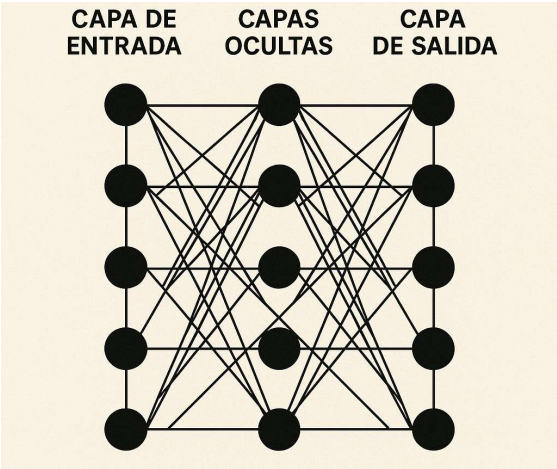
Visión Computacional

Ivan Sipiran

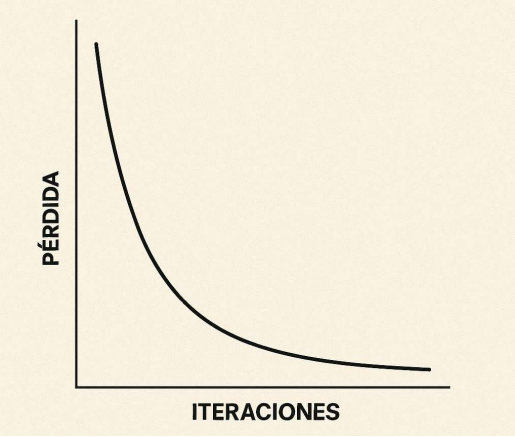
Ingredientes

Datos

Modelo



Loss function

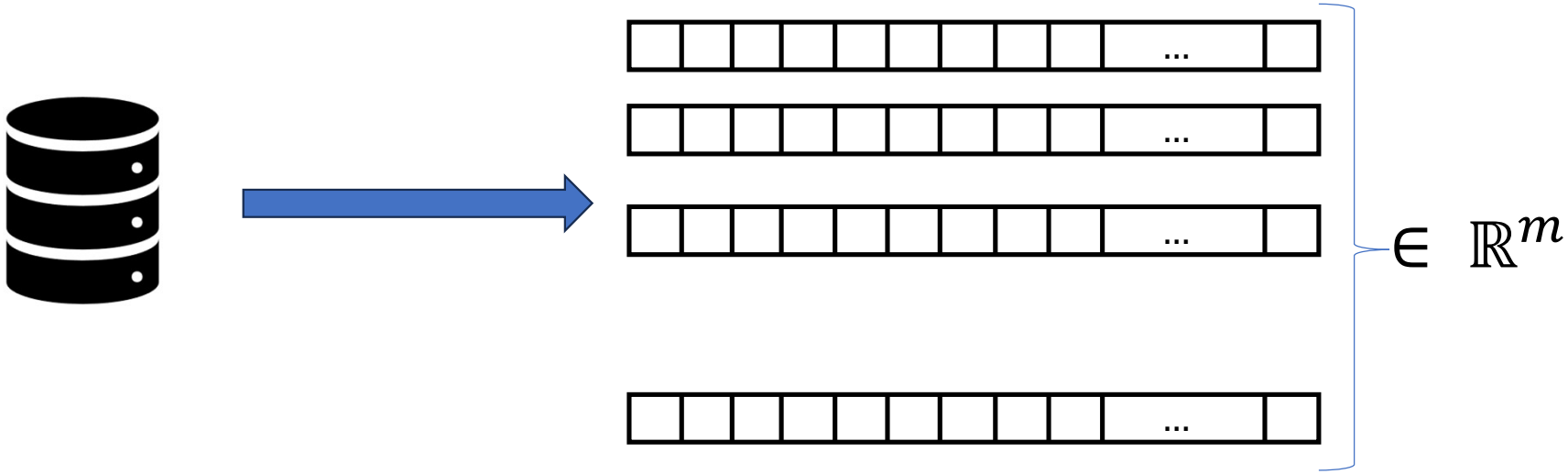


Algoritmo de Aprendizaje

Datos

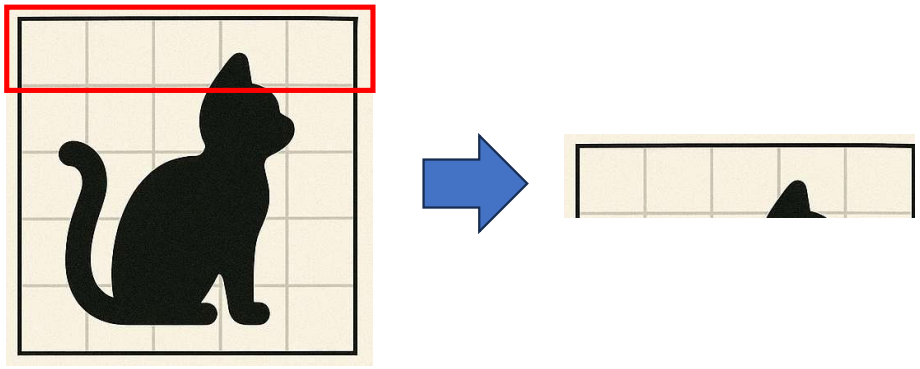
Los datos

- Redes neuronales son modelos numéricos
  - Todos los datos tienen que ser números
- Representación vectorial de los datos



