

Redes Neuronales y Deep Learning

Inteligencia Artificial e Ingeniería del Conocimiento

Constantino Antonio García Martínez

Universidad San Pablo Ceu

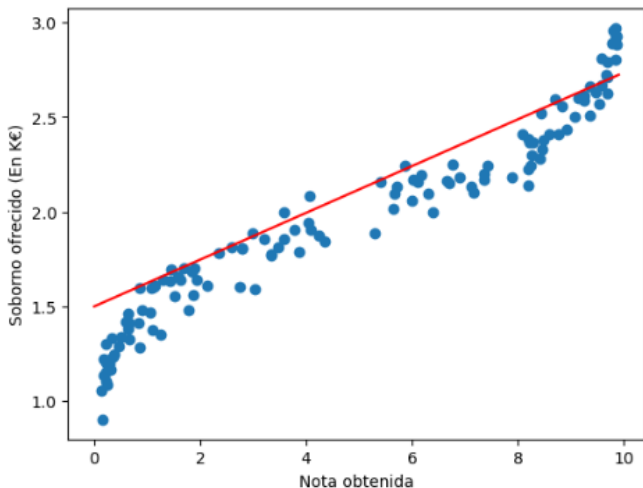
Introducción a Redes Neuronales y Deep Learning

Introducción a Redes Neuronales y Deep Learning

Redes Neuronales

Example: Sobornos

¿Cómo podríamos mejorar la salida del modelo lineal del problema de los sobornos?

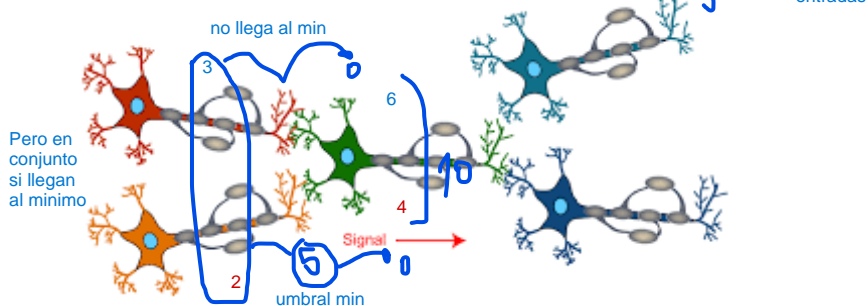


La limitación del modelo surge del hecho de que los **modelos lineales solo producen hiperplanos como salidas.** Nuestra predicción no se puede curvar

Solo genera planos/hiperplanos
Lineas rectas basicamente

Inspiración biológica: neuronas

Un sistema no lineal interesante es la neurona:



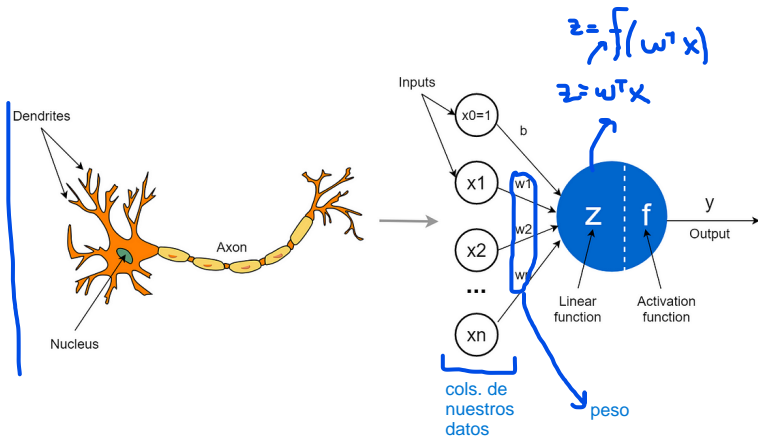
¡Observa la **activación no lineal**!

