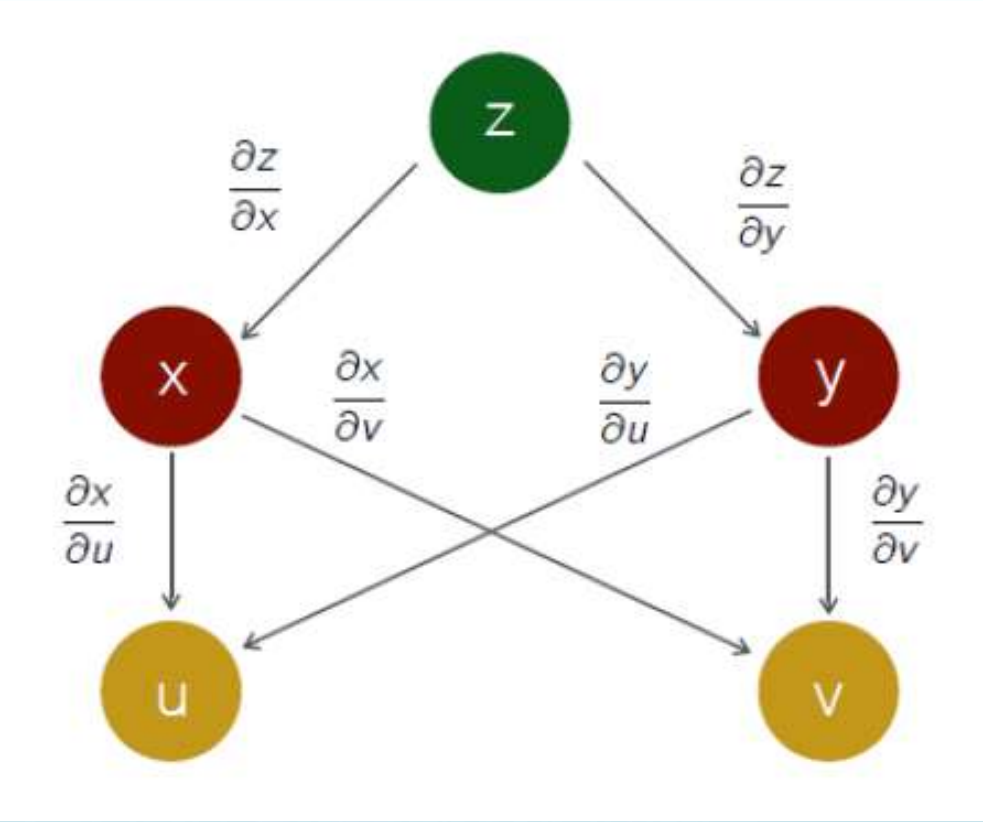
$$w_{s0}^{(\ell)} \leftarrow w_{s0}^{(\ell)} - \eta \frac{\delta E}{\delta w_{s0}^{(\ell)}}$$

- El hiperparámetro η se conoce como la tasa de aprendizaje o **learning rate**.

