Aspectos Matemáticos y Computacionales del Aprendizaje Automático

Escuela de Otoño en Matemáticas para la Formación de Investigadoras e Investigadores

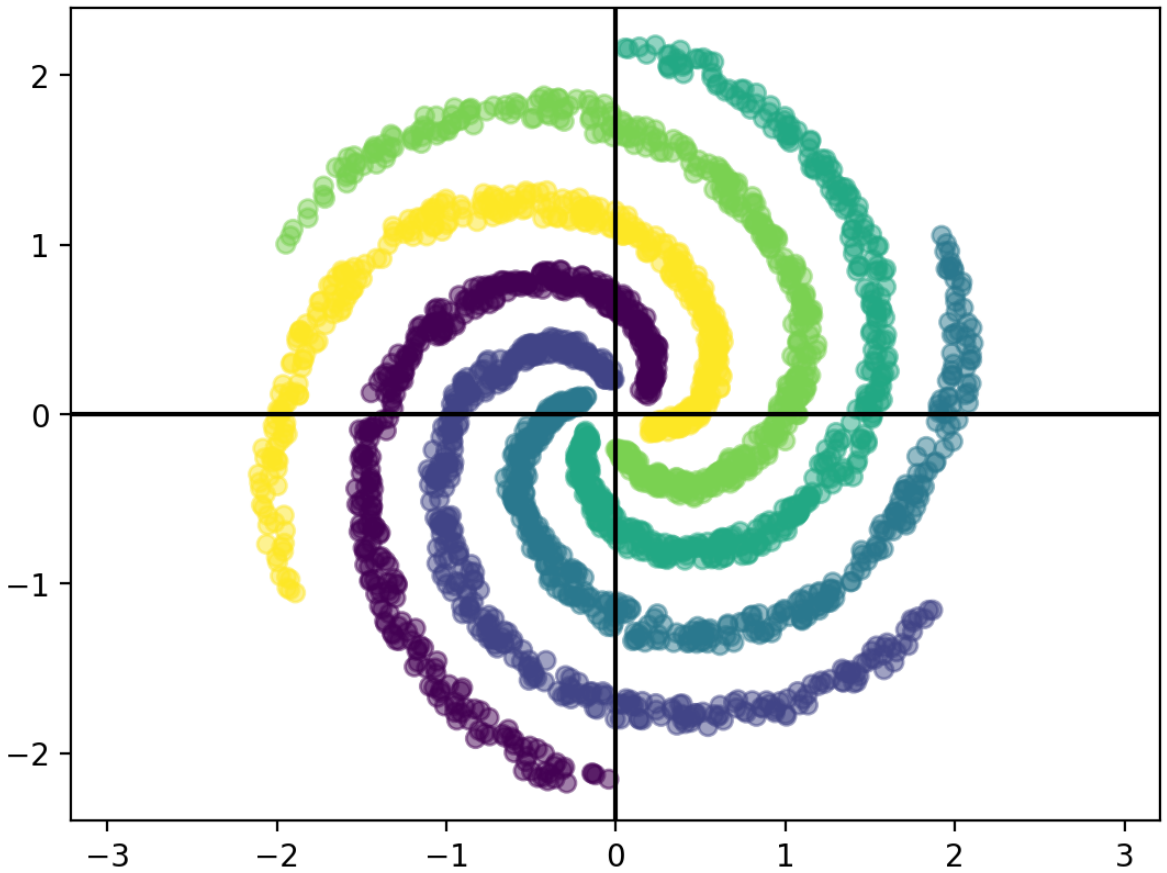
Mauricio Toledo-Acosta

Departamento de Matemáticas
Universidad de Sonora

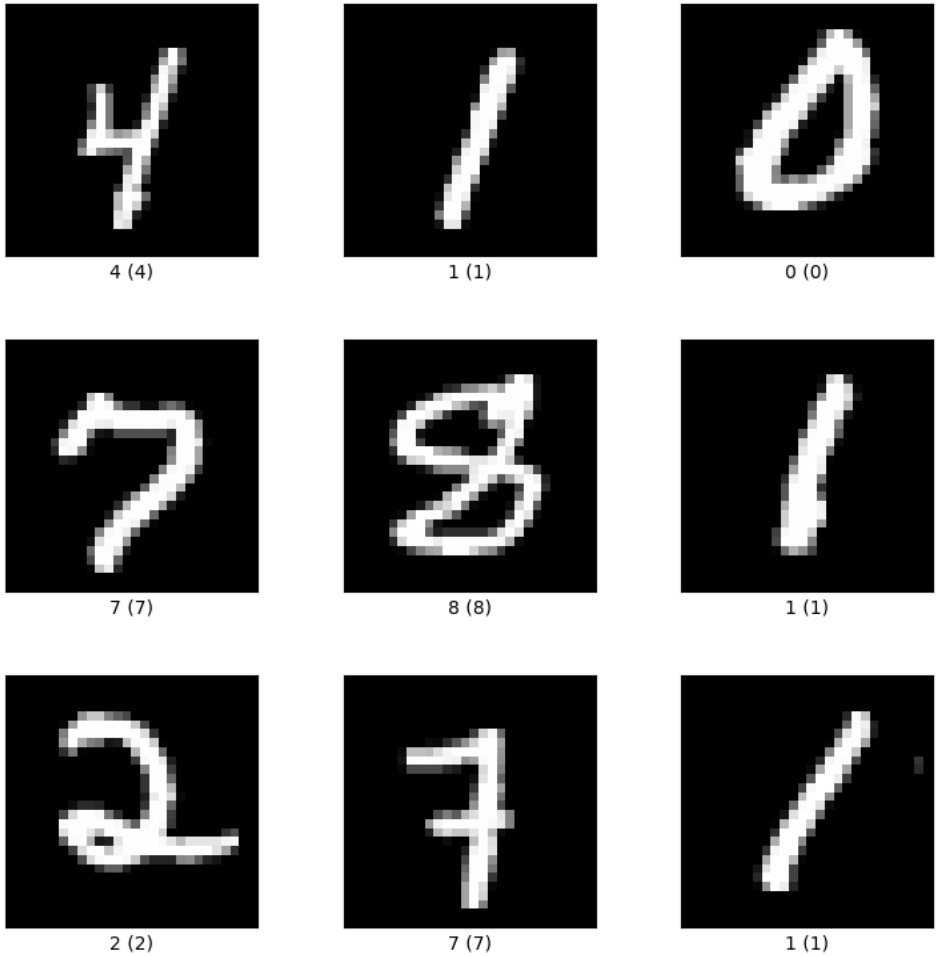
Repositorio del minicurso

Table of Contents

Datos: Puntos en $\mathbb{R}^D$



Datos: Imágenes



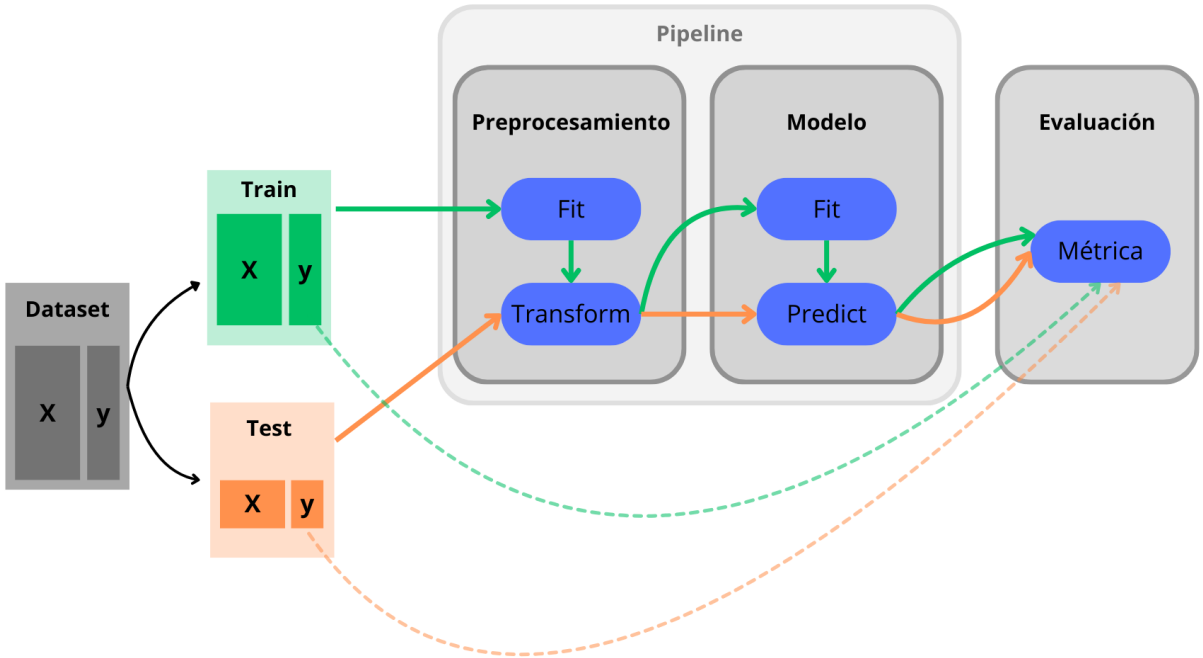
Datos: Texto


```
I saw this movie once a long time ago, just once, and I didn't know where it had...:
        positive
In the history of cinema, every great film-maker had to create a first film. Man...:
        positive
An excellent interpretation of Jim Thompson's novel, this neo-noir thriller has ...:
        positive
Sometimes when I hear an A-list cast will be bunched up together for 2 hours in ...:
        negative
I probably give this more credit than it deserves because it's Halloween, I was ...:
        negative
```

¿Dónde encontramos Deep Learning en la vida real?

- **Asistentes virtuales**:
  - LLMs: DeepSeek, Qwen, ChatGPT, Claude, etc.
  - Siri, Alexa (reconocimiento de voz y respuestas).
- **Redes sociales y fotos**:
  - Etiquetado automático de personas en fotos.
  - Filtros de cámara.
  - Sistemas de recomendación.
- **Seguridad y tecnología**:
  - Reconocimiento facial.
  - Detección de fraudes en tarjetas de crédito.
- **Transporte y mapas**:
  - Recomendación de rutas en Google Maps/Waze.
  - Coches autónomos.
- **Salud**:
  - Diagnóstico médico asistido.
  - Wearables.

El Aprendizaje Automático



Machine Learning y Matemáticas Teóricas

MACHINE LEARNING APPROACHES TO THE SHAFAREVICH-TATE GROUP OF ELLIPTIC CURVES