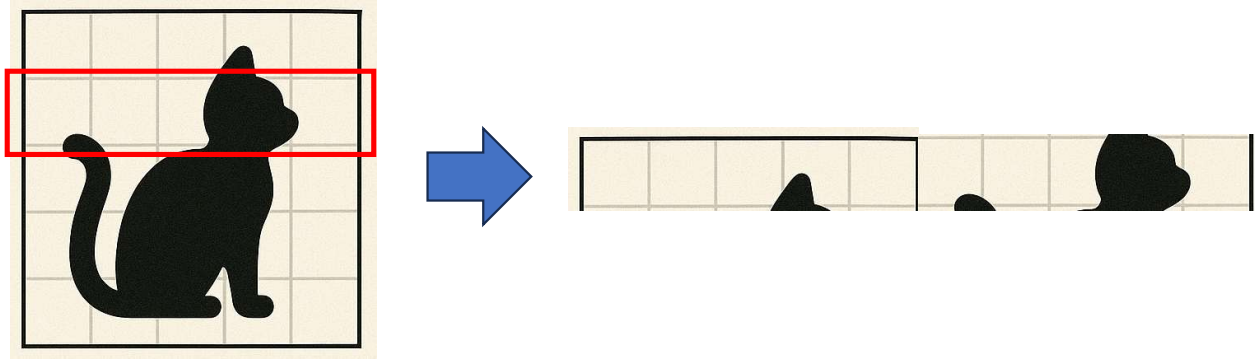
Los datos

- Ejemplo: imágenes



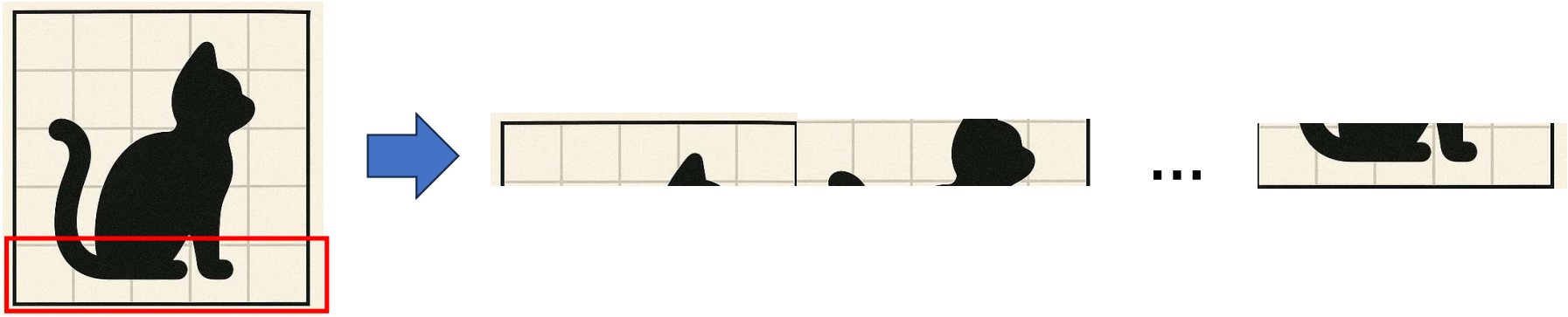
Los datos

- Ejemplo: imágenes



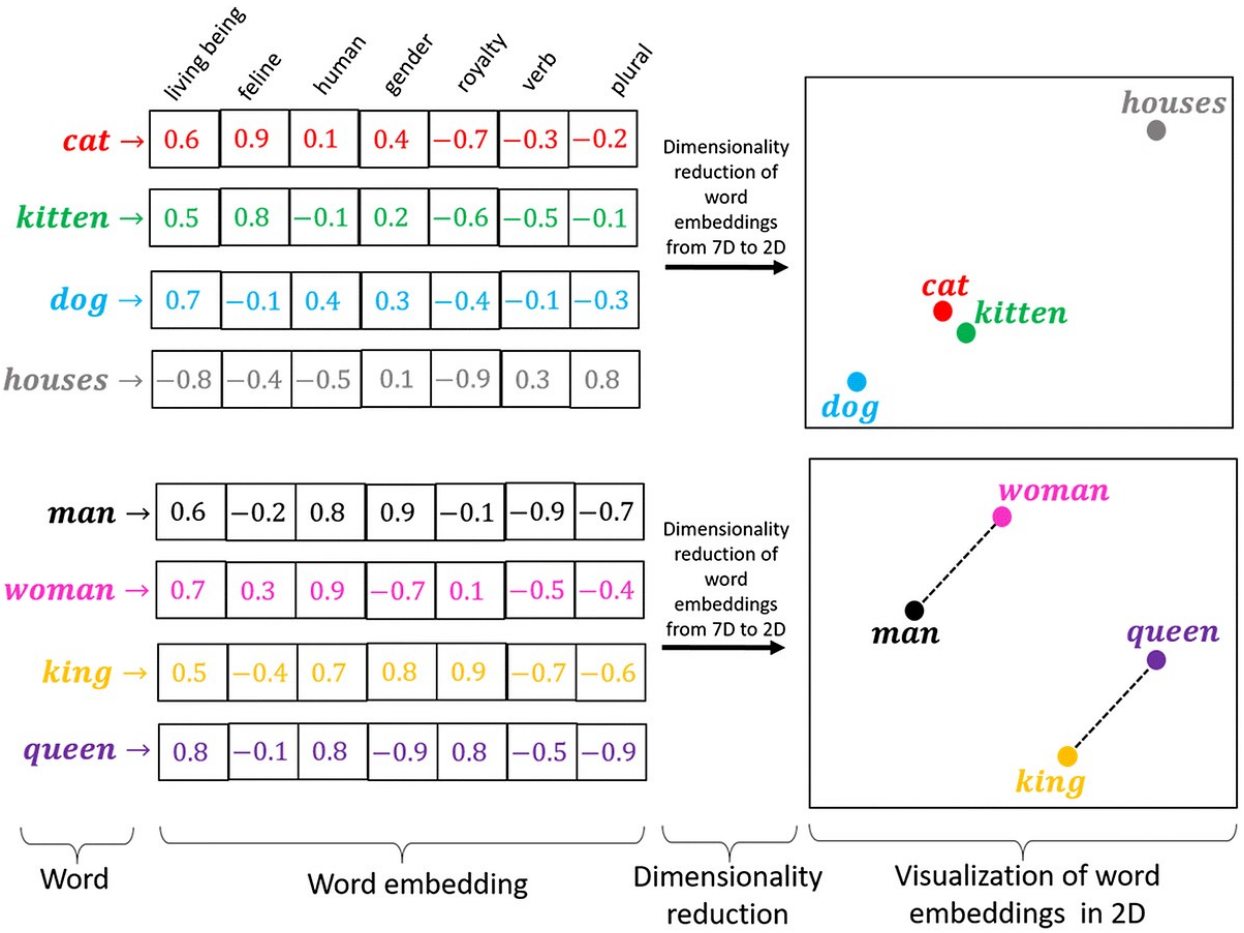
Los datos

- Ejemplo: imágenes



Los datos

- Ejemplo: texto



Los datos - Proyecciones

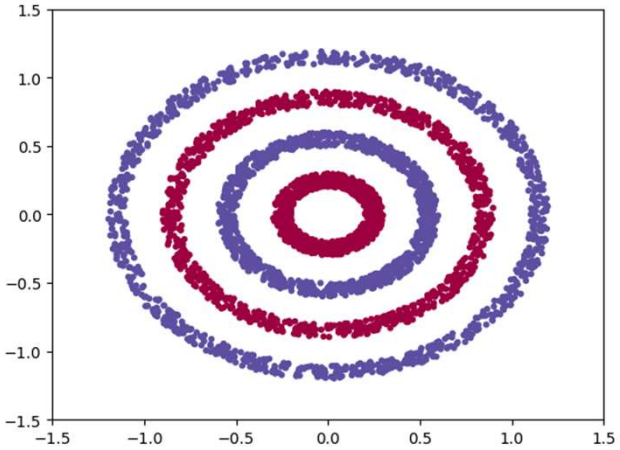
- Un dato se puede representar entonces como un vector numérico X de dimensión $1 \times m$
- Una proyección linear es una operación que convierte X en otro vector X'de una dimensión distinta
  - Esta operación se puede representar con una matriz

$$X' = X \times P_{m,q}$$

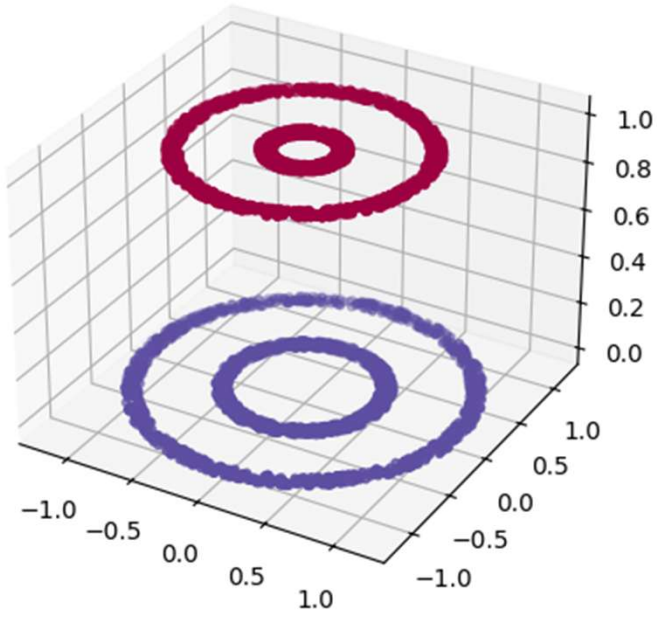
Los valores m y q pueden ser cualquier número entero:

- Si $m < q$, hay un up-dimension
- Si $m > q$, hay un down-dimension

Los datos - Proyecciones



2D a 3D



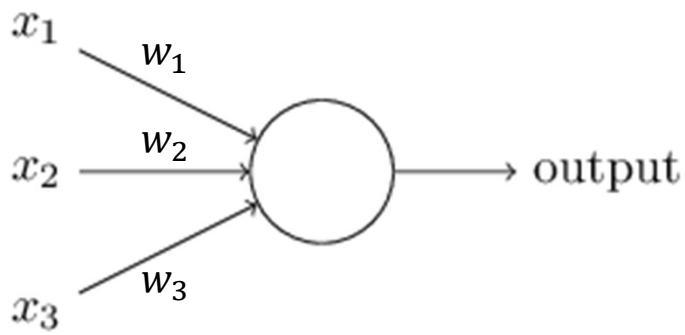
$$[1.5 \quad 2.5] \begin{bmatrix} 0.5 & 0.6 & -0.8 \\ 0.6 & -1.2 & 0.2 \end{bmatrix} = [1.0 \quad 2.5 \quad 0.5]$$

Arquitectura

Perceptron

Representación matemática de una neurona

Entradas binarias



Salida binaria

Pesos expresan la importancia de una entrada dada para generar una salida. Existe una regla para la salida:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Umbral permite a la neurona activarse o inhibirse

Perceptron

La regla del perceptrón puede ser escrita en una forma distinta:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

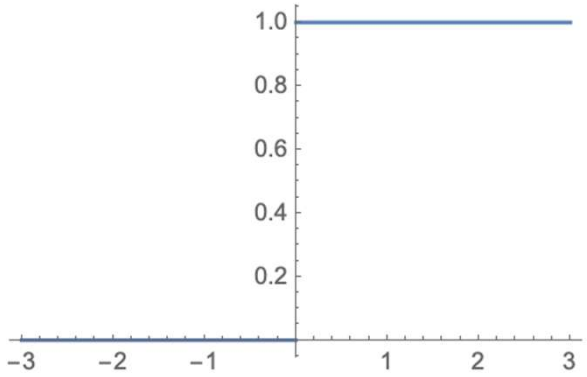
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Donde $b = -threshold$, comúmente conocido como el bias.

El preceptrón se activa solo si el producto interno entre pesos y entradas es más grande que el bias. **El bias es el parámetro que representa que tan fácil es activar una neurona.**

Función de activación

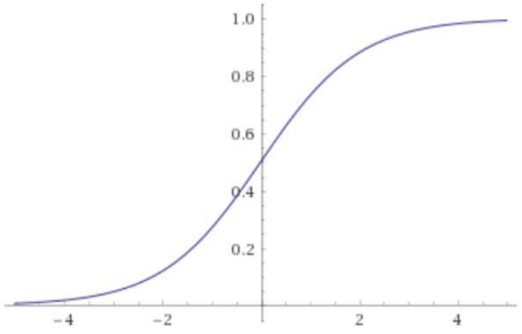
Razón: umbralización simple es una función no suave



Solución: Reemplazar la función step con una función suave de comportamiento similar

Función Sigmoide

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

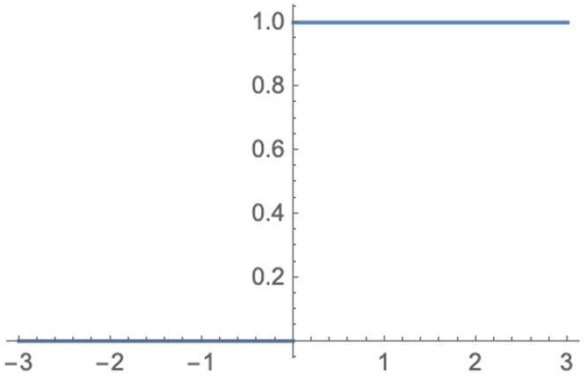


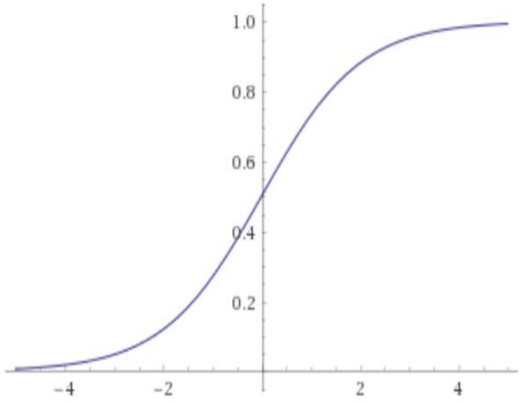
Así que la salida de la neurona es:

$$\sigma(w.z + b) = \frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

Aprendizaje de perceptrones

Razonamiento: ambas funciones lucen muy distintas matemáticamente, sin embargo tienen un comportamiento muy similar