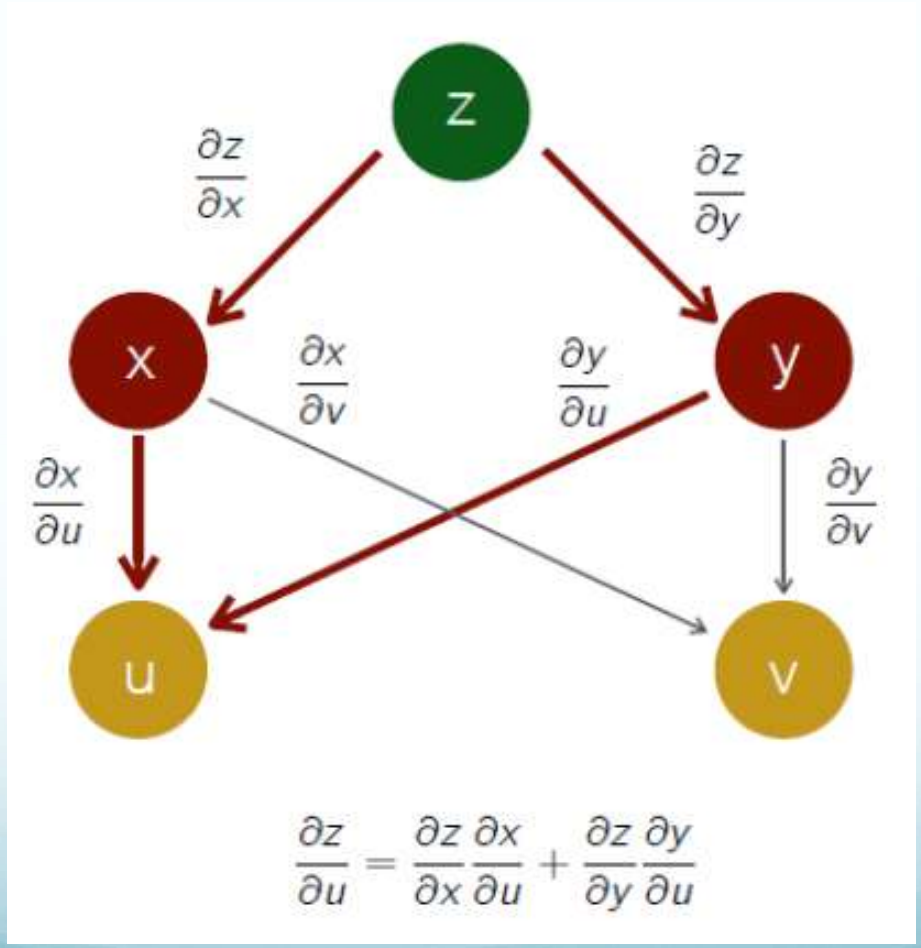
Regla de la cadena



Regla de la cadena (2)



Regla de la cadena (3)



Regla de la cadena (4)

$$\delta_s^\ell = \sum_k \delta_k^{\ell+1} w_{ik}^{\ell+1} \sigma'(z_j^\ell)$$

$$\delta^\ell = W^{\ell+1} \delta^{\ell+1} \odot \sigma'(z^\ell)$$

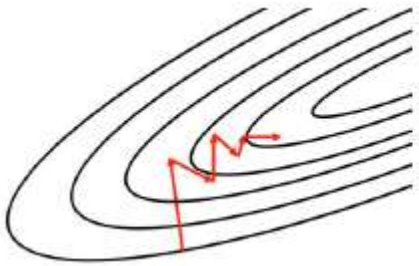
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial w_{ij}}$$

Optimización vía gradiente

- La idea clave de Backpropagation es modificar iterativamente los pesos de la red neuronal siguiendo la dirección indicada por el gradiente de la función de error elegida (para la tarea en cuestión).
- Esta estrategia de optimización se denomina **gradiente descendente** (GD).

Optimización vía gradiente (2)

- La mayoría de los algoritmos de aprendizaje profundo implican la optimización de algún tipo.



Problem $\min_{\boldsymbol{w}} E(\boldsymbol{w})$

Algorithm

1 **for** $t = 1, \ldots, T$ **do**
2 $\quad \boldsymbol{w}^{(t)} \leftarrow \boldsymbol{w}^{(t-1)} - \eta \frac{\delta E}{\delta \boldsymbol{w}}$;
3 **end**