ANGELICA BABEI, BARINDER S. BANWAIT, AJ FONG, XIAOYU HUANG, AND DEEPENDRA SINGH

ABSTRACT. We train machine learning models to predict the order of the Shafarevich-Tate group Ш of an elliptic curve over $\mathbb{Q}$. Building on earlier work of He, Lee, and Oliver, we show that a feed-forward neural network classifier trained on subsets of the invariants arising in the Birch–Swinnerton-Dyer conjectural formula yields higher accuracies (> 0.9) than any model previously studied. In addition, we develop a regression model that may be used to predict orders of Ш not seen during training and apply this to the elliptic curve of rank 29 recently discovered by Elkies and Klagsbrun. Finally we conduct some exploratory data analyses and visualizations on our dataset. We use the elliptic curve dataset from the *L-functions and modular forms database* (LMFDB).

Machine learning invariants of arithmetic curves

Yang-Hui He [a,b,c], Kyu-Hwan Lee [d], Thomas Oliver [e]

[a] *London Institute, Royal Institution, 21 Albemarle St, London W1S 4BS, UK*
[b] *Merton College, University of Oxford, OX14JD, UK*
[c] *School of Physics, NanKai University, Tianjin, 300071, PR China*
[d] *Department of Mathematics, University of Connecticut, Storrs, CT, 06269-1009, USA*
[e] *SCEDT, Teesside University, Middlesbrough, TS1 3BX, UK*

ARTICLE INFO

Article history:
Received 7 December 2021
Received in revised form 29 June 2022
Accepted 15 August 2022
Available online 22 August 2022

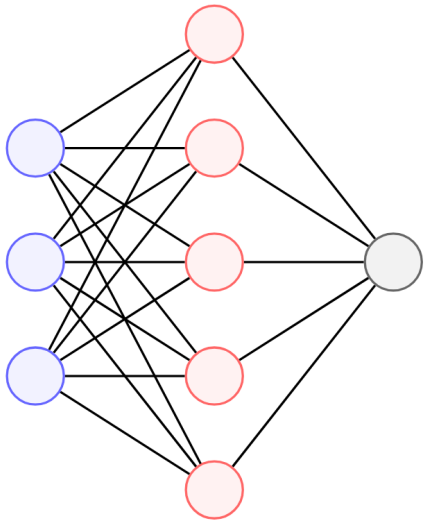
Keywords:
Machine-learning
Arithmetic geometry
Elliptic curves
Hyper-elliptic curves
Birch-Swinnerton-Dyer conjecture