Nueronas que se conectan entre si
Vamos a replicar los impulsos electricos con un numero

Pero estamos mirnado por separado -> modelo lineal

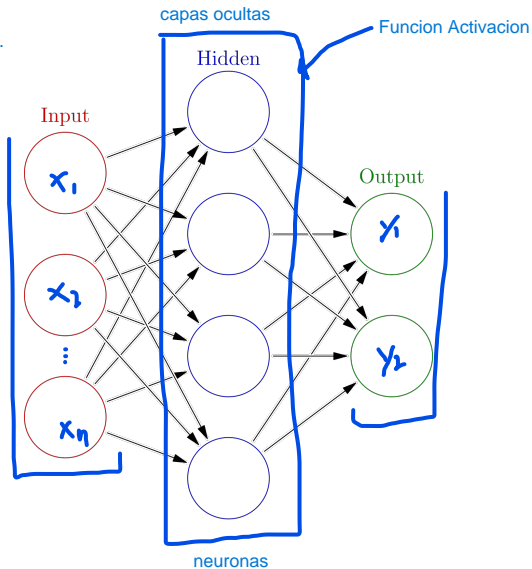
Pero las neuronas no emiezan adisparar impulsos hasta que recibe un minimo

Neuronas artificiales



Redes Neuronales

Conjunto de neuronas.
Conectadas todas entre si.



El problema de los sobornos y redes neuronales

Example: Sobornos

¿Cuál es la arquitectura de una red neuronal con una única **capa oculta** en el problema de los sobornos?

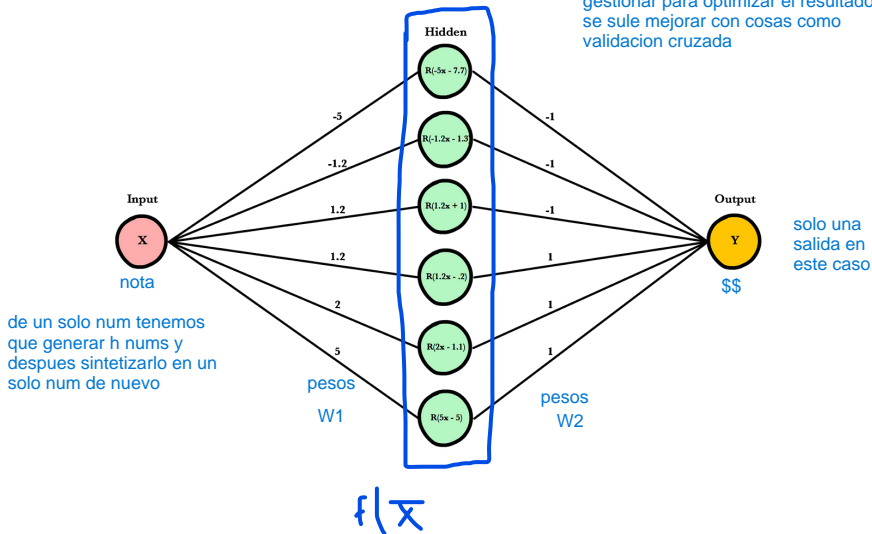
El problema de los sobornos y redes neuronales

Example: Sobornos

¿Cuál es la arquitectura de una red neuronal con una única **capa oculta** en el problema de los sobornos?

h neuronas

h es un hiperparametro que tienes que gestionar para optimizar el resultado. se suele mejorar con cosas como validacion cruzada



El problema de los sobornos y redes neuronales

Example: Sobornos

Para aplicar una red neuronal a nuestro problema de sobornos (versión de una variable) necesitamos resolver dos problemas:

1. ¿Cómo mapear nuestra entrada única a h capas ocultas?
2. ¿Cómo "comprimir" la salida de las neuronas para simular la activación no lineal?

Example: Sobornos

Para aplicar una red neuronal a nuestro problema de sobornos (versión de una variable) necesitamos resolver dos problemas:

1. ¿Cómo mapear nuestra entrada única a h capas ocultas?
2. ¿Cómo “comprimir” la salida de las neuronas para simular la activación no lineal?

El problema de los sobornos y redes neuronales

Example: Sobornos

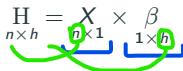
Para aplicar una red neuronal a nuestro problema de sobornos (versión de una variable) necesitamos resolver dos problemas:

1. ¿Cómo mapear nuestra entrada única a h capas ocultas?
2. ¿Cómo “comprimir” la salida de las neuronas para simular la activación no lineal?