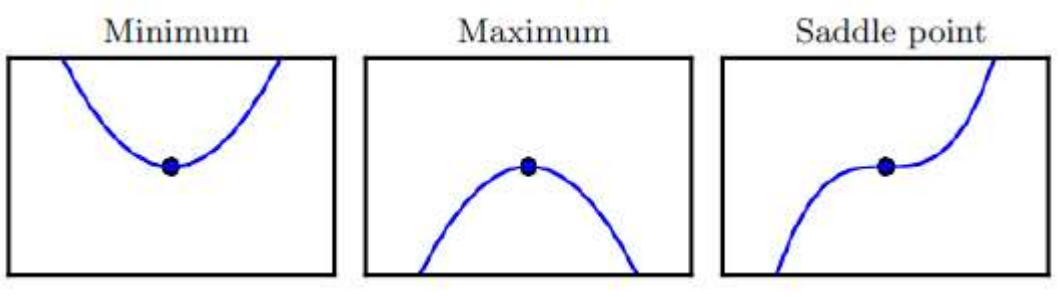
- Localmente, el gradiente indica la dirección del mayor aumento/ disminución de la función objetivo.

Optimización vía gradiente (3)

- El tamaño de los pasos que toma el gradiente en la dirección del mínimo local está determinado por la llamada tasa de aprendizaje, y determina qué tan rápido o lento nos moveremos hacia los pesos óptimos.
- Para que el Descenso Gradiente alcance el mínimo local, debemos elegir un valor apropiado para la tasa de aprendizaje, que no sea ni demasiado bajo ni demasiado alto.

Optimización vía gradiente (4)

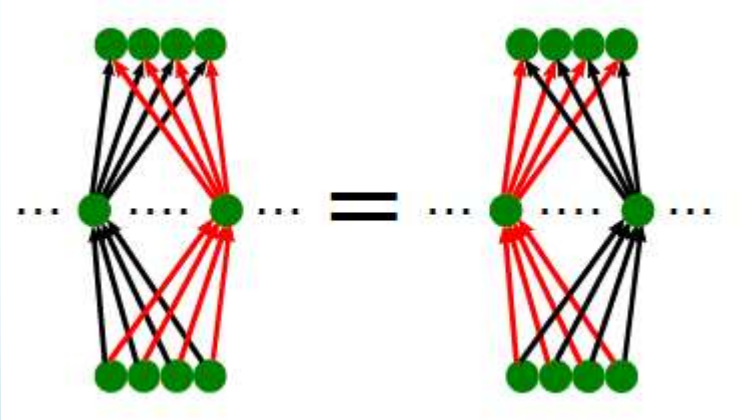
- La convergencia ocurre cuando $f'(x) = 0$. Esto ocurre o en un mínimo o en un máximo o en un punto silla.

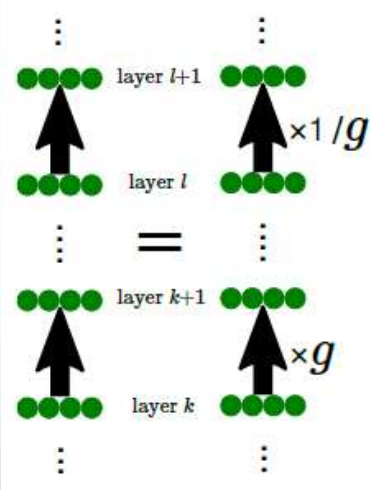


- Cuando E es convexa, el gradiente siempre apunta en la dirección del óptimo.
- En el contexto del aprendizaje profundo (deep), E es generalmente una función no convexa.
- Por lo tanto, puede tener muchos mínimos locales que no son óptimos, y muchos puntos de silla rodeados de regiones planas.

Problema de identificabilidad

- Debido a que la red tiene muchos parámetros, podemos construir exactamente el mismo modelo a partir de varias combinaciones diferentes para esos parámetros. Esto hace que existan muchas regiones del espacio de los parámetros con el mismo valor para la función de error E.





- Ver "Skip Connections as Eective Symmetry-Breaking" Emin Orhan, 2018.

Optimización versus aprendizaje

- En minería de datos, generalmente es "suficiente encontrar un buen mínimo local", ya que la función objetivo suele ser una aproximación del riesgo real y por lo tanto, no siempre tiene sentido optimizarlo intensivamente.
- De hecho, en grandes redes, el óptimo global sobre el conjunto de entrenamiento a menudo conduce a sobreajuste.