ABSTRACT

We show that standard machine learning algorithms may be trained to predict certain invariants of low genus arithmetic curves. Using datasets of size around 10^5, we demonstrate the utility of machine learning in classification problems pertaining to the BSD invariants of an elliptic curve (including its rank and torsion subgroup), and the analogous invariants of a genus 2 curve. Our results show that a trained machine can efficiently classify curves according to these invariants with high accuracies (> 0.97). For problems such as distinguishing between torsion orders, and the recognition of integral points, the accuracies can reach 0.998.

Table of Contents

¿Qué es una red neuronal?



Una red neuronal es una *función* que tiene una propensión natural para almacenar conocimiento experiencial y hacerlo disponible para su uso. Se asemeja al cerebro en dos aspectos:

1. El conocimiento es adquirido por la red a través de un proceso de aprendizaje.
2. Las fuerzas de conexión entre neuronas, conocidas como pesos sinápticos, se utilizan para almacenar el conocimiento.

S. Haykin (1994), Neural Networks, MacMillan

Entrenamiento de una red neuronal

En estos problemas a modelar tenemos un conjunto de entradas $x = \{x_1, \ldots, x_n\} \in \mathbb{R}^D$ y algún proceso que resulta en un conjunto correspondiente de salidas $y = (y_1, \ldots, y_m) \in \mathbb{R}^m$.
La hipótesis es que el proceso está dado por alguna función matemática, es decir,

$$y = G(x)$$

para alguna función G que puede ser muy complicada.

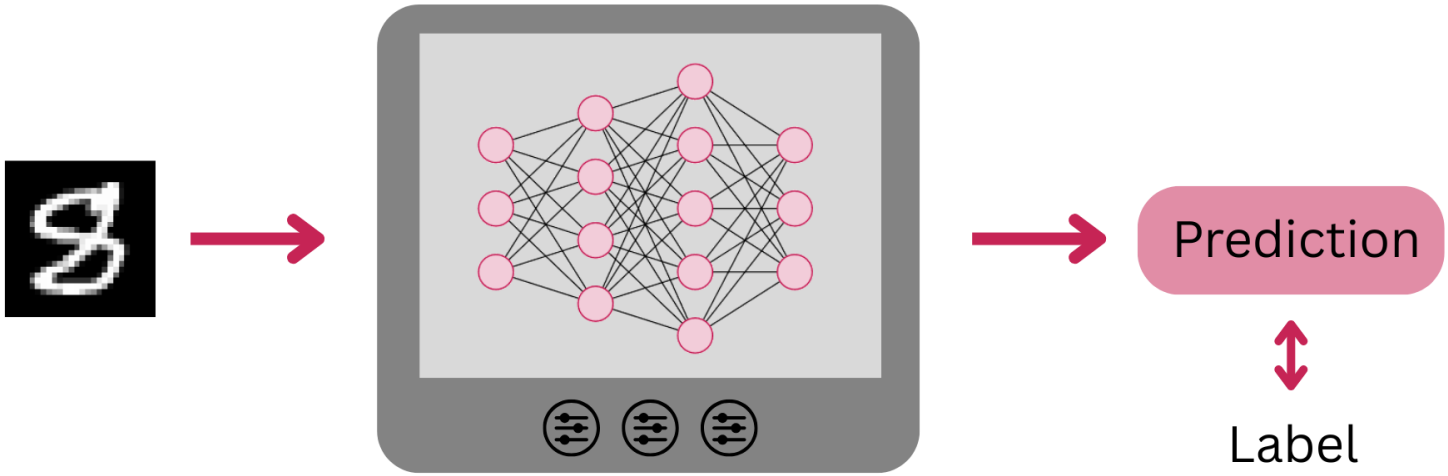
Entrenamiento de una red neuronal

Para aproximar la función G elegimos una función *candidata* F de algún conjunto parametrizado de funciones. Encontramos los parámetros de F usando un conjunto dado de ejemplos, es decir, algunas entradas x y salidas *correctas* asociadas $y = G(x)$.

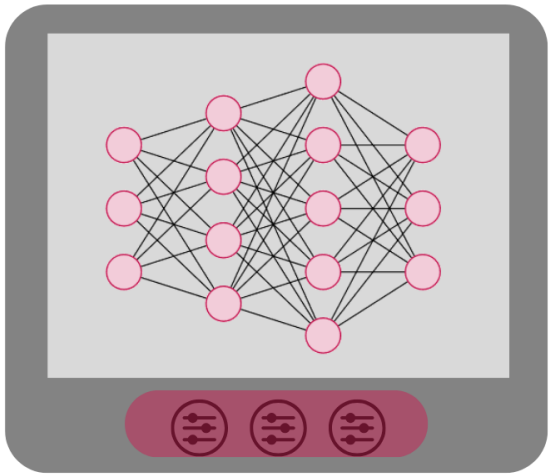
Entrenar una red neuronal significa encontrar estos parámetros. El entrenamiento se compone de dos pasos:

1. **Feed Forward**: Producir salidas con F usando algunos parámetros.
2. **Back Propagation**: Ajustar los parámetros midiendo el error de las salidas contra las salidas reales.