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

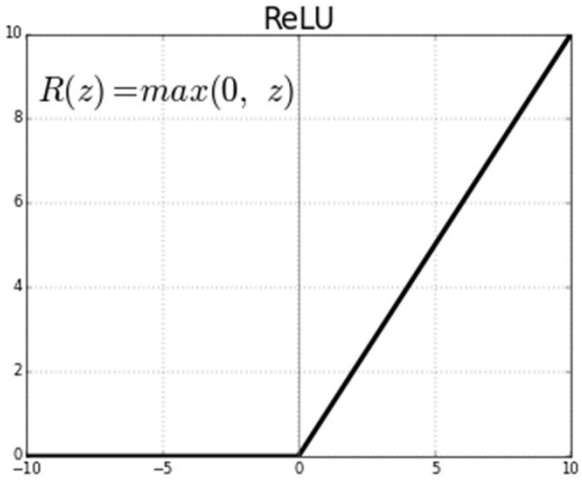
- Cuando z es un número positivo grande, entonces $e^{-z} \approx 0$, $\sigma(z) \approx 1$ — **Like step function**
- Cuando z es un número negativo grande, entonces $e^{-z} \to \infty$, $\sigma(z) \approx 0$ — **Like step function**
- La diferencia es cuando z tiene valores cercanos a cero

$\sigma(z)$ es suave alrededor de cero, también diferenciable

La búsqueda de una función de activación

La función sigmoide no es útil en arquitecturas profundas, porque los gradientes se desvanecen
El éxito de los métodos de Deep learning es gracias a funciones de activación más efectivas

La función ReLU



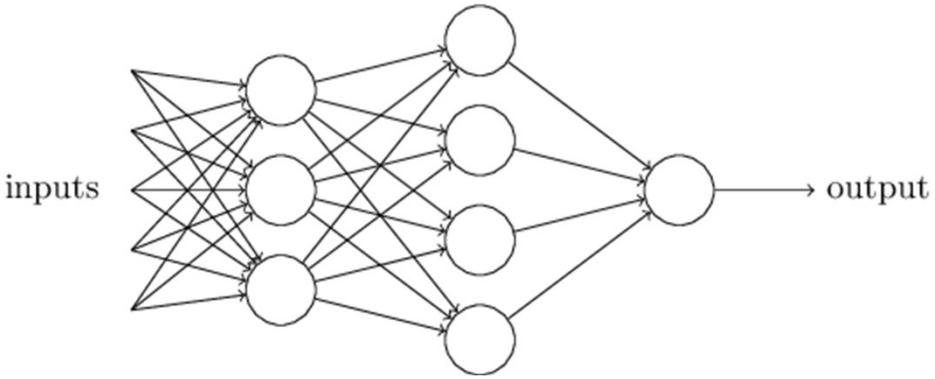
- Función más simple
- Valores negativos se truncan a cero
- El gradiente es muy simple:

$$\frac{\partial R}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Perceptron Multicapa

Un perceptrón implementa una regla de decisión muy simple. Es imposible solucionar un problema real con una sola neurona.

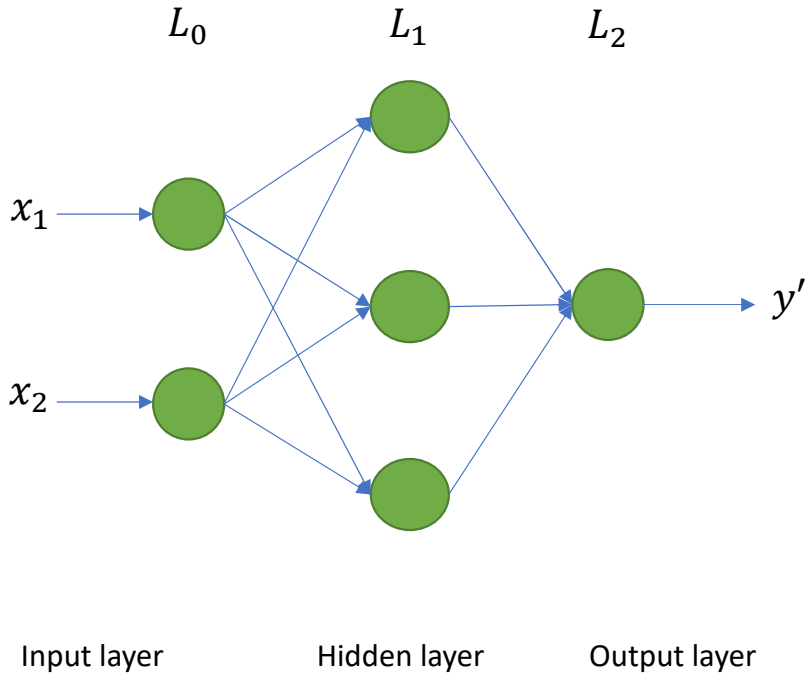
Para hacer reglas de decisión más complejas, podemos implementar conexiones complejas de perceptrones



Perceptrones pueden ser organizados en capas: capas de entrada (reciben los datos), capa de salida (computa la salida), y capas ocultas (para tomar decisiones internas)
Este modelo es comúnmente conocido como Multi-layer perceptron o MLP.

Arquitectura de red

Diseñamos una red de neuronas sigmoide: secuencia de capas con neuronas



Neuronas de entrada son representadas pero no realizan computación

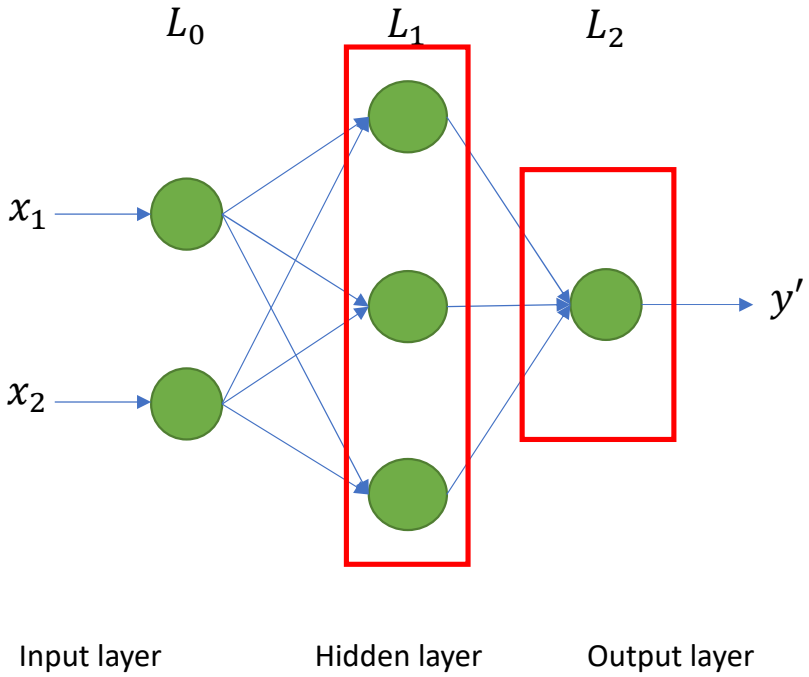
Toda flecha en esta figura representa un peso (parámetro)

Por conveniencia, empaquetamos los pesos de la primera capa oculta en una matriz W^1 de dimensión 2×3. Pesos en la capa de salida son empaquetados en una matriz W^2 de dimensión 3×1.

Biases son organizados en vectores: b^1 de dimensión 3 y b^2 de dimensión 1.

Arquitectura de red

Diseñamos una red de neuronas sigmoide: secuencia de capas que contienen neuronas



Computación de la red

Sea $x = [x_1, x_2]$ un vector fila, computamos

$$X_1 = \sigma(x \cdot W^1 + b^1)$$

$$y' = \sigma(X_1 \cdot W^2 + b^2)$$

Un vector de dimensión 3
Suma con b es un broadcast

Por favor, chequear dimensiones

En general, la salida de una capa es

$$X_i = \sigma(X_{i-1} \cdot W^i + b^i)$$

Loss Function

Función de costo

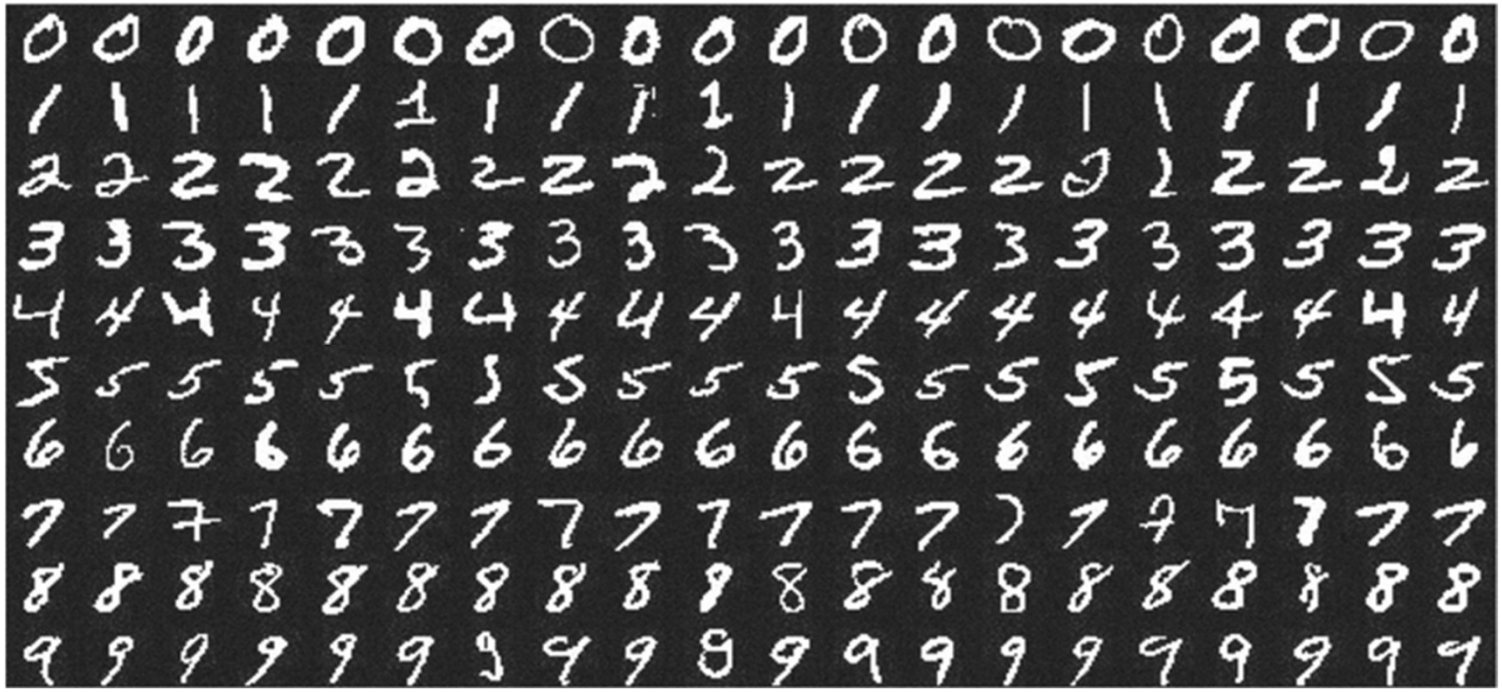
- Para problemas supervisados, tenemos un conjunto de datos $\{x_i, y_i\}_{i=1}^{N}$, donde $x_i \in \mathbb{R}^m$ y $y_i \in \mathbb{R}^l$. Cada dato está compuesto por vector de entrada x_i y salida deseada y_i.
- Si la salida de la red neuronal es y_i', es necesario medir que tan buena es ésta salida con respecto al deseado y_i.
- En un problema binario, $y_i = \{0,1\}$, un costo útil puede ser:

$$C = \frac{1}{2}(y_i' - y_i\ )^2$$

- Si la red produce una salida igual a la esperada el costo es cero.
- Si la red produce una salida diferente a la esperada, el costo es ½.

Loss function multiclase

Dataset con dígitos manuscritos – 10 clases



Objetivo

???

0 1 2 3 4 5 6 7 8 9

Objetivo

???

0 1 **2** 3 4 5 6 7 8 9

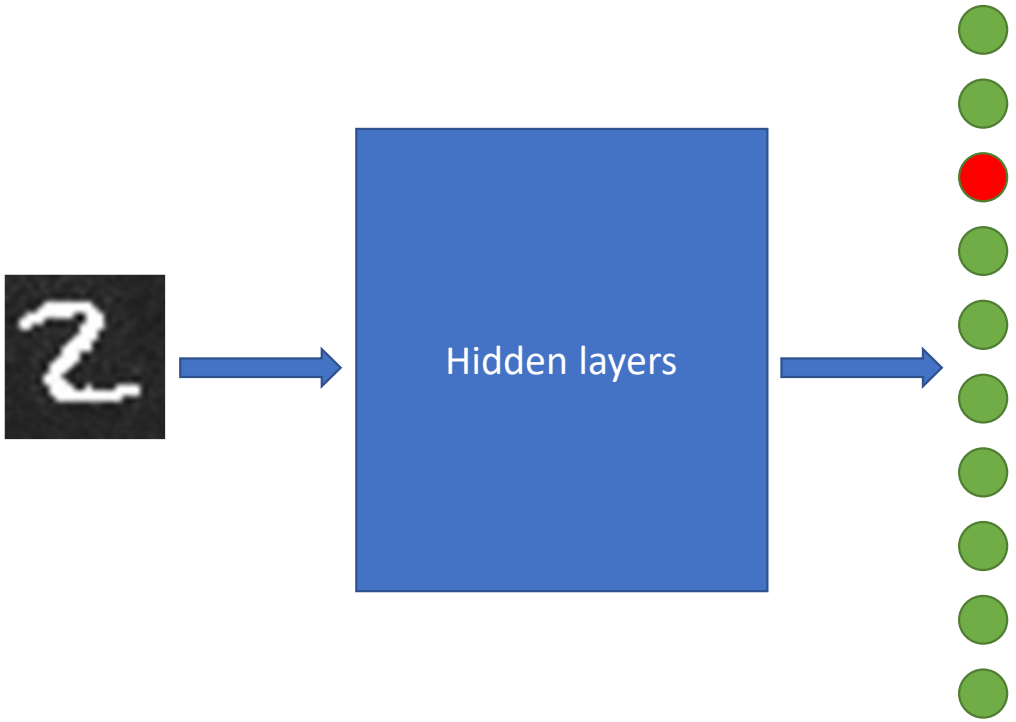
Nuestra red debería computar una probabilidad de que la entrada pertenezca a una clase.

La salida ideal debería ser el one-hot encoding para la clase correcta

[0, 0, 1, 0, 0, 0, 0, 0, 0, 0] → Target de entrenamiento!

Salida multi-clase

- Necesitamos una red neuronal con n neuronas de salida



Podemos usar neuronas sigmoide en la capa de salida, sin embargo esto no garantiza que la salida sea una función de densidad de probabilidades(pdf).
Podemos normalizar la salida para obtener una pdf.

Función de activación Softmax

Recordemos la computación de la función lineal en la última capa de una NN (antes de la función de activación)

$$z_L = X_{L-1} \cdot W^L + b^L$$

L es el número de capas

z_L es un vector real de dimensión n. Por ejemplo, supongamos que tenemos una NN con 3 neuronas en la salida:

$$z_L = \begin{bmatrix} -1.45 \\ -7.82 \\ -3.78 \end{bmatrix}$$

Cuál es la interpretación de estos valores?

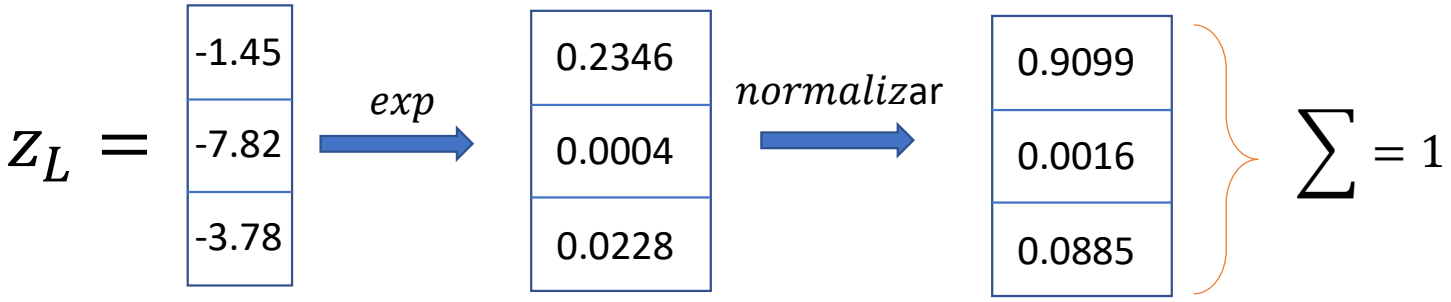
Función de activación Softmax

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

$$z_L = \begin{bmatrix} -1.45 \\ -7.82 \\ -3.78 \end{bmatrix} \xrightarrow{exp} \begin{bmatrix} 0.2346 \\ 0.0004 \\ 0.0228 \end{bmatrix}$$

Función de activación Softmax

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$



Softmax transforma cualquier salida en una *función de densidad de probabilidades (pdf)*

Softmax como función de activación de la capa de salida

Cross-entropy Loss

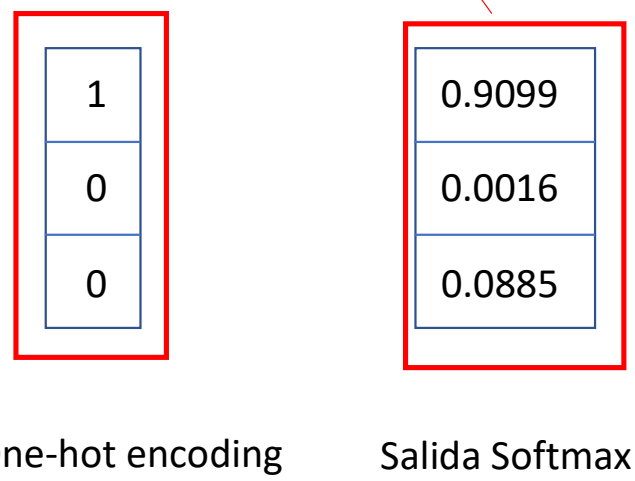
- Nuestra red ahora produce una pdf
- La salida objetivo (one-hot encoding) es también una pdf
- Cómo comparamos dos pdf’s: Cross-entropy loss

$$L(y, y') = -\sum_{i=1}^{n} y_i \log(y_i')$$

Cross-entropy Loss

- Por ejemplo, para el problema de 3 clases, podemos tener

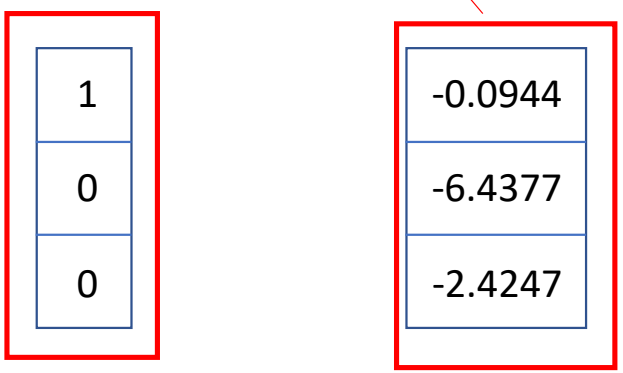
$$L(y, y') = -\sum_{i=1}^{n} y_i \log(y'_i)$$