1. El primer problema se resuelve mediante multiplicación matricial:

n notas, con matriz de $n \times 1$

multiplicación de matrices

$$H_{n \times h} = X_{n \times 1} \times \beta_{1 \times h}$$


matriz de notas por matriz de pesos

El problema de los sobornos y redes neuronales

Example: Sobornos

Para aplicar una red neuronal a nuestro problema de sobornos (versión de una variable) necesitamos resolver dos problemas:

1. ¿Cómo mapear nuestra entrada única a h capas ocultas?
 2. ¿Cómo “comprimir” la salida de las neuronas para simular la activación no lineal?
1. El primer problema se resuelve mediante multiplicación matricial:

$$\underset{n \times h}{H} = \underset{n \times 1}{X} \times \underset{1 \times h}{\beta}$$

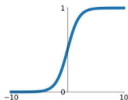
2. Simplemente aplicamos cualquier **función no lineal** a la salida de cada **neurona**. Esto se conoce como **función de activación**. Un ejemplo simple es la función sigmoide.

ReLU y Leaky ReLU son las dos mas usadas

Activation Functions

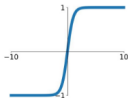
Sigmoid

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



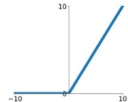
tanh

$$\tanh(x)$$



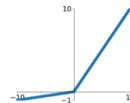
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

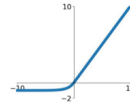


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



originales
pero
deprecated

Una función de activación que suele funcionar bien es **ReLU**.

La idea fundamental es comprimir la salida para emular el comportamiento de una neurona.

El problema de los sobornos y redes neuronales

Ya se hace la multiplicacion de matrices por detras con Keras

compuesta de


- TensorFlow
- JAX
- Pytorch

Code Example: NNet con Keras

```
model = keras.Sequential([  
    keras.layers.InputLayer(shape=(1,)), entrada con una sola col  
    keras.layers.Dense(512, activation='leaky_relu'), capa oculta con 512 neurona  
    keras.layers.Dense(1, activation=None), 1 salida, sin f de activacion  
])  
no usar f de act en  
salida es tipico en  
problemas de regresion
```

$$y \in (-\infty, \infty)$$

cunado la salida esta entre 0 y 1
se aplica el sigmoide



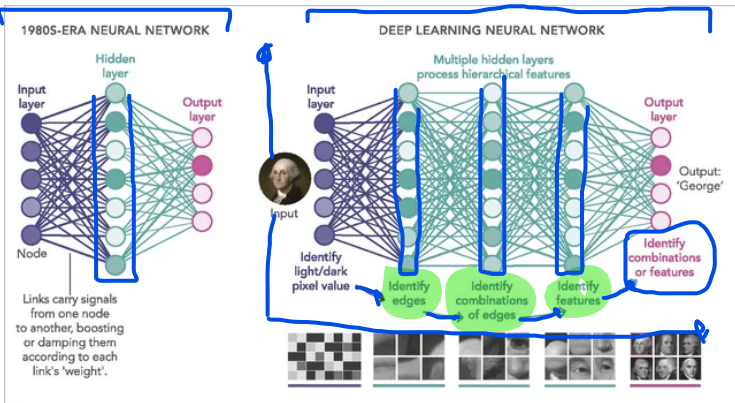
Introducción a Redes Neuronales y Deep Learning

Deep Learning

Redes Neuronales vs. Deep Learning

red neuronal es pequeña

muchas capas mas pequeñas que van haciendo cosas mas complicadas y sacando las características



xgboost gana cuando entradas son datos estilo csv.

neural gana cuando son datos raw, fotos audio, etc

Típicamente, las redes neuronales (y la mayoría de algoritmos de ML) requieren ingeniería de características para funcionar bien. ¡Deep Learning automatiza la extracción de características!

Deep learning crea capas muy profundas, aunque las capas sean menos anchas que las redes.

En las primeras capas intenta identificar figuras geométricas sencillas.

Después conjuntos

Capas finales más el aspecto

- ^{procesar imágenes} **Capas Convolucionales** (ConvNets o CNNs): Componente clave en CNNs para datos de imagen, aprende jerarquías espaciales.
- ^{secuencias como videos} **Capas Recurrentes** (LSTM, GRU): Usadas en RNNs para datos secuenciales, con celdas de memoria para retener información.
- **Capas Completamente Conectadas (Dense)**: Capa de propósito general donde cada neurona se conecta con todas las neuronas de la capa anterior.
- **Arquitecturas Populares:**
 - CNNs para imágenes y datos espaciales (ej., ResNet, VGG).
 - RNNs/LSTMs para datos secuenciales (ej., texto, series temporales).
 - **Transformers** para self-attention y modelos de lenguaje grandes (LLMs)