Optimización versus aprendizaje (2)

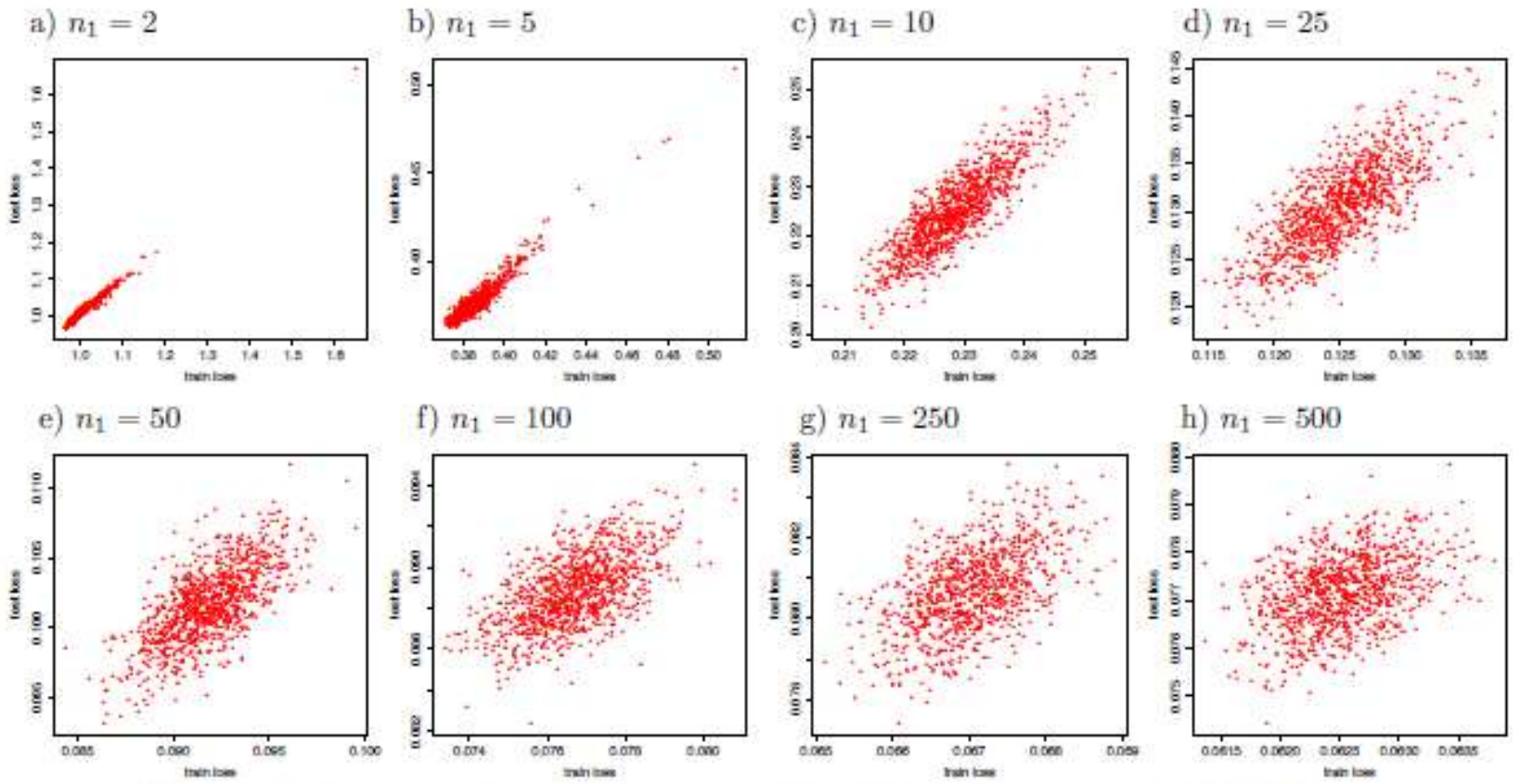