Cross-entropy Loss

- Por ejemplo, para el problema de 3 clases, tenemos

$$L(y, y') = -\sum_{i=1}^{n} y_i \log(y_i')$$

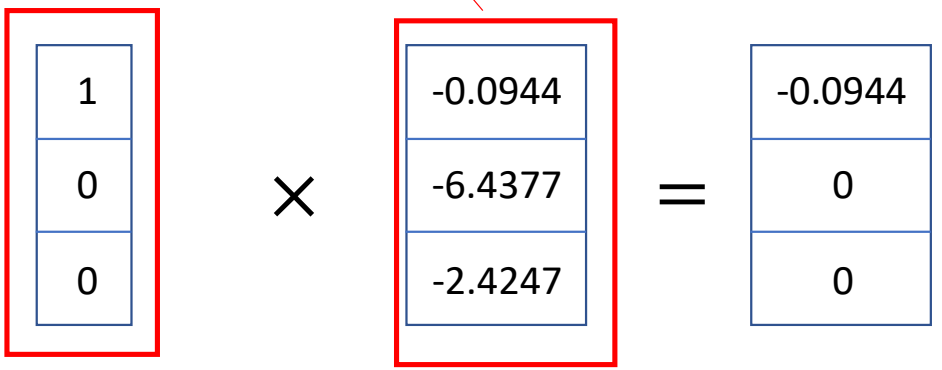


One-hot encoding

Cross-entropy Loss

- Por ejemplo, para el problema de 3 clases, tenemos

$$L(y, y') = -\sum_{i=1}^{n} y_i \log(y_i')$$



One-hot encoding

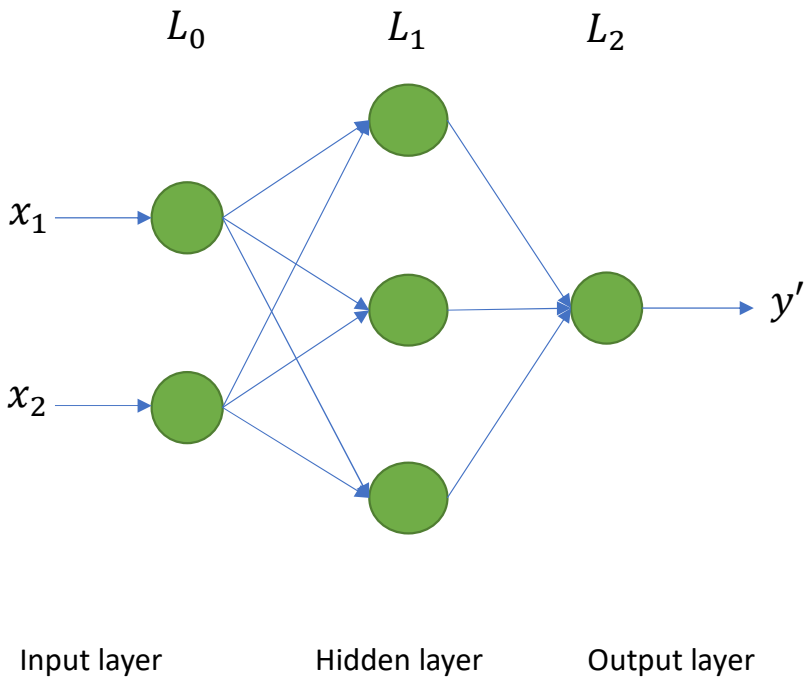
Por lo tanto,

$$L(y, y') = 0.0944$$

Algoritmo de Aprendizaje

Red neuronal como función

- Tomemos el siguiente ejemplo de red



Sea $x = [x_1, x_2]$ un vector fila, computamos

$$X_1 = \sigma(x \cdot W^1 + b^1)$$

$$y' = \sigma(X_1 \cdot W^2 + b^2)$$

Y la loss function es

$$C = \frac{1}{2}\sum_{i=1}^{N}(y_i' - y_i)^2$$

$$C = \frac{1}{2}\sum_{i=1}^{N}(\sigma((\sigma(x \cdot W^1 + b^1)) \cdot W^2 + b^2) - y_i)^2$$

$C(W^1, W^2, b^1, b^2)$

Cómo cambiamos los parámetros para disminuir el costo?

Gradiente descendiente

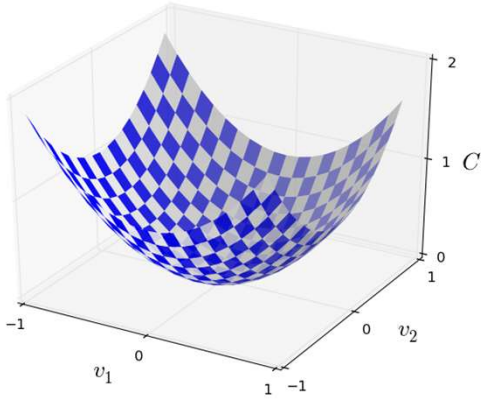
Aprendizaje como un problema de optimización:

Encontrar pesos y biases los cuales minimizan la función de costo cuadrático

Cómo podemos hacer eso? Veamos alguna intuición

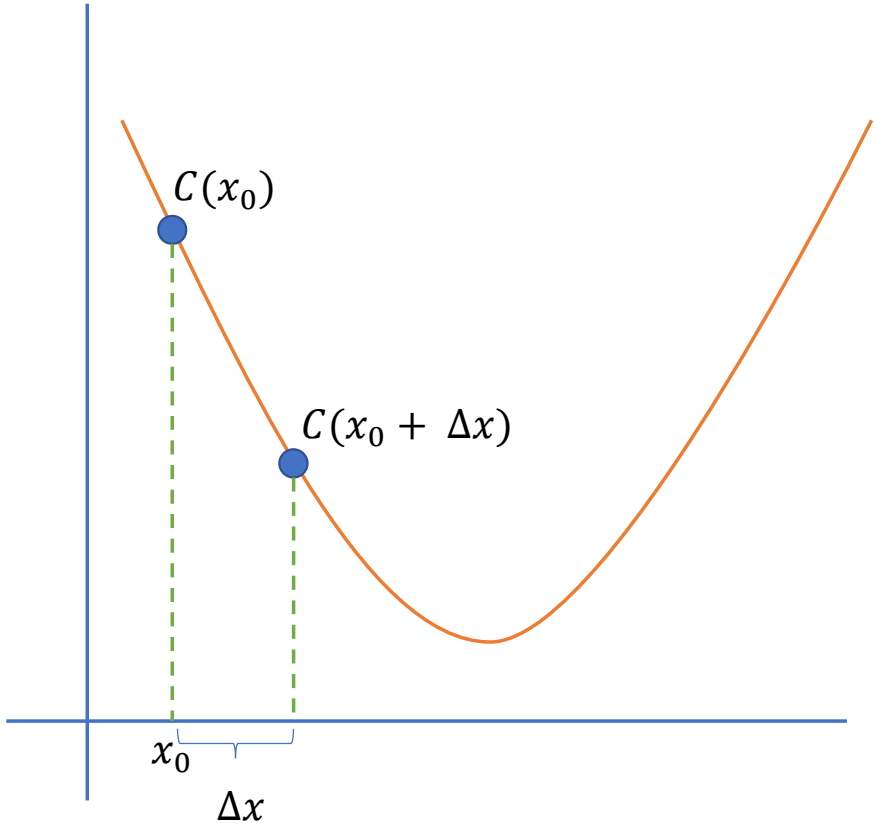
Supongamos que tenemos una función de valores reales de muchas variables $C(v)$, definida como

$$C\colon R^n \to R$$



El problema es encontrar el mínimo global de la función de optimización
Podríamos intentar analíticamente con cálculo, pero recuerda que las redes tienen millones de parámetros

Gradiente descendiente



Derivada de C en el punto x_0

$$\frac{\partial C}{\partial x} = \frac{C(x_0 + \Delta x) - C(x_0)}{\Delta x} = \frac{\Delta C}{\Delta x}$$

$$\Delta C = \frac{\partial C}{\partial x} \Delta x$$

El cambio en la función depende de la derivada de la función y el cambio en los parámetros

Gradiente descendiente

Para generalizar la regla a muchas más variables, la función C puede depender de un conjunto de variables $v = (v_1, v_2, \cdots, v_m)$

$$\Delta C = \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + \cdots + \frac{\partial C}{\partial v_m}\Delta v_m$$

Necesitamos escoger $\Delta v_1, \Delta v_2, \cdots, \Delta v_m$ tal que ΔC es negativo (un cambio negativo en C)

Escribimos la ecuación en una forma conveniente:

$$\Delta C \approx \nabla C \cdot \Delta v$$

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \cdots, \frac{\partial C}{\partial v_m}\right)^T \qquad \Delta v = (\Delta v_1, \Delta v_2, \cdots, \Delta v_m)^T$$

Gradiente descendiente

Para hacer ΔC negativo, escogeremos

$$\Delta v = -\eta \nabla C$$

Note que si reemplazamos en la ecuación

$$\Delta C \approx \nabla C \cdot \Delta v$$

Tendremos $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$

Como $\|\nabla C\|^2 \geq 0$, esto garantiza que $\Delta C \leq 0$

Conclusión: tomando $\Delta v = -\eta \nabla C$ siempre hará que la función C se decremente

Regla de actualización iterativa $v = v - \eta \nabla C$

Para cada parámetro en la función

Gradiente descendiente

El rol del hiper-parámetro η

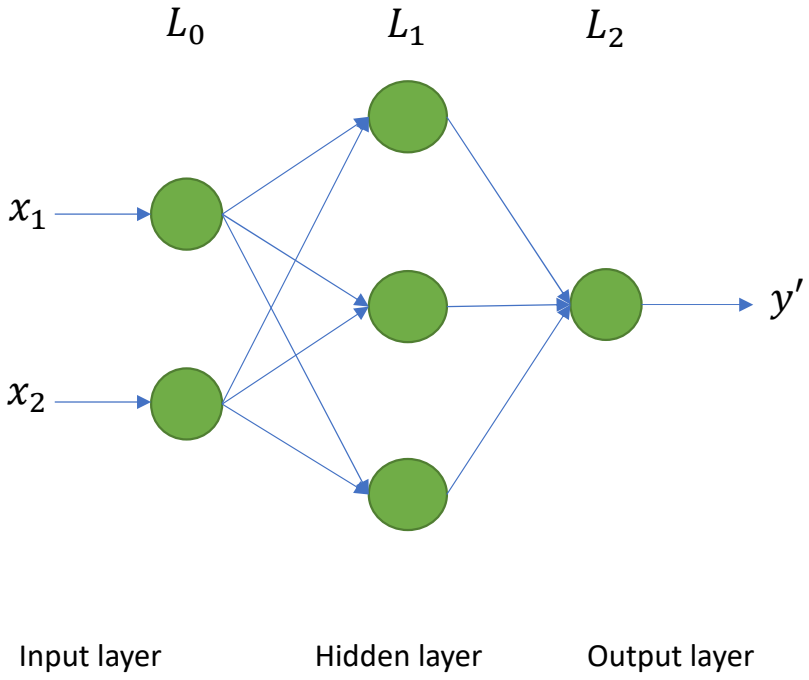
$$v = v - \eta \nabla C$$

- Si η es muy pequeño — Realizaremos pasos pequeños en la minimización, GD será lento
- Si η es muy grande — Cerca del mínimo, podemos obtener $\Delta C > 0$. Salto alrededor del mínimo

En la práctica, heurísticas empíricas son útiles, las veremos después en el curso!