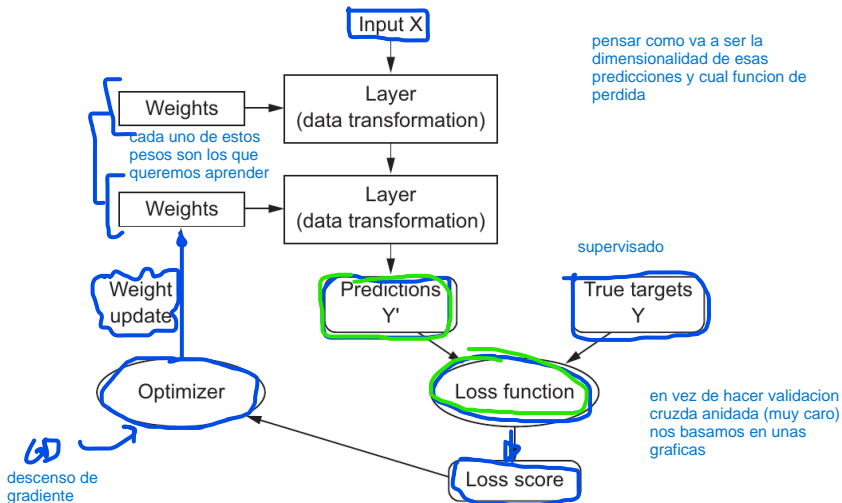
han jubilado a las capas recurrentes y han revolucionado chatbots y esas cosas

Introducción a Redes Neuronales y Deep Learning

Ingredientes Clave del Deep Learning

Ingredientes Clave del Deep Learning

Practicamente identico a regresion logistica



- **Regresión:**
 - **Dimensión de salida es 1.** *neurona de salida unica*
 - **Sin Activación:** La salida coincide directamente con los valores finales del modelo.
 - **Error Cuadrático Medio (MSE):** *normalmente usamos error cuadratico medio*

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Función de Pérdida y Diseño de la Capa de Salida

- **Clasificación Binaria:**

- **Dimensión de salida es 1.** Probabilidad de la clase positiva (la decidimos nosotros)
- **Activación Sigmoid:** Mapea salidas a probabilidades entre 0 y 1:

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

- **Entropía Cruzada Binaria:**

$$\text{BCE} = -\frac{1}{N} \sum_{i=1}^N (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

- **Clasificación Multiclase:**

- **Dimensión de salida es C.** MNIST(10) -> porque hay 10 dígitos
- **Activación Softmax:** Normaliza salidas a probabilidades de clase:

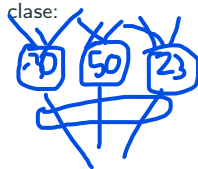
$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- **Entropía Cruzada Categórica:**

$$\text{CCE} = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c})$$

se usaba sigmoide

3 mascotas
[perro: 0.2, gato: 0.5, rata: 0.3]
(probabilidades)

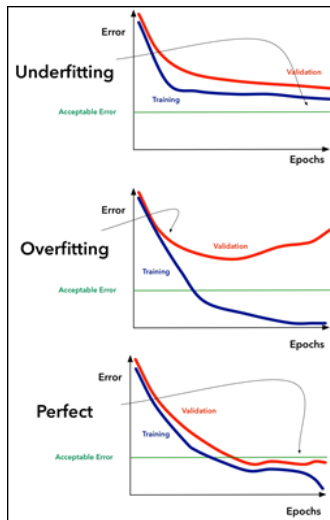


Monitorización de la convergencia

Medimos para cada ciclo de aprendizaje cual es el error.

Para el conjunto de entrenamiento y de test.

es importante monitorizar estas graficas porque no usamos validacion anidada, asi que las miramos y vamos ajustando



cuando el modelo es demasiado sencillo. los dos conjuntos estan lejos del valor deseado

memoriza en vez de aprender, cuando cambia la entrada no sabe. cuando es demasiado complicado

Code Example: Problema de sobornos con redes neuronales

Entrenamiento de redes neuronales

Entrenamiento de redes neuronales

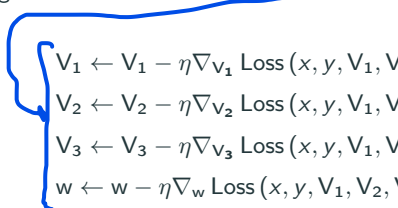
Grafos computacionales, AD y Backprop

Motivación: regresión con red neuronal de cuatro capas

Pérdida en un ejemplo:

$$\text{Loss}(x, y, \overset{\text{pesos}}{V_1, V_2, V_3}, w) = \underbrace{(w \cdot \sigma(V_3 \sigma(V_2 \sigma(V_1 \phi(x)))) - y)^2}_{\text{derivadas}}$$