1. Feed Forward



2. Back Propagation



Prediction

Label

El perceptron: una neurona

Un perceptrón o neurona es una función $P : \mathbb{R}^D \to \mathbb{R}$ dada por

$$P(x_1, ..., x_D) = f\left(w_0 + \sum_{i=1}^{D} w_i x_i\right)$$

donde $f : \mathbb{R} \to \mathbb{R}$ es continua, llamada función de activación y $w_0, ..., w_D \in \mathbb{R}$ son pesos. A w_0 se le llama sesgo o bias. Podemos reescribir como

$$P(x_1, ..., x_D) = f\left(\begin{pmatrix} w_0 & w_1 & ... & w_D \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ ... \\ x_D \end{pmatrix}\right)$$

Funciones de activación

- Sigmoide:

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

- Tangente hiperbólica:

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- ReLU (Rectified Linear Unit):

$$f(x) = \begin{cases} x, & \text{si } x \geq 0, \\ 0, & \text{si } x < 0. \end{cases}$$

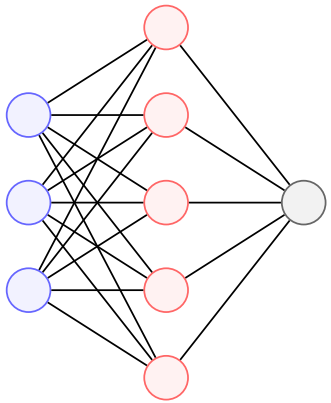
Capas de neuronas

Una capa de m neuronas es una función $P : \mathbb{R}^D \to \mathbb{R}^m$:

$$P(x_1, ..., x_D) = f\left(W \cdot \tilde{x}\right)$$

$$= f\left(\begin{pmatrix} w_{1,0} & w_{1,1} & \dots & w_{1,D} \\ w_{2,0} & w_{2,1} & \dots & w_{2,D} \\ \dots & \dots & \dots & \dots \\ w_{m,0} & w_{m,1} & \dots & w_{m,D} \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ \dots \\ x_D \end{pmatrix}\right)$$

$W \in \mathcal{M}_{m\times(D+1)}(\mathbb{R})$ son los pesos $w_{i,j}$ y sesgos $w_{i,0}$. La función $f : \mathbb{R}^m \to \mathbb{R}^m$ aplica la misma función de activación en cada una de las m componentes de $W \cdot \tilde{x}^T$.

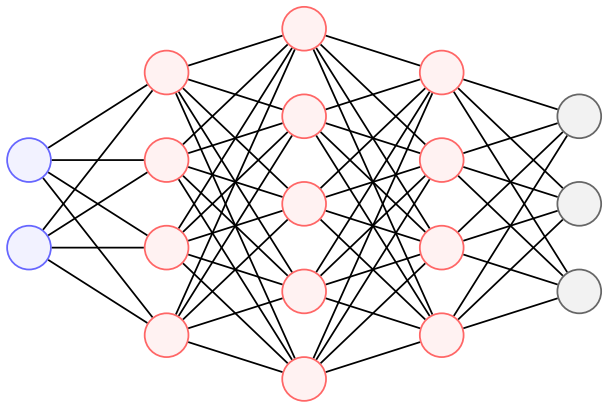


Varias capas de neuronas

Podemos apilar L capas de $m_1, \ldots, m_L$ neuronas respectivamente:

1. La primera capa oculta es una función $P_1 : \mathbb{R}^{m_0} \to \mathbb{R}^{m_1}$ donde $m_0 = D$.
2. Continuamos de forma análoga hasta llegar a la L-ésima capa oculta, que es una función $P_L : \mathbb{R}^{m_{L-1}} \to \mathbb{R}^{m_L}$.

Las capas $P_1, ..., P_L$ se denominan capas ocultas. Apilamos una capa de salida $P_{L+1} : \mathbb{R}^{m_L} \to \mathbb{R}^{m_{L+1}}$. Este ensamble de capas es una red MultiLayer Perceptron (MLP) con L capas ocultas y arquitectura $(m_0 = D, m_1, \ldots, m_L, m_{L+1})$.



Teoremas Universales de Aproximación

Sea $\sigma \in C(\mathbb{R})$ una función de activación. Definimos la clase de funciones

$$\mathcal{M}(\sigma) = \text{span}\{x \mapsto \sigma(w \cdot x - \theta) \,|\, \theta \in \mathbb{R}, w \in \mathbb{R}^n\}$$

Teorema Universal de Aproximación

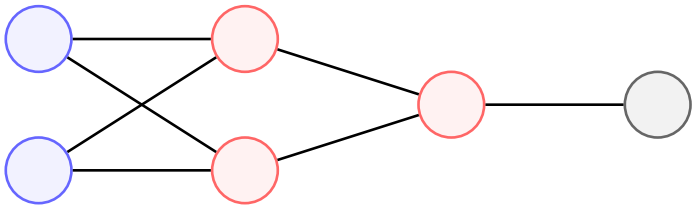
Sea $\sigma \in C(\mathbb{R})$. Entonces $\mathcal{M}(\sigma)$ es densa en $C(\mathbb{R}^n)$, en la topología de convergencia uniforme en compactos, si y solo si σ no es un polinomio.

Es decir, siempre podemos encontrar una red neuronal que aproxime tanto como queramos a una función continua $f : \mathbb{R}^n \to \mathbb{R}$ dada, usando funciones de activación no polinomiales.

¿Qué pasa si la función de activación es polinomial?

La flexibilidad de una red MLP está dada por las funciones de activiación no lineales. Tomemos dos capas de 1 neurona con activación lineal. Si $P_1(x_1, x_2) = w_0^{(1)} + w_1^{(1)} x_1 + w_2^{(1)} x_2$ y $P_2(z) = w_0^{(2)} + w_1^{(2)} z$ entonces

$$P_2 \circ P_1(x_1, x_2) = \begin{pmatrix} w_0^{(2)} + w_1^{(2)} w_0^{(1)} \\ w_1^{(2)} w_1^{(1)} \\ w_1^{(2)} w_2^{(1)} \end{pmatrix}^T \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$$

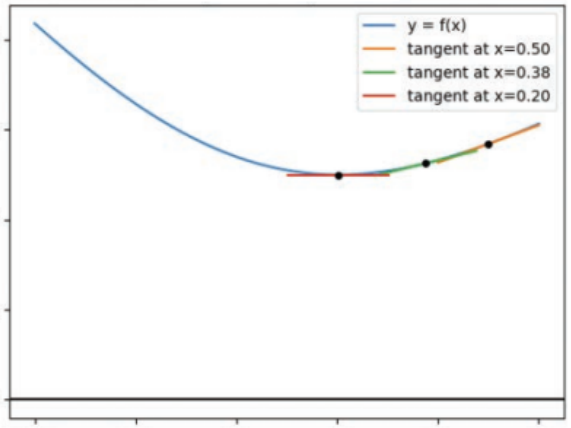


¿Cómo encontramos el mínimo de una función?

- Sea $f(x)$ una función, queremos encontrar el mínimo de f.

¿Cómo encontramos el mínimo de una función?