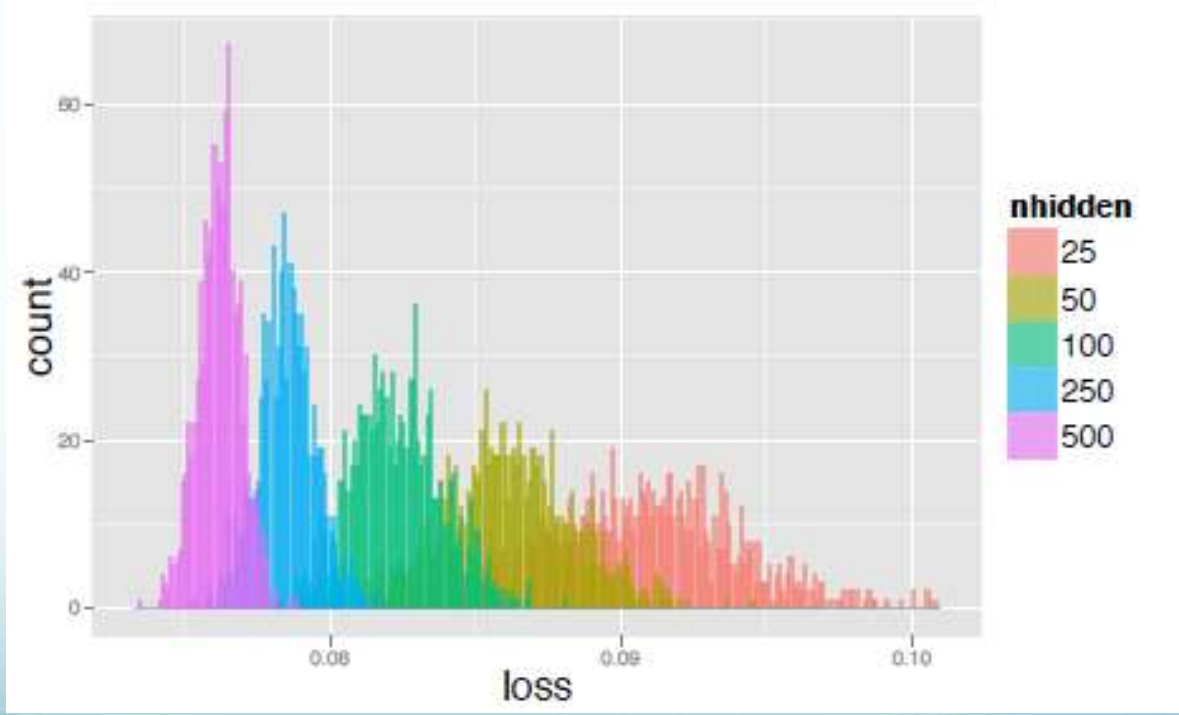


