MS AI PATHWAY

ML SPECIALIST V3.3 | 24 SEMANAS

LÍNEA 1 : MACHINE LEARNING

ÍNDICE GENERAL

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

📚 GUÍA MAESTRA: MS AI PATHWAY - ML SPECIALIST (v3.1)

De Python Básico a Candidato del MS in AI de CU Boulder
24 Semanas (6 Meses Exactos) | Enfoque: Línea 1 - Machine Learning
Filosofía: "Matemáticas Aplicadas a Código"

🎯 Objetivo de Esta Guía

Dominio absoluto de las 3 materias de la Línea de Machine Learning del Performance-Based Admission Pathway:

⭐ LÍNEA 1: Machine Learning (3 créditos) - FOCO PRINCIPAL

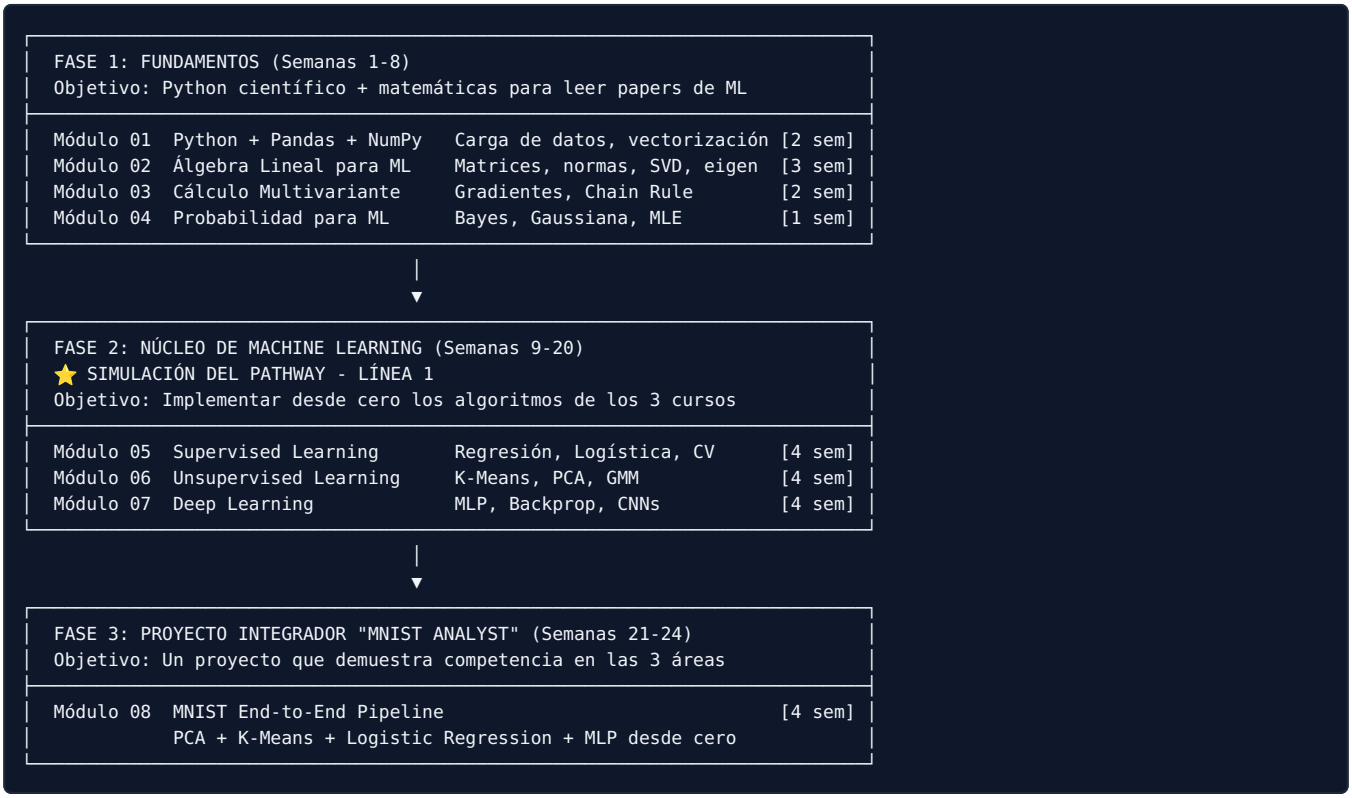
Curso del Pathway	Módulo de Esta Guía
Introduction to Machine Learning: Supervised Learning	**Módulo 05**
Unsupervised Algorithms in Machine Learning	**Módulo 06**
Introduction to Deep Learning	**Módulo 07**