Backpropagation

Diseñamos una red de neuronas sigmoide: secuencia de capas que contienen neuronas



Y el algoritmo de entrenamiento?

A primera vista, la computación de gradientes parece más complicada ahora. Sin embargo, veremos una estrategia.

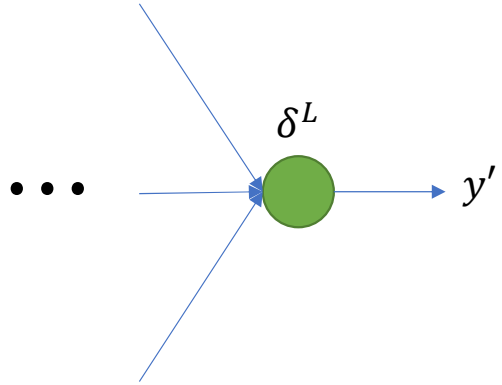
Definición: dada la j-ésima neurona en la capa l, el error se define como

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Donde, C es la función de costo y z es el resultado de la ecuación lineal en la neurona

Arquitectura de red

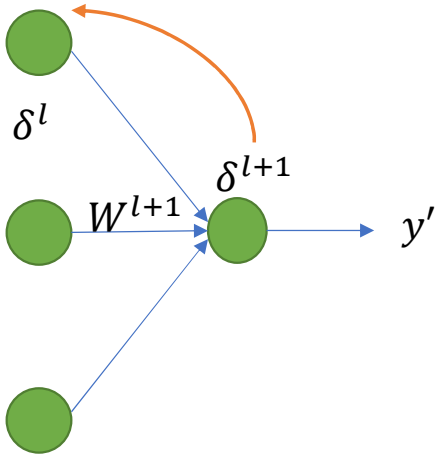
El error en la capa de salida



$$\delta^L = \frac{\partial C}{\partial y'}\sigma'(z^L)$$

$$= (y' - y) \cdot \sigma'(z^L)$$

El error de una capa en términos del error de la siguiente capa



$$\delta^l = (\delta^{l+1}(W^{l+1})^T) \odot \sigma'(z^l)$$

Error va hacia atrás a través de los pesos

Arquitectura de red

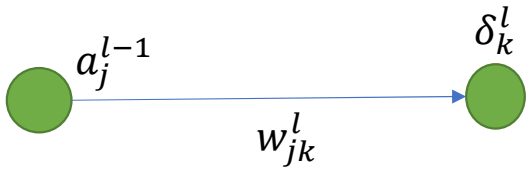
Gradiente de biases

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

El gradiente en el bias, es de hecho,el error

El gradiente de los pesos

$$\frac{\partial C}{\partial w_{jk}^l} = a_j^{l-1}\delta_k^l$$



Backpropagation

Algoritmo GD
Input: Conjunto de muestras $\{(x_1, x_2, y)\}$, learning rate η, número de épocas E

Para cada muestra de entrenamiento x: la activación de entrada es a^1

- **Feedforward:** Para cada capa $l = 2, 3, \cdots, L$ computar

$$z^l = a^{l-1}W^l + b^l$$
$$a^l = \sigma(z^l)$$

- **Error de salida:** Computar

$$\delta^L = \nabla C \cdot \sigma'(z^l)$$

- **Retropropagar el error**: Para cada capa $l = L-1, L-2, \cdots$, computar

$$\delta^l = (\delta^{l+1}(W^{l+1})^T) \odot \sigma'(z^l)$$

- **Computar gradiente**

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \frac{\partial C}{\partial w_{jk}^l} = a_j^{l-1}\delta_k^l$$

Ingredientes Adicionales

Mejor optimización

Gradiente descendente revisitado

Nuestro procedimiento de optimización consiste de aplicar la regla de actualización en los parámetros para reducir el error de la función de costo

$$w = w - \eta \frac{\partial C}{\partial w}$$

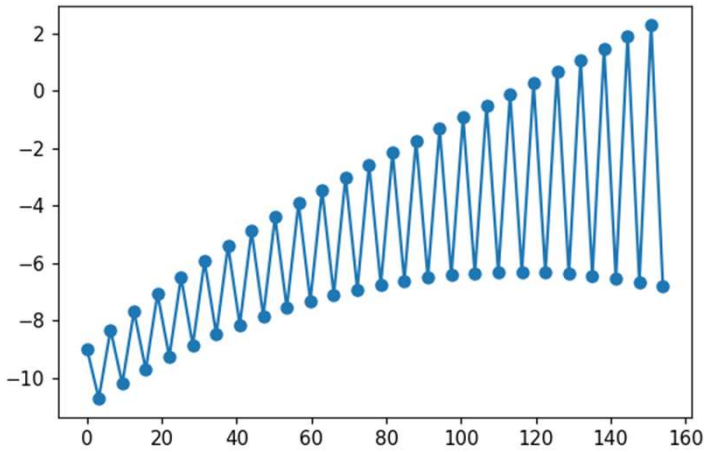
Problemas con algoritmo clásico:

- Dependencia de learning rate η
- Direcciones oscilantes en el mínimo
- Quedarse pegado en plateau o saddle points

Necesitamos mejores estrategias para actualizar parámetros

Exponentially weighted average

Supongamos que tenemos una secuencia de valores que varían en el tiempo y_t (time observations)



Queremos computar la tendencia de la secuencia aplicando promedios en una ventana de tiempo.

En lugar de usar un promedio simple en una ventana de tiempo, usamos pesos con decaimiento exponencial para el promedio y un estimado inicial v_0. Para estimar el t-ésimo elemento en la secuencia, usamos

$$v_t = \beta v_{t-1} + (1-\beta) y_t$$

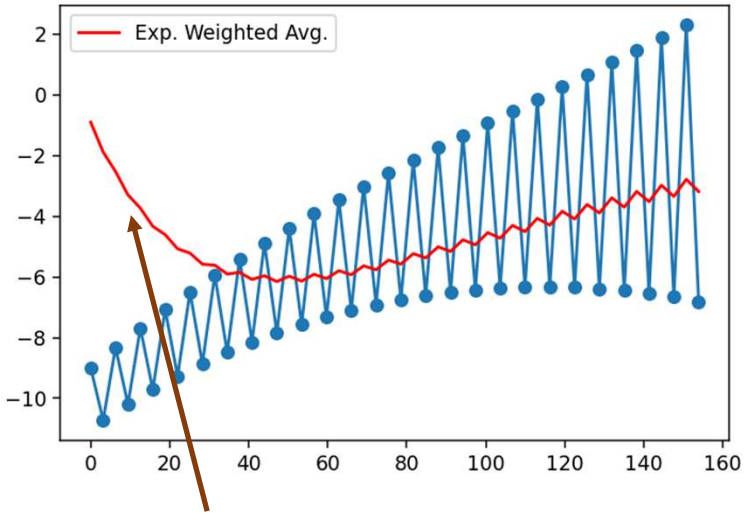
Promedio del tiempo previo

Observación actual

Parámetro $\beta < 1$ es el factor para el promedio ponderado

Exponentially weighted average

Supongamos que tenemos una secuencia de valores que varían en el tiempo y_t(time observations)



Primeros promedios no tienen suficiente información

Queremos computar la tendencia de la secuencia aplicando promedios en una ventana de tiempo.

En lugar de usar un promedio simple en una ventana de tiempo, usamos pesos con decaimiento exponencial para el promedio y un estimado inicial v_0. Para estimar el t-ésimo elemento en la secuencia, usamos

$$v_t = \beta v_{t-1} + (1-\beta) y_t$$

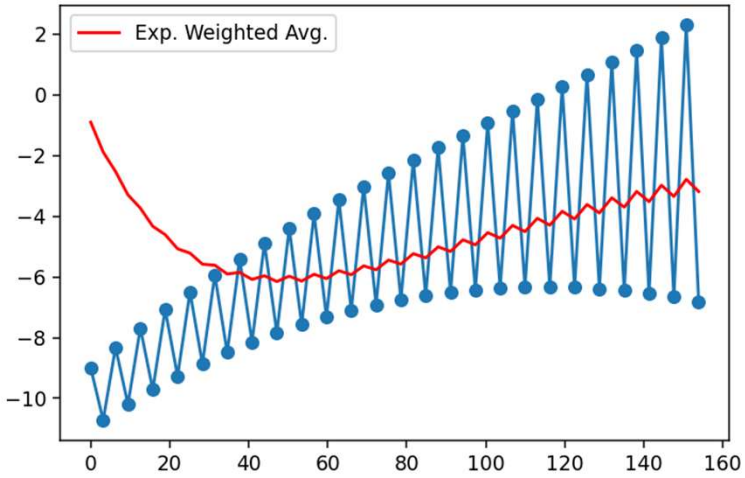
Promedio del tiempo previo

Observación actual

Parámetro $\beta < 1$ es el factor para el promedio ponderado

Exponentially weighted average

Supongamos que tenemos una secuencia de valores que varían en el tiempo y_t(time observations)



Queremos computar la tendencia de la secuencia aplicando promedios en una ventana de tiempo.