Figure 7: Test loss versus train loss for networks with different number of hidden units n_1.

Optimización versus aprendizaje (3)

- "The Loss Surfaces of Multilayer Networks", Choromanska et al. 2015.



Tipos de gradiente descendente

- Gradiente descendente batch
- Gradiente descendente estocástico
- Mini-batch

Gradiente descendente batch

- El gradiente descendente batch calcula el error para cada ejemplo dentro del conjunto de entrenamiento, pero solo después de que todos los ejemplos de entrenamiento hayan sido evaluados, el modelo se actualiza usando el gradiente promedio sobre todos los ejemplos. Todo este proceso es como un ciclo y se llama época (**epoch**) de entrenamiento.
- Las ventajas de esto son que es computacionalmente eficiente, produce un gradiente de error estable y una convergencia estable.

Gradiente descendente batch (2)

- El gradiente descendente batch tiene la desventaja de que el gradiente de error estable a veces puede dar como resultado un estado de convergencia que no es lo mejor que puede lograr el modelo.
- También requiere que todo el conjunto de entrenamiento esté en la memoria y esté disponible para el algoritmo.

Gradiente descendente estocástico (SGD)

- El descenso de gradiente estocástico (SGD) en contrario, actualiza el modelo (los pesos) después de calcular el error de cada ejemplo dentro del conjunto de entrenamiento.
- Esto puede hacer que el SGD sea más rápido que el gradiente batch, dependiendo del problema. Una ventaja es que las actualizaciones frecuentes nos permiten tener una tasa de mejora bastante detallada.
- La cuestión es que las actualizaciones frecuentes son más costosas desde el punto de vista computacional a medida que se aproxima Batch Gradient Descent.

Gradiente descendente estocástico (SGD)(2)

- La frecuencia de esas actualizaciones también puede generar gradientes ruidosos, lo que puede hacer que la tasa de error aumente, en lugar de disminuir lentamente.
- El algoritmo SGD oscila entorno al óptimo, en lugar de converger a éste. Para lograr convergencia debemos hacer la tasa de aprendizaje a cero.

Mini-batch

- Gradient descente Mini-batch es un intermedio entre SGD y gradiente descendente batch. Simplemente divide el conjunto de datos de entrenamiento en pequeños batches (lotes) y realiza una actualización para cada uno de estos lotes. Por lo tanto, crea un equilibrio entre la robustez del descenso de gradiente estocástico y la eficiencia del descenso de gradiente por lotes.
- Comúnmente los tamaños de los mini-batches oscilan entre 50 y 256, pero al igual que con cualquier otra técnica de aprendizaje automático, no existe una regla clara, ya que pueden variar para diferentes aplicaciones.
- Esta técnica es la más usada para entrenar reds neuronales profundas.

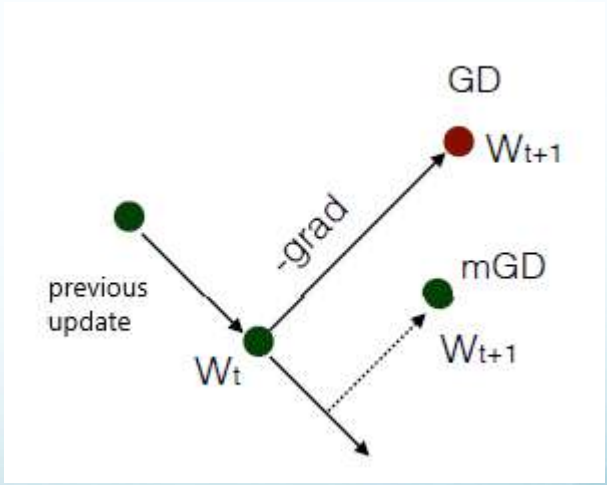
Variantes de Gradiente Descendente

- Los métodos anteriores determinan la cantidad de datos a utilizar para actualizar el modelo en cada época.
- A continuación veremos variantes del gradiente estocástico clásico:
  - Momentum
  - Momentum Nesterov
  - Adam
  - RMSprop
  - Adagrad

Momentum

- En lugar de depender solo del gradiente actual para actualizar el peso, el descenso del gradiente con momentum (Polyak, 1964) reemplaza el gradiente actual con v_t (que significa velocidad).

$$
\begin{aligned}
g_t &\longleftarrow \nabla\left(E(W^{(t)})\right) \\
v_{t+1} &\longleftarrow \gamma v_t + \eta g_t \\
w_{t+1} &\longleftarrow \gamma w_t - v_{t+1}
\end{aligned}
$$



- El parámetro γ usualmente es 0,9.

Momentum (2)

- El método calcula la media móvil exponencial de los gradientes actuales y pasados (es decir, hasta el tiempo t).

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \eta g_1 \\
v_2 &= \eta g_2 + \gamma \eta g_1 \\
v_3 &= \eta g_3 + \gamma \eta g_2 + \gamma \eta^2 g_1 \\
v_t &= \eta \sum_{j=1}^{t} \gamma^{j-t} g_j
\end{aligned}
$$

- El momentum es una de las técnicas más utilizadas para actualizar los pesos.