📖 LÍNEA 2: Probabilidad y Estadística (Lectura Opcional)

Curso del Pathway	Estado
Probability Foundations for Data Science and AI	Lectura opcional
Discrete-Time Markov Chains and Monte Carlo Methods	Lectura opcional
Statistical Estimation for Data Science and AI	Lectura opcional

Nota: La Línea 2 pertenece a la especialización de Estadística. Esta guía incluye solo la probabilidad esencial para ML (Módulo 04).

🗺️ El Mapa de Ruta: 3 Fases Críticas



Total: 8 módulos obligatorios | 24 semanas | ~864 horas

👤 Perfil de Entrada

```
┌─────────────────────────────────────────────────┐
│  PERFIL IDEAL DE ENTRADA                        │
├─────────────────────────────────────────────────┤
│  ✅ Python básico (variables, funciones, listas, diccionarios) │
│  ✅ Lógica de programación (if/else, loops)                    │
│  ✅ Matemáticas de bachillerato (álgebra básica)               │
│  ✅ Ganas de entender "cómo funciona por dentro"               │
│  ⚠️  NO se requiere: numpy, pandas, sklearn, ML previo         │
└─────────────────────────────────────────────────┘
```

📖 Módulos Obligatorios

FASE 1: Fundamentos (Semanas 1-8)

Python científico con Pandas, matemáticas esenciales y probabilidad básica para ML.

#	Módulo	Descripción	Tiempo	Archivo
01	**Python + Pandas + NumPy**	Carga de datos, limpieza, vectorización	2 sem	01_PYTHON_CIENTIFICO.md
02	**Álgebra Lineal para ML**	Vectores, matrices, normas, SVD, eigenvalues	3 sem	02_ALGEBRA_LINEAL_ML.md
03	**Cálculo Multivariante**	Derivadas parciales, gradiente, Chain Rule	2 sem	03_CALCULO_MULTIVARIANTE.md
04	**Probabilidad para ML**	Teorema de Bayes, Gaussiana, MLE	1 sem	04_PROBABILIDAD_ML.md

Entregables Fase 1:
- Script de carga y limpieza de CSV con Pandas
- Librería `linear_algebra.py` con proyecciones y distancias
- Gradient Descent manual para minimizar funciones
- Implementación de MLE para estimar parámetros de Gaussiana

FASE 2: Núcleo de Machine Learning (Semanas 9-20) ⭐ PATHWAY LÍNEA 1

Los 3 cursos del Pathway implementados desde cero.

#	Módulo	Curso del Pathway	Tiempo	Archivo
05	**Supervised Learning**	Introduction to ML: Supervised Learning	4 sem	05_SUPERVISED_LEARNING.md
06	**Unsupervised Learning**	Unsupervised Algorithms in ML	4 sem	06_UNSUPERVISED_LEARNING.md
07	**Deep Learning**	Introduction to Deep Learning	4 sem	07_DEEP_LEARNING.md

Entregables Fase 2:
- `logistic_regression.py` con regularización L2
- `kmeans.py` y `pca.py` funcionales
- `neural_network.py` con backprop manual (MLP)
- Teoría de CNNs (convolución, pooling, stride)

FASE 3: Proyecto Final MNIST Analyst (Semanas 21-24)

Pipeline completo en 4 semanas. MNIST es simple, no necesita más.