En lugar de usar un promedio simple en una ventana de tiempo, usamos pesos con decaimiento exponencial para el promedio y un estimado inicial v_0. Para estimar el t-ésimo elemento en la secuencia, usamos

$$v_t = \beta v_{t-1} + (1-\beta) y_t$$

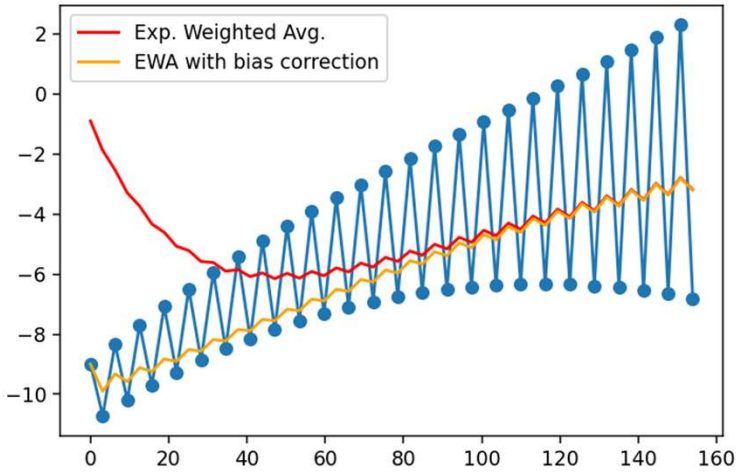
$$v_t = \frac{v_t}{1-\beta^t}$$

Bias correction

Parámetro $\beta < 1$ es el factor para el promedio ponderado

Exponentially weighted average

Supongamos que tenemos una secuencia de valores que varían en el tiempo y_t(time observations)



Queremos computar la tendencia de la secuencia aplicando promedios en una ventana de tiempo.

En lugar de usar un promedio simple en una ventana de tiempo, usamos pesos con decaimiento exponencial para el promedio y un estimado inicial v_0. Para estimar el t-ésimo elemento en la secuencia, usamos

$$v_t = \beta v_{t-1} + (1-\beta) y_t$$

$$v_t = \frac{v_t}{1-\beta^t}$$

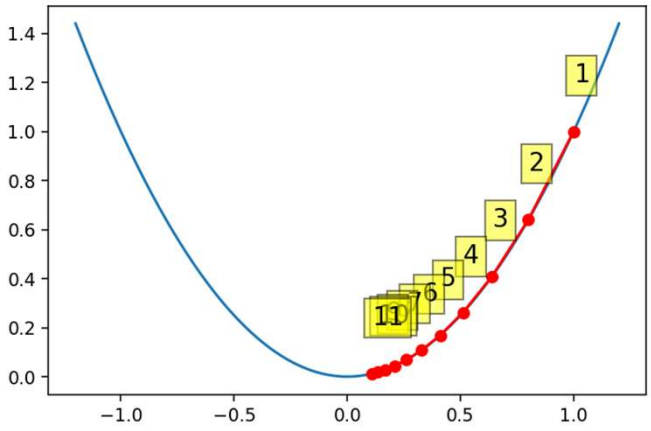
Bias correction

Parámetro $\beta < 1$ es el factor para el promedio ponderado

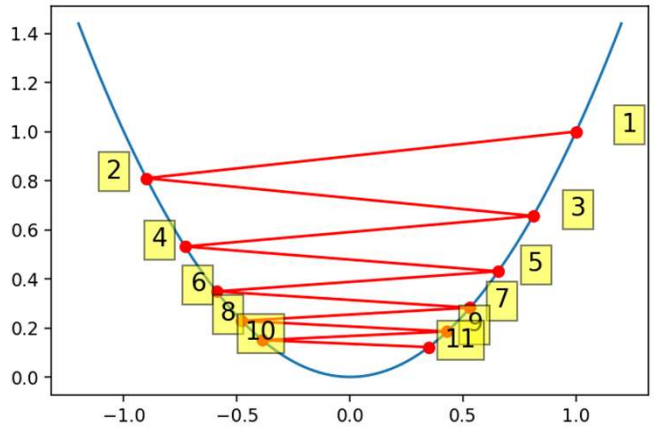
Gradiente descendente con momento

Tratamos de optimizar la función de costo con gradiente descendente, el vector gradiente es la dirección para alcanzar el mínimo.
Un learning rate grande puede producir oscilaciones.

$\eta = 0.1$



$\eta = 0.95$



Observación: vectores gradientes forman una secuencia que varía en el tiempo.

Momentum!

Usar exponentially weighted average sobre gradientes para suavizar transiciones

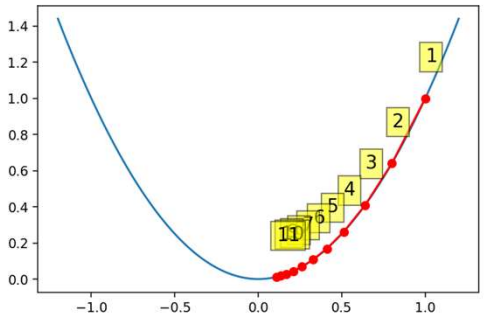
Gradiente descendente con momento

La regla de actualización ahora es

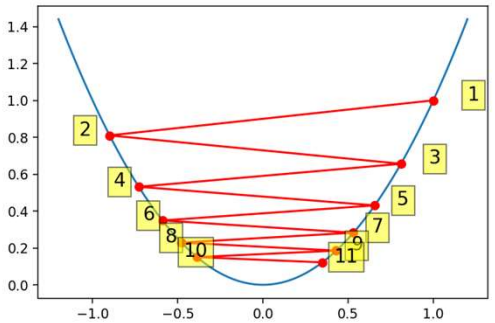
$$v_t = \beta v_{t-1} + (1-\beta)\frac{\partial C}{\partial w}$$

$$v_t = \frac{v_t}{1-\beta^t}$$

$$w = w - \eta v_t$$



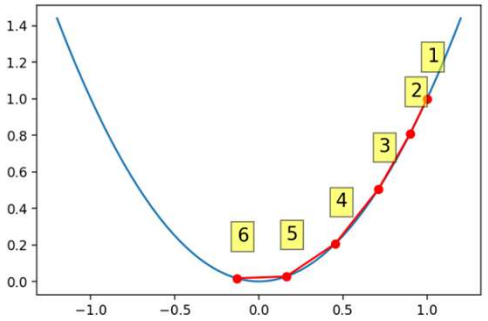
$\eta = 0.1$



$\eta = 0.95$

Momentum converge más rápido

In la práctica, parámetro β es 0.9



$\eta = 0.95$

RMSProp

Idea: Realizar exponentially weighted average sobre la magnitud de los gradientes en lugar de los gradientes

$$v_t = \beta v_{t-1} + (1-\beta)\left(\frac{\partial C}{\partial w}\right)^2$$

$$w = w - \frac{\eta}{\sqrt{v_t + \epsilon}}\frac{\partial C}{\partial w}$$

Cada parámetro tiene una variable caché que registra el promedio

Adam (Adaptive Moment Estimation)

Combinación de Momentum + RMSProp

- Exponentially weighted average para gradientes
- Exponentially weighted average para las maagnitudes
- Bias correction

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial C}{\partial w}$$

$$\hat{m}_t = \frac{m_t}{1-{\beta_1}^t}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial C}{\partial w}\right)^2$$

$$\hat{v}_t = \frac{v_t}{1-{\beta_2}^t}$$

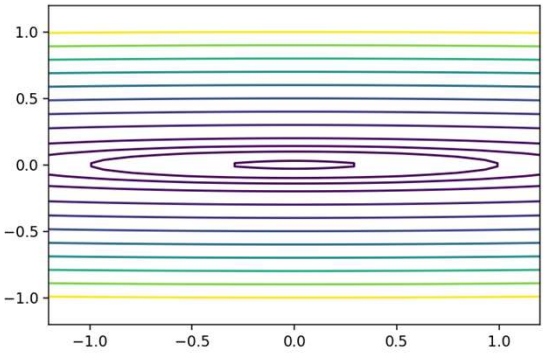
$$w = w - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

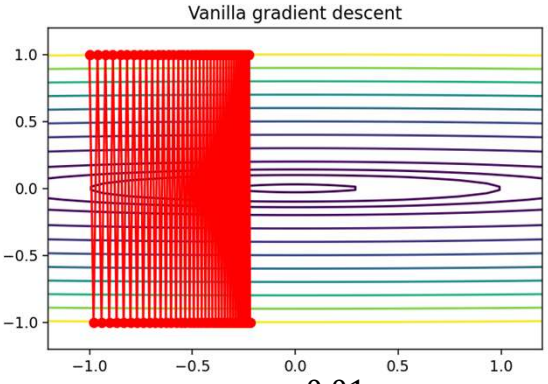
Hyper-parameters típicos

- $\beta_1 = 0.9$
- $\beta_2 = 0.999$

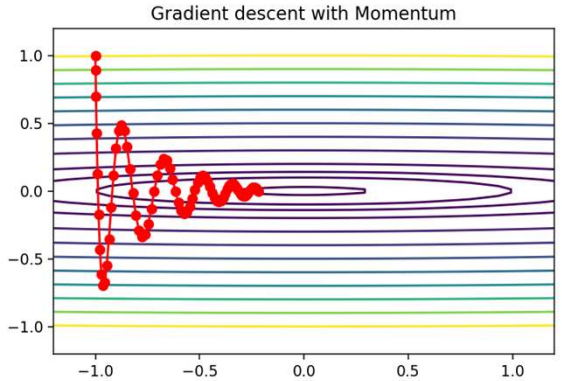
Comparación de algoritmos

Definimos un dominio de optimización 2D y visualizamos los contornos de nivel

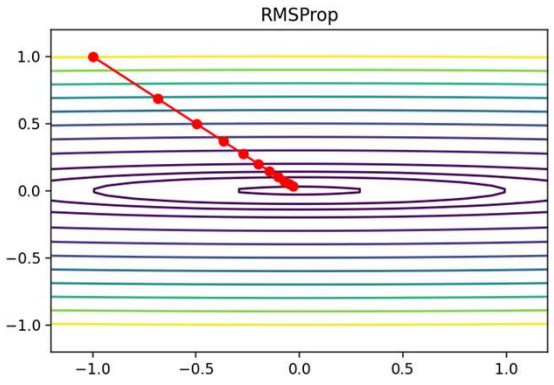




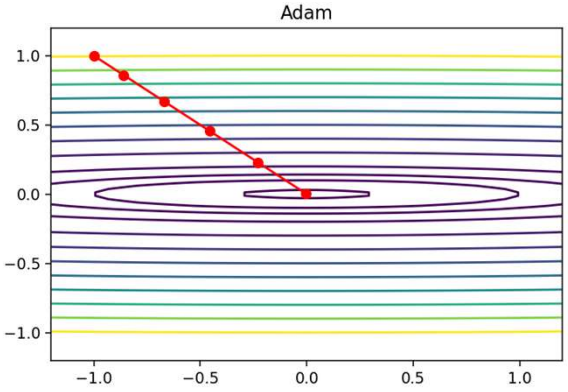
$\eta = 0.01$



$\eta = 0.01$



$\eta = 0.1$



$\eta = 0.1$

Regularización

El problema de overfitting

Con cuatro parámetros puedo crear un elefante, y con cinco puedo hacer que mueva su trompa.
John von Neumann

- Un número grande de parámetros pueden describir lo que sea.
- La meta no es memorizar la data, sino generalizar a nuevos ejemplos
- A Von Neumann le preocupaban los modelos que tenían cuatro parámetros. No le digamos que ChatGPT tiene miles de millones!