Momentum Nesterov

- La diferencia entre el momentum de Nesterov y el momentum estándar radica en donde se evalúa el gradiente.
- El impulso de Nesterov evalúa el gradiente después de aplicar la velocidad actual.
- Por lo tanto, podemos ver el impulso de Nesterov como un intento de agregar un factor de corrección al método estándar de momentum.

Momentum Nesterov (2)

$$
\begin{aligned}
g_t &\longleftarrow \nabla(E(W^{(t)} - \gamma v_t) \\
v_{t+1} &\longleftarrow \gamma v_t + \eta g_t \\
w_{t+1} &\longleftarrow \gamma w_t - v_{t+1}
\end{aligned}
$$

- γ también suele usarse en 0,9.

Algoritmos con learning rate adaptivo

- Idea: Tiene sentido asígnar valores diferentes para la tasa de aprendizaje de cada parámetro y adaptar automáticamente estas tasas de aprendizaje a lo largo del entrenamiento.

Progressive decay

- Como discutimos anteriormente, para un valor de x, el algoritmo de gradiente estocástico puede no converger al óptimo.
- Para mitigar el problema, usamos decadencia progresiva (**progressive decay**)

1. Inicializar $\eta = \eta_0$
2. En cada iteración:
   1. $\eta(s) = \frac{\eta_0}{1+s\eta_d}$

- η_d es la tasa de decaimiento.

Adagrad

- Adagrad adapta individualmente las tasas de aprendizaje de todos los parámetros del modelo al escalarlos de forma inversamente proporcional a la raíz cuadrada de la suma de todos los cuadrados de los gradientes anteriores.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t + \varepsilon}} \cdot \frac{\delta E}{\delta w_t}$$

- Donde

$$S_t = S_{t-1} + \left(\frac{\delta E}{\delta w_t}\right)^2$$

- $S_0 = 0$

Adagrad (2)

- Observe que se agrega ε al denominador. Keras llama a esto el *fuzz factor*, un pequeño valor de punto flotante para garantizar que nunca tendremos una división por cero.
- Valores por defecto (de Keras):
  - $\alpha = 0.01$
  - $\varepsilon = 10^{-7}$

RMSprop

- RMSprop es una mejora de AdaGrad, en lugar de tomar la suma acumulativa de gradientes cuadrados, tomamos su media móvil exponencial.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t + \varepsilon}} \cdot \frac{\delta E}{\delta w_t}$$

- Donde

$$S_t = \beta S_{t-1} + (1-\beta)\left(\frac{\delta E}{\delta w_t}\right)^2$$

- Valores por defecto (de Keras):
  - $\alpha = 0{,}001$
  - $\beta = 0{,}9$
  - $\varepsilon = 10^{-6}$

Adam

- Adam es una combinación entre momentum y RMSprop:
  1. Usa un gradiente V, que es la media móvil exponencial de los gradientes, como en el momentum.
  2. El learning rate α se divide por la raíz cuadrada de S, que es la media móvil exponencial de los gradientes cuadrados, como en RMSprop.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{S}_t} + \varepsilon} \cdot \hat{V}_t$$

Adam (2)

- Donde

$$\hat{V}_t = \frac{V_t}{1 - (\beta_1)^{t+1}}$$

$$\hat{S}_t = \frac{S_t}{1 - (\beta_2)^{t+1}}$$

- son las correcciones.

Adam (3)

- Y

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1) \frac{\delta E}{\delta w_t}$$

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2) \left(\frac{\delta E}{\delta w_t} \right)^2$$

- con V y S inicializados a 0.
- Valores predeterminados propuestos por los autores:
  - $\alpha = 0{,}001$
  - $\beta_1 = 0{,}9$
  - $\beta_2 = 0{,}999$
  - $\varepsilon = 10^{-8}$
- β_1 y β_2 son los factores de olvido para los gradientes y los segundos momentos de los gradientes, respectivamente.