#	Módulo	Descripción	Tiempo	Archivo
08	**MNIST Analyst**	Pipeline end-to-end de clasificación de dígitos	4 sem	08_PROYECTO_MNIST.md

Proyecto: "End-to-End Handwritten Digit Analysis Pipeline"

Semana	Componente	Materia Demostrada
2 1	EDA + PCA + K-Means	Unsupervised Algorithms
2 2	Regresión Logística One-vs-All	Supervised Learning
2 3	MLP con Backprop desde cero	Deep Learning
2 4	Informe + Comparación de Modelos	Integración

🔨 Estructura del Proyecto Final

```
mnist-analyst/
├── src/
│   ├── __init__.py
│   │
│   ├── # FASE 1: FUNDAMENTOS
│   ├── data_loader.py         # Carga con Pandas, limpieza (Módulo 01)
│   ├── linear_algebra.py      # Vectores, matrices, normas (Módulo 02)
│   ├── calculus.py            # Gradientes, derivadas (Módulo 03)
│   ├── probability.py         # Bayes, Gaussiana, MLE (Módulo 04)
│   │
│   ├── # FASE 2: ML CORE
│   ├── logistic_regression.py # Clasificación binaria/multiclase (Módulo 05)
│   ├── metrics.py             # Accuracy, Precision, Recall, F1 (Módulo 05)
│   ├── kmeans.py              # Clustering K-Means++ (Módulo 06)
│   ├── pca.py                 # Reducción dimensional SVD (Módulo 06)
│   ├── neural_network.py      # MLP con backprop (Módulo 07)
│   ├── activations.py         # Sigmoid, ReLU, Softmax (Módulo 07)
│   ├── optimizers.py          # SGD, Adam (Módulo 07)
│   │
│   └── # INTEGRACIÓN
│   └── mnist_pipeline.py      # Pipeline completo (Módulo 08)
│
├── tests/
│   ├── test_linear_algebra.py
│   ├── test_logistic_regression.py
│   ├── test_kmeans.py
│   ├── test_pca.py
│   ├── test_neural_network.py
│   └── test_pipeline.py
│
├── data/
│   └── mnist/                 # Dataset MNIST (28x28 imágenes)
│
├── notebooks/
│   ├── 01_eda_visualization.ipynb
│   ├── 02_pca_kmeans.ipynb
│   ├── 03_logistic_ova.ipynb
│   └── 04_mlp_benchmark.ipynb
│
├── docs/
│   ├── MATHEMATICAL_FOUNDATIONS.md
│   └── MODEL_COMPARISON.md
│
├── README.md                  # Documentación (inglés)
├── pyproject.toml
└── requirements.txt           # numpy, pandas, matplotlib, pytest
```

⏱️ Tiempo Total

Fase	Semanas	Horas (~ 3 6 h/sem)	Enfoque
Fundamentos (0 1 - 0 4)	8	~ 2 8 8 h	Python + Matemáticas + Probabilidad
ML Core (0 5 - 0 7)	1 2	~ 4 3 2 h	Algoritmos del Pathway
Proyecto MNIST (0 8)	4	~ 1 4 4 h	Integración y demo

Fase	Semanas	Horas (~36 h/sem)	Enfoque
TOTAL	24	~864 h	

Duración: 6 meses exactos con 6 h/día (L-S)

❌ Qué Se Eliminó del Plan Original

Para que esto quepa en 6 meses y sea efectivo para la **Línea 1 de ML**:

Eliminado	Razón
Linked Lists, Stacks, Queues	Irrelevante para matemáticas del Pathway
Binary Trees, BST	No se usa en los 3 cursos de ML
Grafos (BFS/DFS)	No es parte del currículo
QuickSort, MergeSort	En ML usas `numpy.sort()`
Inverted Index, TF-IDF	Proyecto de IR, no de CV/ML
Cadenas de Markov	Pertenece a Línea 2 (Estadística)
Motor de Búsqueda	Reemplazado por MNIST Pipeline