- Sea $f(x)$ una función, queremos encontrar el mínimo de f.
- La derivada en el punto x que minimiza el valor de f es 0.



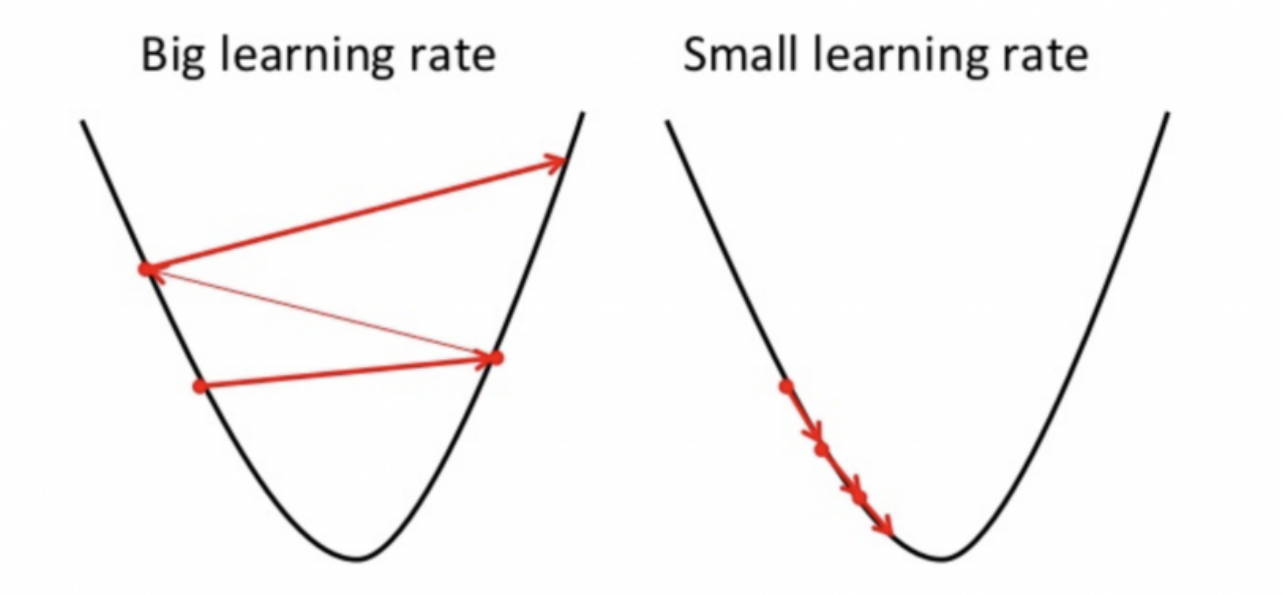
Descenso de gradiente

- A partir de un valor inicial x queremos encontrar un valor que minimiza f.
- El signo de $f'(x)$ nos indica la dirección en la cual *mover* x para reducir el valor de $f(x)$.
- La magnitud de $f'(x)$ nos indica que tanto ajustar x. Usamos un peso η, llamado learning rate,

$$x_{n+1} = x_n - \eta f'(x_n).$$

- Si η es muy grande, el método puede *pasarse* de la solución y no converger.
- El algoritmo no garantiza encontrar el mínimo global, puede quedarse atorado en un mínimo local.

Learning Rate

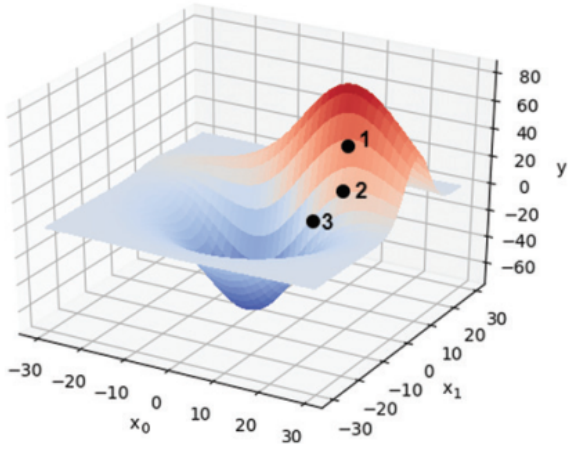


El caso de dimensión alta

- Consideremos la función $f(x_0, x_1)$.

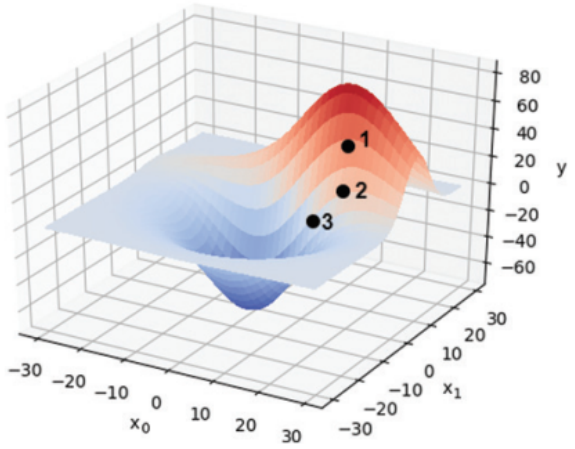
El caso de dimensión alta

- Consideremos la función $f(x_0, x_1)$.
- El gradiente es un vector que consta de las derivadas e indica la dirección en el espacio de features en la que se produce el ascenso más pronunciado en el valor de f.



El caso de dimensión alta

- Consideremos la función $f(x_0, x_1)$.
- El gradiente es un vector que consta de las derivadas e indica la dirección en el espacio de features en la que se produce el ascenso más pronunciado en el valor de f.



- Si estamos en el punto $\mathbf{x_n} = \left(x_0^{(n)}, x_1^{(n)}\right)$ y queremos minimizar f, ajustamos el punto de la siguiente forma

$$\mathbf{x_{n+1}} = \mathbf{x_n} - \eta \nabla f(\mathbf{x_n})$$

Gradient-based Learning

Al aplicar descenso de gradiente a nuestra red neuronal, consideramos las features $\mathbf{x}$ como constantes, el objetivo es encontrar los pesos $\mathbf{w}$ de tal forma que minimicemos el error.

- ¿Qué función queremos minimizar? Una función de perdida.

Gradient-based Learning

Al aplicar descenso de gradiente a nuestra red neuronal, consideramos las features **x** como constantes, el objetivo es encontrar los pesos **w** de tal forma que minimicemos el error.