Descenso de gradiente:


$$\begin{cases} V_1 \leftarrow V_1 - \eta \nabla_{V_1} \text{Loss}(x, y, V_1, V_2, V_3, w) \\ V_2 \leftarrow V_2 - \eta \nabla_{V_2} \text{Loss}(x, y, V_1, V_2, V_3, w) \\ V_3 \leftarrow V_3 - \eta \nabla_{V_3} \text{Loss}(x, y, V_1, V_2, V_3, w) \\ w \leftarrow w - \eta \nabla_w \text{Loss}(x, y, V_1, V_2, V_3, w) \end{cases}$$

¿Cómo obtener el gradiente sin hacer el trabajo manual?

$$\text{Loss}(x, y, V_1, V_2, V_3, w) = (w \cdot \sigma(V_3 \sigma(V_2 \sigma(V_1 \phi(x)))) - y)^2$$

Definición: grafo computacional

Un grafo acíclico dirigido cuyo nodo raíz representa la expresión matemática final y cada nodo representa subexpresiones intermedias.

Objetivo: Calcular automáticamente gradientes mediante el algoritmo general de **backpropagation** (así funcionan PyTorch, TensorFlow y Jax). Las implementa Keras la mas usada

```
import torch
from torch.autograd import Variable

# Let's demonstrate the calculation of a simple derivative:  $y = x^2$ 
x = Variable(torch.tensor([3.0]), requires_grad=True)
y = x**2
y.backward()

# The derivative of y with respect to x is  $dy/dx = 2x$ 
print(f"Theoretical gradient: {2 * x.data}")
# Output: Theoretical gradient: tensor([6.])
print(f"Gradient from autograd: {x.grad}")
# Output: Gradient from autograd: tensor([6.])
```

20 img x 28px x 28px
20vid x 28 x 28

Keras oculta todo esto

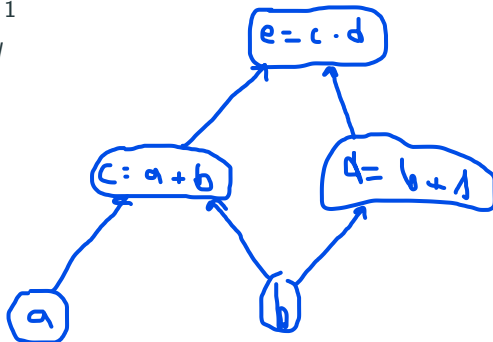
x vale 3

$dy/dx = 2x = 2 \times 3 = 6$

Exercise: Construye el grafo computacional para...

- a y b son parámetros aprendibles.

$c = a + b$
 $d = b + 1$
 $e = c \cdot d$



Exercise: Construye el grafo computacional para...

- a y b son parámetros aprendibles.
- $c = a + b$
- $d = b + 1$
- $e = c \cdot d$

Exercise: Evalúa el grafo si $a=2$ y $b=1$

Exercise: Construye el grafo computacional para...

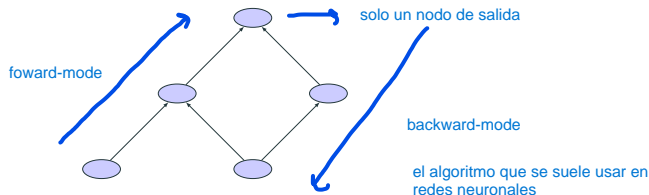
- a y b son parámetros aprendibles.
- $c = a + b$
- $d = b + 1$
- $e = c \cdot d$

Exercise: Evalúa el grafo si $a=2$ y $b=1$

Podemos evaluar expresiones en un **forward pass** pero, ¿cómo calculamos eficientemente las derivadas?

Diferenciación forward- y backward-mode

La **diferenciación forward-mode** y la **diferenciación backward-mode** (o backward-mode) son algoritmos para calcular derivadas eficientemente:



- **Backward-mode** es el algoritmo comúnmente usado en redes neuronales.
- **Backpropagation** es una aplicación especializada de backward-mode AD, adaptada para arquitecturas de redes neuronales:
 - **Backprop** calcula las derivadas de la función de pérdida respecto a todos los parámetros.
 - **Backward-mode AD** puede calcular derivadas de cualquier proceso computacional.