📦 Material de Referencia

Documento	Descripción	Uso
GLOSARIO.md	Definiciones técnicas de ML	Consulta
RECURSOS.md	Cursos y libros externos	Profundizar
CHECKLIST.md	Verificación de entregables	Seguimiento

🚀 Comenzar

→ Módulo 01: Python + Pandas + NumPy

📌 Restricciones del Proyecto

- ✅ **NumPy + Pandas permitidos** - Herramientas reales de ML
- ❌ **Sin sklearn/tensorflow/pytorch** - Algoritmos desde cero
- ✅ **100% local** - Todo se ejecuta en tu máquina
- ✅ **Matemáticas primero** - Entender antes de implementar
- ✅ **MNIST como benchmark** - Dataset estándar de la industria

🎯 Verificación de Competencias del Pathway

Curso del Pathway	¿Cubierto?	Evidencia en el Proyecto
ML: Supervised Learning	✅	Logistic Regression OvA, métricas, CV
ML: Unsupervised Algorithms	✅	K-Means++, PCA con SVD desde cero
ML: Deep Learning	✅	MLP con Backprop + teoría CNNs

✨ Cambios en v3.1 (vs v3.0)

Cambio	Razón
24 **semanas** (antes 26)	Proyecto MNIST reducido a 4 sem (es dataset simple)

Cambio	Razón
Pandas en Módulo 01	Necesario para cargar y limpiar datos reales
Probabilidad para ML (Módulo 04)	Bayes y MLE son esenciales para entender loss functions
CNNs en Módulo 07	El curso de Deep Learning de CU Boulder las cubre

✨ Cambios en v3.2 (vs v3.1)

Cambio	Razón
Debugging NumPy (M01)	5 errores comunes que causan horas de frustración
Estándares Profesionales	`mypy`, `ruff`, `pytest` obligatorios desde Semana 2
Metodología Feynman	"Reto del Tablero Blanco" en cada módulo
Derivación Analítica (M05, M07)	Simula exámenes de posgrado: derivar gradientes a mano
Análisis Bias-Variance (M08)	Concepto central de ML para diseño de modelos
Formato Paper (M08)	Notebook final con estructura académica

✨ Cambios en v3.3 (vs v3.2)

Cambio	Razón
Gradient Checking (M03)	Validación matemática de derivadas (técnica CS231n Stanford)
Log-Sum-Exp Trick (M04)	Softmax numéricamente estable (evita NaN)
Shadow Mode (M05)	Validar implementaciones vs sklearn
Overfit Test (M07)	Si no hace overfit en 10 ejemplos, tiene bug
Análisis de Errores (M08)	Visualizar y explicar fallos (nivel senior)
Curvas de Aprendizaje (M08)	Diagnóstico gráfico de Bias-Variance

Nuevos Entregables v3.3

Módulo	Nuevo Entregable
03	`grad_check.py` - validación numérica de derivadas
04	`softmax` con log-sum-exp trick
05	Comparativa Shadow Mode vs sklearn
07	`overfit_test.py` - debugging de redes
08	Sección "Error Analysis" + Learning Curves

> 💡 **Filosofía v3.3:** Esta guía incluye **validación matemática rigurosa** en cada paso. No confíes en que tu código "parece funcionar"—valídalo con gradient checking, shadow mode y overfit tests. Si completas v3.3, tu código es **matemáticamente correcto y profesionalmente validado**.