- ¿Qué función queremos minimizar? Una función de perdida.

- En el contexto de optimización, la función que evalua una solución candidata se le llama función objectivo. En las redes neuronales, la función objetivo se le llama función costo o perdida (loss function).

Gradient-based Learning

- In the case of the Perceptron, the loss function is given by

$$L^{(0/1)}(\mathbf{w}) = (y_i - \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle)^2$$

Gradient-based Learning

- In the case of the Perceptron, the loss function is given by

$$L^{(0/1)}(\mathbf{w}) = \frac{1}{2}(y_i - \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle)^2 = 1 - y_i \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle$$

- This function is not smooth, we use the smooth surrogate loss function

$$L(\mathbf{w}) = \max\{-y_i \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle, 0\}.$$

Gradient-based Learning

- In the case of the Perceptron, the loss function is given by
- This function is not smooth, we use the smooth surrogate loss function

$$L(\mathbf{w}) = \max\{-y_i \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle, 0\}.$$

- Applying gradient descent

$$\begin{aligned} \mathbf{w}_{n+1} &= \mathbf{w}_n - \eta \nabla L(\mathbf{w}) \\ &= \begin{cases} \mathbf{w}_n + \eta y_i \mathbf{x}_i, & \text{well classified} \\ \mathbf{w}_n, & \text{misclassified} \end{cases} \end{aligned}$$

Loss Functions

Other examples of loss functions

- Keras Losses

Loss Functions: MSE

Mean Square Error (MSE)

$$L(y, t) = (t - y)^2.$$

- L2 loss.

Loss Functions: MSE

Mean Square Error (MSE)

$$L(y, t) = (t - y)^2.$$

- L2 loss.
- Good for regression tasks.

Loss Functions: MSE

Mean Square Error (MSE)

$$L(y, t) = (t - y)^2.$$

- L2 loss.
- Good for regression tasks.
- Trivial derivative for gradient descent.

Loss Functions: MAE

Mean Absolute Error (MAE)

$$L = |t - y|.$$

- L1 loss.
- More robust to outliers than mse.
- Good for regression tasks.
- Discontinuity in its derivative.

Loss Functions: Hinge

Hinge Loss

$$L = \max\{-y_i \hat{y}_i, 0\}.$$

- Used in SVMs and Perceptron.
- Penalizes errors, but also correct predictions of low confidence (probabilities).
- Good for binary classification tasks.

Loss Functions: Categorical cross entropy

Categorical cross entropy

$$L = \sum_{i}^{n} y_i \log(\hat{y}_i).$$

- Good for multi-class classification problems.
- Considers y to be a one-hot encoding vector in n classes.

Tamaño de paso y `batch_size`

`batch_size`

El `batch_size` es el número de ejemplos que se pasan a la red antes de actualizar pesos.

Empiezar con un batch pequeño (32 o 64) y luego se puede aumentar. Aumentar si:

- El modelo tarda mucho en entrenar.
- Hay GPU disponible y se quiere acelerar el entrenamiento.

Tamaño de paso y `batch_size`

Analogía

Imagina que estás aprendiendo a lanzar dardos:

- Batch pequeño: Ajustas tu brazo después de cada lanzamiento (aprendes rápido pero con mucha variabilidad).
- Batch grande: Ajustas tu brazo después de 10 lanzamientos (aprendes más lento pero de forma más consistente).

Optimizers: Gradient Descent

Gradient descent is the most basic and intuitive optimization algorithm. Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function.

Advantages:

- Easy to implement.
- Easy to understand.

Optimizers: Gradient Descent

Gradient descent is the most basic and intuitive optimization algorithm. Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function.

Advantages:

- Easy to implement.
- Easy to understand.

Disadvantages:

- May get trap at local minima.
- Weights are changed after calculating gradient on the whole dataset (or batch). May take a long time to converge.
- Requires large memory to calculate gradient on the whole dataset (or batch).

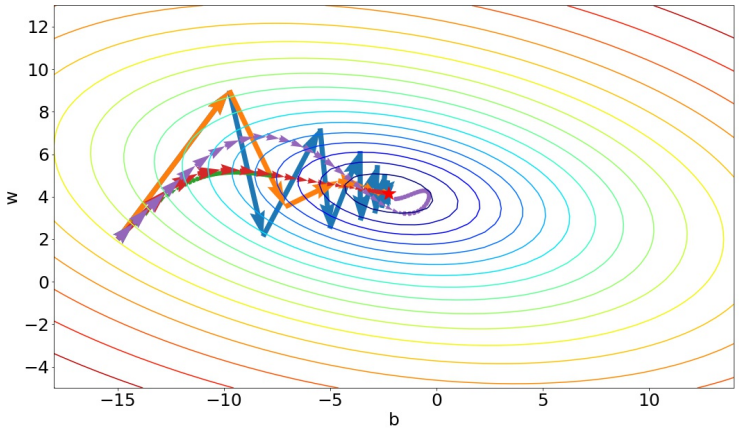
Optimizers

We can use other optimizers:

- Gradient Descent
- Stochastic Gradient Descent
- Stochastic Gradient Descent with momentum
- Mini-Batch Gradient Descent
- Adagrad
- RMSProp
- AdaDelta
- Adam

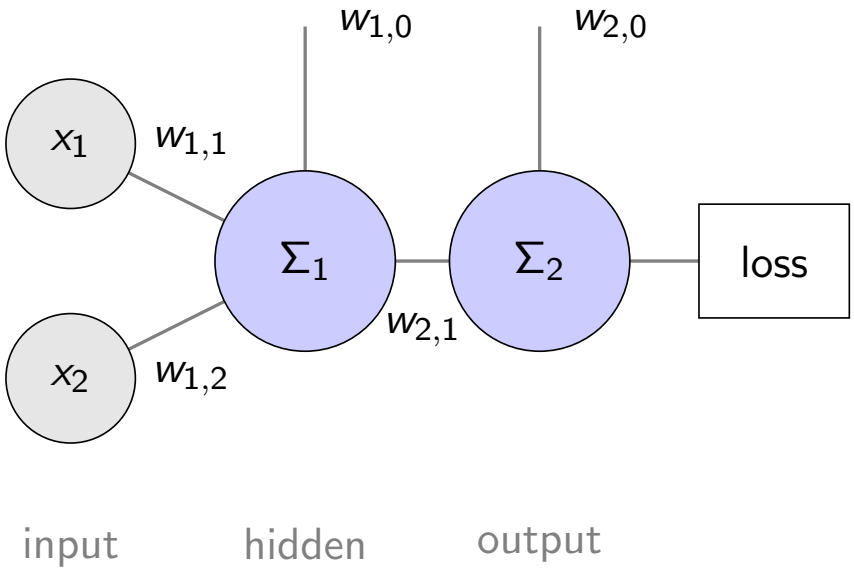
▸ Keras Optimizers

Different Optimizers



Visualizaciones, Several optimizers

Example: Back-propagation



This neural network implements

$$f = S\left(w_{2,0} + w_{2,1}\tanh(w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2)\right)$$