¿Por qué Backward-mode?

Exercise: Evalúa las derivadas usando forward-mode

Exercise: Evalúa las derivadas usando backward-mode

¿Por qué Backward-mode?

Exercise: Evalúa las derivadas usando forward-mode

Exercise: Evalúa las derivadas usando backward-mode

- En **forward-mode** obtenemos la **derivada de todas las salidas respecto a una entrada.** una entrada -> todas las salidas
- En **backward-mode** obtenemos la **derivada de una salida respecto a todas las entradas.** una salida -> todas las entradas
 - ¡Por eso **backward-mode** se usa típicamente en ML!
 - En el **forward pass** calculamos los valores de los nodos, y en el **backward pass** calculamos las derivadas.

Hace falta hacer un forward-mode para calcular los valores de los nodos y después un backward-mode para calcular las derivadas.



La pega de usar esto es que gasta mucha memoria, no como hacerlo manualmente. También tiene un coste computacional muy grandes.

En cosas como regresiones lineales puede valer la pena hacerlo a mano. En redes neuronales es imposible

Exercise: Red neuronal de dos capas

Supongamos un único ejemplo de entrenamiento en un problema de regresión.

La función de pérdida es MSE y la red neuronal se define como

$y_{out} = \mathbf{w} \cdot \sigma(\mathbf{V}\phi(x))$. Dibuja el grafo computacional y calcula los gradientes usando backprop.

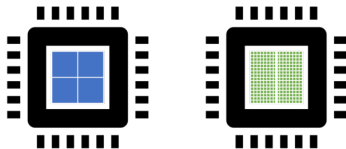
$$\frac{d}{dx} (\sigma(x)) = \sigma(x) (1 - \sigma(x))$$

$\phi(x)$

Entrenamiento de redes neuronales

**Infraestructura Moderna de Entrenamiento:
GPUs**

CPUs vs. GPUs



CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

RAM

VRAM

- **¿Por qué GPUs?**

- **Alto Paralelismo:** Las redes neuronales involucran grandes multiplicaciones matriciales, que pueden paralelizarse efectivamente.
- **Alto Rendimiento:** Las GPUs procesan miles de cálculos simultáneamente, ideal para las necesidades computacionales del deep learning.
- **Hardware Especializado:** Las GPUs están diseñadas para manejar operaciones matemáticas repetitivas más eficientemente que las CPUs.

lenguaje
para gpus

- **CUDA (Compute Unified Device Architecture):**

Plataforma de NVIDIA que permite el control directo de operaciones GPU, usando *kernels personalizados* y *gestión eficiente de memoria*.

- **Aceleración con CUDA:**

Bibliotecas como **cuDNN** proporcionan rutinas optimizadas (ej., convoluciones) para deep learning, reduciendo significativamente el tiempo de entrenamiento.

- **Beneficios:**

Entrenamiento más rápido, menor latencia en inferencia y escalabilidad para grandes conjuntos de datos.

librerías
especializadas
para gpus

Entrenamiento de redes neuronales

Mini-batch SGD y Optimizadores

Mini-batch Stochastic Gradient Descent

toda esa info no cabe en vram ni en la memoria de un ordenador

- **Restricciones de memoria en redes neuronales**/Deep Learning:
 - **Parámetros del modelo**
 - **Activaciones** (calculadas en el **forward pass** y necesarias para backprop)
 - **Gradientes** (backward pass)
- Enfoques de entrenamiento:
 - **Gradient Descent** (GD, también llamado **Batch GD**): Todos los datos a la vez \Rightarrow Imposible con datos grandes y redes grandes.
 - **Stochastic GD (SGD)**: **Una muestra a la vez** \Rightarrow reduce el uso de memoria pero aumenta la varianza en las estimaciones del gradiente.
 - **Mini-batch SGD**: Lo mejor de ambos mundos.

No usa todos, ni de uno en uno, usa los ejemplos que le quepan en la gpu. Cada uno de estos grupos se les llama mini-batch.

- **Procesa un subconjunto** (mini-batch) de datos en cada paso, permitiendo actualizaciones más estables que SGD usando menos memoria que GD completo.
- **Tamaños típicos de batch son 32, 64, 128, o hasta 256**, dependiendo del tamaño del modelo y la memoria disponible.