Estrategias para evitar overfitting

- Incrementar la cantidad de datos
  - Más data: variabilidad y reducción de memorización.
  - No obstante, en la práctica, suele ser lo más complicado.
- Reducir el tamaño de la red
  - Podría ser una contradicción, queremos redes grandes por su potencial y capacidad

- Cómo evitamos overfitting sin tener más datos y manteniendo el tamaño del modelo?

Regularización

Regularización L_2

- También conocido como weight decay
- Añade un término extra a la function loss

$$\hat{C} = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

Donde

- C_0 es la function los original (MSE ó cross-entropy loss)
- λ es el parámetro de regularización
- n es el tamaño del mini-batch

Regularización L_2

$$\hat{C} = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

- Hacer que la red aprenda pesos pequeños
- La derivada es simple, no mayor costo extra

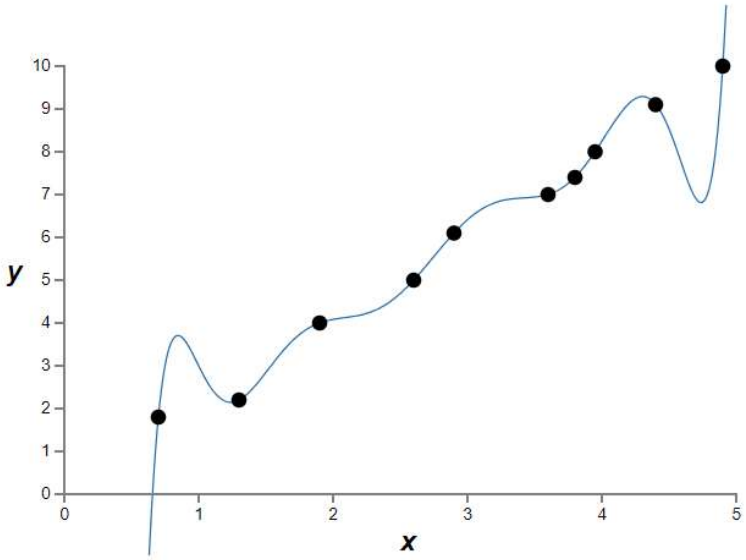
$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$

- Regla de actualización

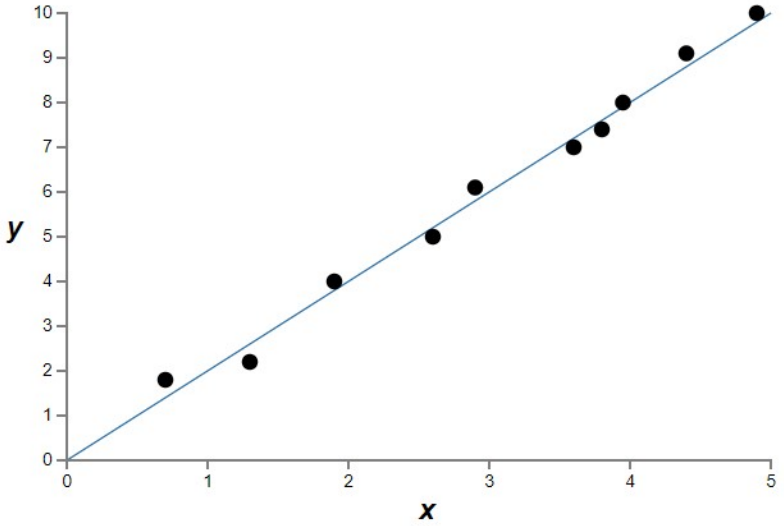
$$w = w - \eta\left(\frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w\right) \quad \Longrightarrow \quad w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta\frac{\partial C_0}{\partial w}$$

Weight decay

Regularización L_2



Qué modelo describe mejor la data?

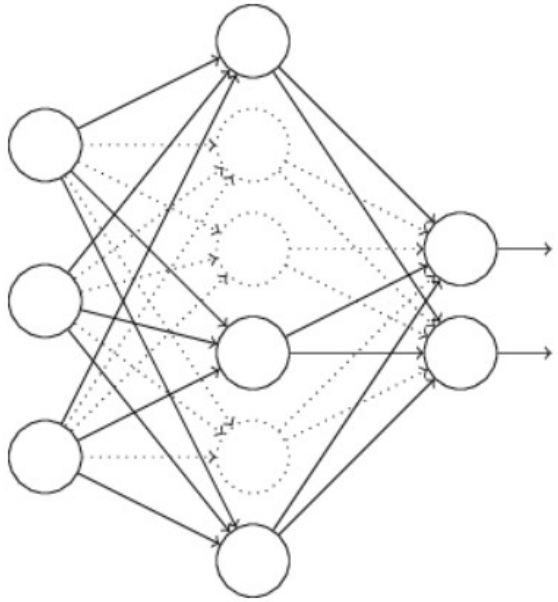


Para prediccicón, el modelo de la derecha es más resiliente a ruido

Weight decay ayuda a construir modelos más simples, reduciendo el overfitting

Dropout

- Estrategia simple
  - En cada ejecución de la red, desactivar algunas neuronas en las capas ocultas con propabilidad p



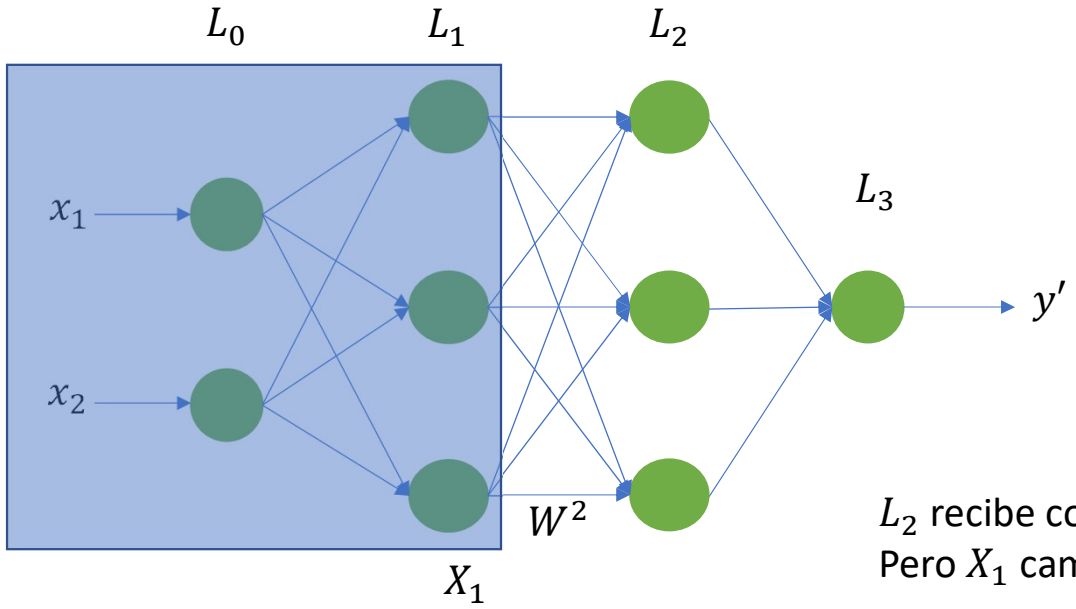
El efecto de desactivar aleatoriamente algunas neuronas es como promediar las respuestas de múltiples redes.
En cada ejecución estamos entrenando una red neuronal diferente.

Data augmentation

- Más data siempre reduce el overfitting, porque tenemos variabilidad.
- Conocimiento a priori puede ser usado para crear más data durante el entrenamiento.
- En el caso de imágenes, se aplican transformaciones
  - Pequeñas rotaciones
  - Flipping
  - Recortes aleatorios

Batch Normalization

Introducción



La computación en L_2 depende de

$$X_2 = \sigma(X_1 \cdot W^2 + b^2)$$

and

$$X_1 = \sigma(X_0 \cdot W^1 + b^1)$$

L_2 recibe como input X_1, así que L_2 aprende la distribución de X_1
Pero X_1 cambia durante el entrenamiento!

En cada época, una capa tiene que aprender una nueva distribución: covariate shift

La consecuencia es que la optimización converge lentaamente

Batch normalization

Batch normalization: Normalizar la activación de la capa previa a media cero y varianza uno

La nornmalización es en cada batch

Después de la normalización, es necesario darle a la red la oportunidad de corregir la distribución si es necesario

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch normalization

Con mejor convergencia, se puede incrementar el learning rate de forma segura

- Una red base tomó 30M iteraciones para alcanzar 72.2% de accuracy de clasificación

- BN con el mismo learning rate: 13M para llegar a 72.2%
- BN con 5*learning_rate: 2M para alcanzar 72.2%
- BN con 30*learning_rate: 2.7M para alcanzar 72.2%