We use the MSE loss:

$$e(f) = \frac{1}{2}(y - f)^2.$$

Example: Back-propagation

- The error is given by

$$e(\mathbf{w}) = \frac{1}{2}\left(y - S\left(w_{2,0} + w_{2,1}\tanh(w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2)\right)\right)^2$$

where $\mathbf{w} = (w_{1,0}, w_{1,1}, w_{1,2}, w_{2,0}, w_{2,1})$

- We write it as:

$$\begin{aligned} e(f) &= \frac{1}{2}(y - f)^2 \\ f(z_f) &= S(z_f) \\ z_f(w_{2,0}, w_{2,1}, g) &= w_{2,0} + w_{2,1}g \\ g(z_g) &= \tanh(z_g) \\ z_g(w_{1,0}, w_{1,1}, w_{1,2}) &= w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2 \end{aligned}$$

Example: Back-propagation

Compute the partial derivatives

$$\frac{\partial}{\partial w_{2,0}} e = -(y - f) S'(z_f)$$

$$\frac{\partial}{\partial w_{2,1}} e = -(y - f) S'(z_f) g$$

$$\frac{\partial}{\partial w_{1,0}} e = -(y - f) S'(z_f) w_{2,1} \tanh'(z_g)$$

$$\frac{\partial}{\partial w_{1,1}} e = -(y - f) S'(z_f) w_{2,1} \tanh'(z_g) x_1$$

$$\frac{\partial}{\partial w_{1,2}} e = -(y - f) S'(z_f) w_{2,1} \tanh'(z_g) x_2$$

Example: Back-propagation

Finally, we update the weights, via gradient descent

$$w_{2,0} \leftarrow w_{2,0} - \eta \frac{\partial e}{\partial w_{2,0}}$$

$$w_{2,1} \leftarrow w_{2,1} - \eta \frac{\partial e}{\partial w_{2,1}}$$

$$w_{1,0} \leftarrow w_{1,0} - \eta \frac{\partial e}{\partial w_{1,0}}$$

$$w_{1,1} \leftarrow w_{1,1} - \eta \frac{\partial e}{\partial w_{1,1}}$$

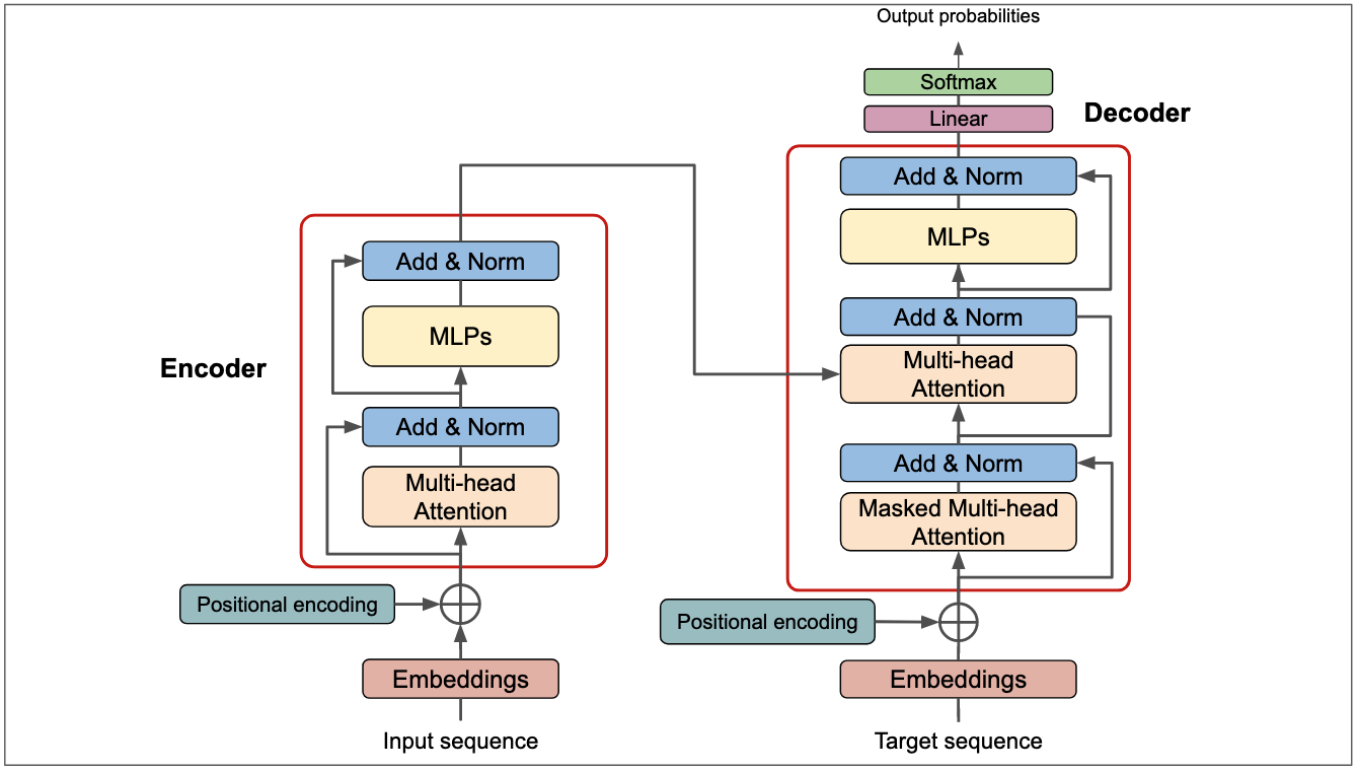
$$w_{1,2} \leftarrow w_{1,2} - \eta \frac{\partial e}{\partial w_{1,2}}$$