Si peta la mem a lo mejor deberías bajar el batch.

```
optimizer.SGD(...)  
model.fit(..., batchsize = 32,...)
```

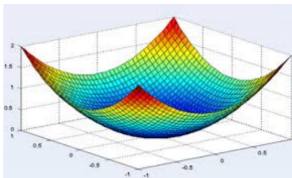
Entrenamiento de redes neuronales

Inicialización de Pesos, Optimizadores

La Optimización de redes neuronales es Difícil

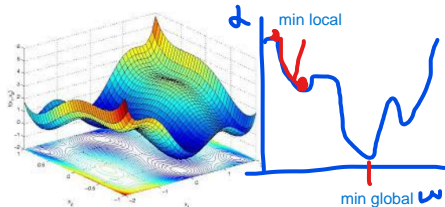
Si tenemos algoritmos lineales, si calc su f de pérdida, siempre tiene forma de bowl. Los pesos bajan perfectamente y predeciblemente.

Linear predictors



(convex)

Neural networks



(non-convex)

Las redes neuronales no...

no hay garantía de encontrar el min global

La optimización es difícil:

- La optimización de redes neuronales es no convexa, lo que lleva a múltiples mínimos locales y puntos de silla.
- Gradientes que Desaparecen o Explotan:** Los gradientes pueden volverse demasiado pequeños (desaparecen) o demasiado grandes (explotan), ralentizando o desestabilizando el entrenamiento.

calculas las derivadas y se va a infinito

La Optimización de redes neuronales es Difícil

$$y = w \cdot x$$



peso inicial es aleatorio, pero hay que ver como inicializarlo mejor

Las redes neuronales fueron abandonadas por un tiempo hasta que se desarrollaron varias técnicas para entrenar modelos profundos:

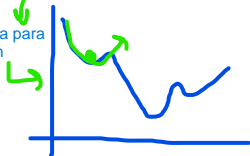
1. Mejor Inicialización

- Xavier/Glorot (Relus)
- He initialization

2. Optimizadores Avanzados

- SGD con Momentum
- Métodos adaptativos como Adam

meter inercia para que 'puedan saltar'



hacer que tengan diferentes tasas de aprendizaje

misma tasa de aprendizaje

$$v_1 \leftarrow v_1 - \eta \nabla_{v_1} \mathcal{L}$$
$$v_2 \leftarrow v_2 - \eta \nabla_{v_2} \mathcal{L}$$

adam los separa

Exercise: Inicialización en Keras

Lee la API de Keras para aprender cómo usar diferentes métodos de inicialización en las capas Dense. ¿Cuál es el método por defecto?

Code Exercise: Cambiando arquitectura, optimizador e hiperparámetros en el problema de los sobornos

Por culpa de que las redes neuronales usan CPUs, hay tamaños de memoria limitados, así que entrenamos en mini-batches

No solemos usar descenso de gradiente, otros optimizadores más sofisticados.
Como ADAM, que cada parámetro tenga su learning rate.

Entrenamiento de redes neuronales

Regularización y Estabilidad

Penalizar que el modelo sobre-aprenda

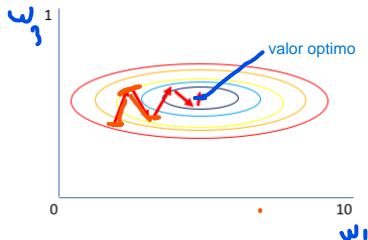
Que sean más estables lol

Técnicas de estabilización: Normalización de entradas

Siempre debemos normalizar los datos:
Porque afecta al calculo de los gradientes

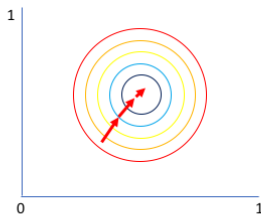
x_1	x_2	x_3
-100	0.01	0
100	0.01	1
	0.1	2

Why normalize?



Gradient of larger parameter
dominates the update

La pelotita va zigzageando, Y basicamente va rebotando por la 'tuberia', Muy poco optimo



Both parameters can be
updated in equal proportions

Es como un cuenco, y facilita mucho la optimizacion

Solo necesario en las columnas de
modelos que se basen en descenso
de gradiente

Por ejemplo a Arboles (y subcategorias de arboles), no les afecta
mucho, pero tampoco hace mal, asi que mejor hacer siempre

Técnicas de estabilización: Batch Normalization

Después de muchas iteraciones de capas, se puede perder la normalización, así que se meten capas intermedias que reciben las salidas de la capa anterior y las normaliza de nuevo (capas verdes)

- Normaliza la entrada a una capa de la red.
- **Beneficios:**
 - Acelera el entrenamiento y mejora la convergencia del modelo.
 - Permite tasas de aprendizaje más altas, mitigando problemas como gradientes que desaparecen/explotan.
- Dos pasos principales:
 1. **Normalización:** Para una entrada x :

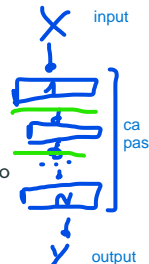
$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

donde: μ es la media del mini-batch, σ^2 es la varianza del mini-batch, ϵ es una pequeña constante para estabilidad numérica.

2. **Escalado y Desplazamiento:** Después de la normalización, las salidas se escalan y desplazan para permitir que el modelo aprenda la representación óptima:

$$y = \gamma \hat{x} + \beta$$

donde: γ es el parámetro de escala, β es el parámetro de desplazamiento.

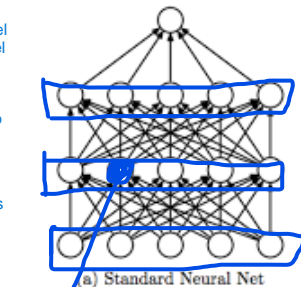


Técnicas de Regularización: Dropout

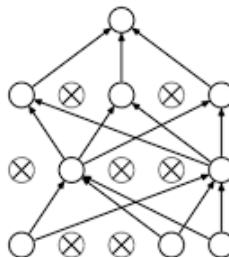
- **Regularización de Pesos:** Lasso, Ridge, ElasticNet
 - **Regularización L1:** Fomenta la dispersión penalizando la suma absoluta de pesos. Normalmente en DL se usa L2, si se usa Reg de Pesos
 - **Regularización L2:** Añade una penalización proporcional al cuadrado de los pesos, fomentando pesos más pequeños (**weight decay**).
- **Dropout:**
 - Desactiva aleatoriamente neuronas durante el entrenamiento para reducir el sobreajuste.
 - Ayuda a prevenir la co-adaptación de neuronas, haciendo la red más robusta.

Hay neuronas que si el resto están haciendo el trabajo, no aprenden nada, así que no son útiles (se basan demasiado en el trabajo de sus compañeros).

En el entrenamiento 'matamos' a diferentes neuronas. Esto obliga a que la neurona vaga se 'ponga las pilas', las fuerza a trabajar



(a) Standard Neural Net



(b) After applying dropout.

Un ej específico de Dropout.
Ej: iteración 3.

En diferentes iteraciones, se van cambiando de manera aleatoria

El único parámetro que tiene es el como de agresivo queremos ser con la desactivación de neuronas.
Probabilidad:
0 -> ninguna desactivada, 1 - todas desactivadas

$p(0 \text{ a } 1)$

vaga

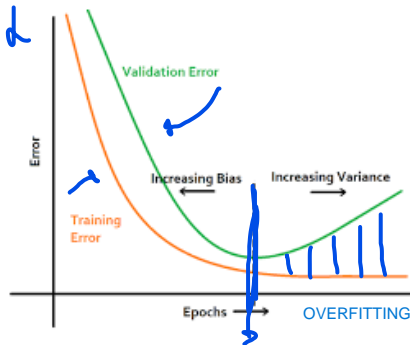
Técnicas de Regularización: Early Stopping

Se basa en analizar las curvas de entrenamiento.

- **Early Stopping:**

- Monitoriza el rendimiento de validación para detener el entrenamiento cuando comienza el sobreajuste.
- Ahorra tiempo de cómputo y previene la degradación del rendimiento del modelo.

Una buena forma de ver como va el entrenamiento es dibujar la curva de error. Esto lo hacemos para el conjunto de entrenamiento y de validación



Empieza a separarse el error de validación, mientras que el error de entrenamiento sigue bajando. Esto suele indicar overfitting,

Early stopping indica que dejemos de entrenar en ese punto.

Testear sobre Test

Aquí se para

Code Exercise: PetFinder.my

Code Exercise: MNIST