Algunas arquitecturas

Algunos tipos de arquitecturas de redes neuronales:

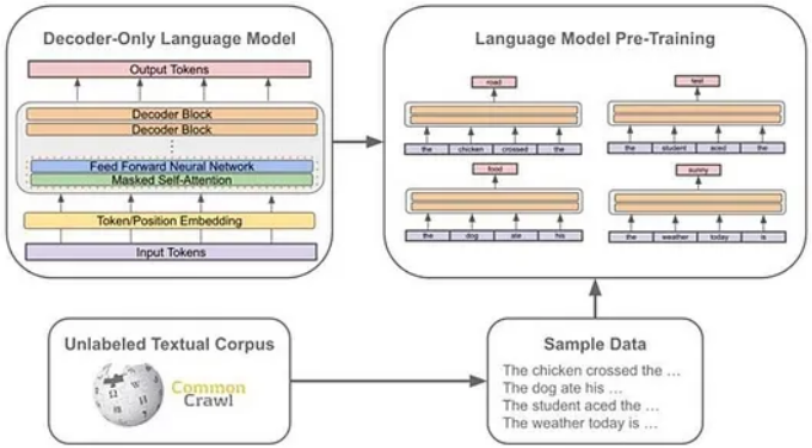
- Perceptrones Multicapa (MLP).
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).
- Self-Organizing Map (SOM).
- Transformadores: GPT, BERT, DeepSeek, Claude
- Redes de Difusión (Diffusion Models)
- Redes Siamesas (Siamese Networks)
- Redes Neuronales de Grafos (GNN - Graph Neural Networks)

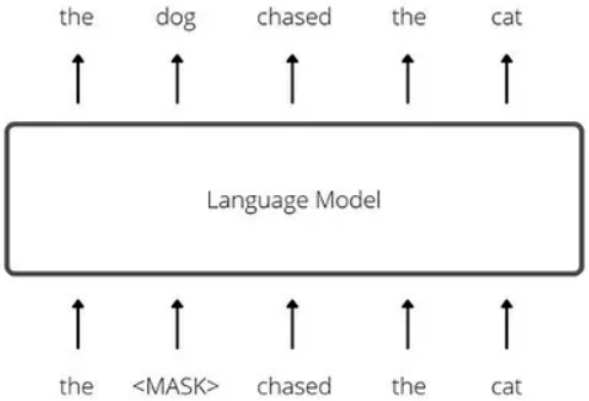
Transformadores: Arquitectura



Entrenamiento explicado

Transformadores: Entrenamiento de un LLM





Entrenamiento explicado

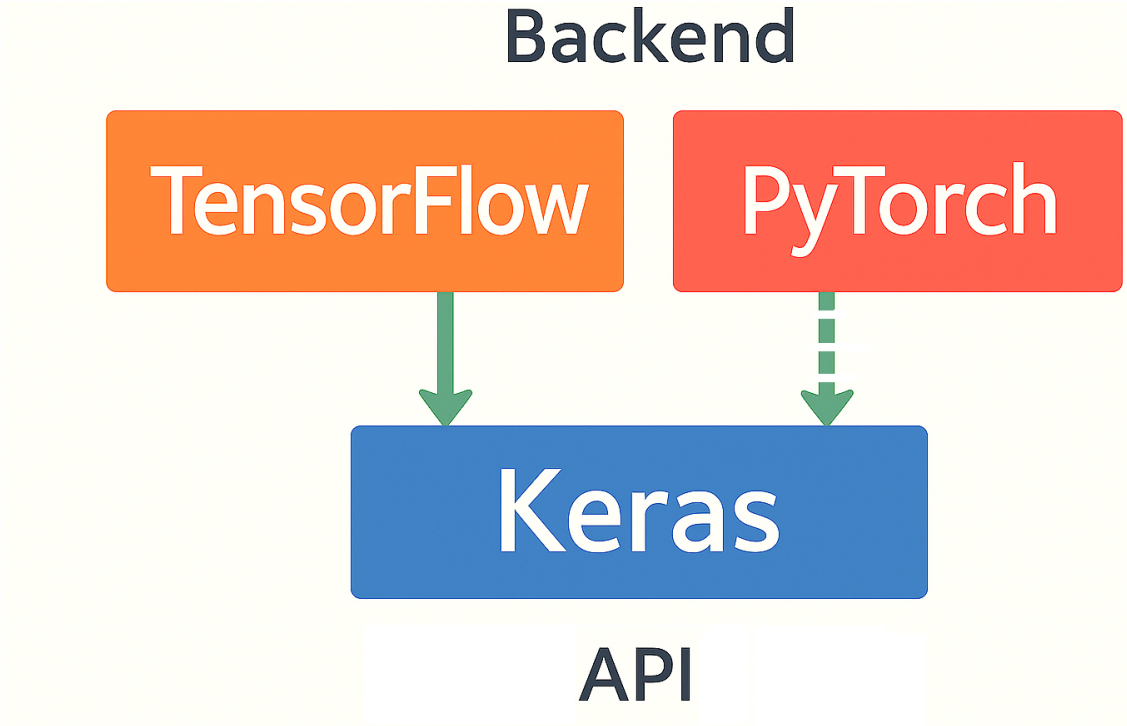
Table of Contents

Deep Learning Libraries

Tensorflow

Pytorch

Keras, Tensorflow and Pytorch



Comparación de los frameworks


```keras
from keras.models import Sequential
from keras.layers import Dense, Flatten

model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(8, activation='relu'),
    Dense(10, activation='softmax')
])
```

Keras

```tensorflow
import tensorflow as tf

# 1. Definir variables (pesos y biases)
W1 = tf.Variable(tf.random.normal([28*28, 8]))
b1 = tf.Variable(tf.zeros([8]))
W2 = tf.Variable(tf.random.normal([8, 10]))
b2 = tf.Variable(tf.zeros([10]))

# 2. Función forward
def model(x):
    x = tf.reshape(x, [-1, 28*28])  # Flatten
    x = tf.nn.relu(tf.matmul(x, W1) + b1)
    return tf.nn.softmax(tf.matmul(x, W2) + b2)
```

TensorFlow Puro

```pytorch
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.dense1 = nn.Linear(28*28, 8)
        self.dense2 = nn.Linear(8, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.dense1(x))
        return torch.softmax(self.dense2(x), dim=1)

model = MLP()
```

PyTorch