PLAN DE ESTUDIOS

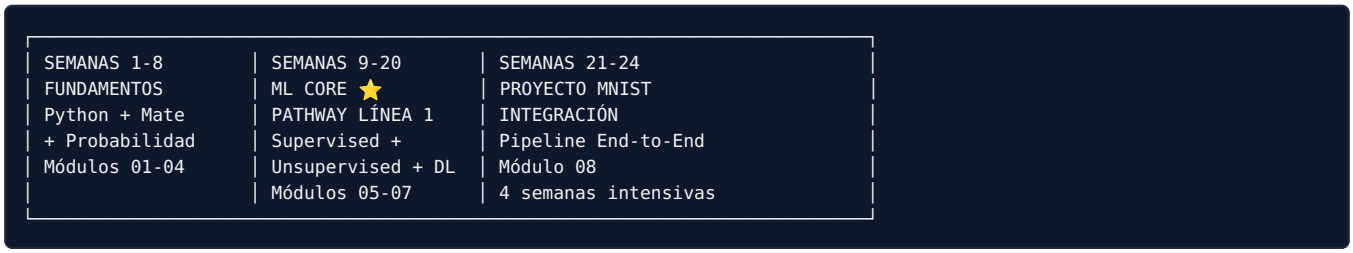
MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

📅 Plan de Estudios - ML SPECIALIST v3.1

24 Semanas | 6 horas/día | Lunes a Sábado
Preparación para MS in AI Pathway - Línea 1: Machine Learning

🗓️ Vista General: 24 Semanas



Dedicación total: 36 horas/semana × 24 semanas = ~864 **horas**

Los 8 Módulos Obligatorios

Semanas	Módulo	Tema	Curso del Pathway
1-2	01	Python + Pandas + NumPy	- (Fundamento)
3-5	02	Álgebra Lineal para ML	- (Fundamento)
6-7	03	Cálculo Multivariante	- (Fundamento)
8	04	Probabilidad para ML	- (Fundamento)
9-12	05	Supervised Learning	Introduction to ML: Supervised Learning
13-16	06	Unsupervised Learning	Unsupervised Algorithms in ML
17-20	07	Deep Learning + CNNs	Introduction to Deep Learning
21-24	08	Proyecto MNIST Analyst	Integración de las 3 materias

Filosofía v3.1: "Matemáticas Aplicadas a Código". Pandas para datos, NumPy para matemáticas, probabilidad para loss functions.

📌 Distribución Diaria Típica

Bloque	Horario	Actividad	Duración
🌅 Mañana	08:00 - 10:30	Teoría matemática + notación	2.5 h
☕ Pausa	10:30 - 11:00	Descanso	30 min
🌇 Mediodía	11:00 - 13:30	Implementación en NumPy	2.5 h
🌙 Tarde	15:00 - 16:00	Ejercicios + visualización	1 h

🔷 FASE 1: FUNDAMENTOS MATEMÁTICOS (Semanas 1-8)

Objetivo: Leer notación matemática y traducirla a Python/NumPy

🗓️ SEMANA 1-2: Python + Pandas + NumPy (Módulo 01)

Objetivo: Dominar Pandas para datos reales + NumPy para matemáticas
Por qué: En el mundo real, los datos vienen en CSVs sucios. Pandas es esencial para cargar y limpiar datos antes de aplicar ML.

Semana 1: Pandas + NumPy Básico

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Pandas: DataFrame y Series	`pd.read_csv()`, `.head()`, `.info()`	Ejercicio 1.1
M	Limpieza de datos	`dropna()`, `fillna()`, dtypes	Ejercicio 1.2
X	Selección y filtrado	`.loc[]`, `.iloc[]`, condiciones	Ejercicio 1.3
J	NumPy: Arrays vs listas	Crear arrays, dtypes	Ejercicio 1.4
V	NumPy: Indexing y slicing	Extraer submatrices	Ejercicio 1.5
S	**Checkpoint**	Pandas → NumPy: `.to_numpy()`	Pipeline de carga

Semana 2: NumPy Vectorizado

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Broadcasting (concepto)	Operaciones elemento a elemento	Ejercicio 1.6
M	np.dot, np.matmul, @	Producto matricial	Ejercicio 1.7
X	Reshape, flatten, transpose	Manipulación de formas	Ejercicio 1.8
J	Agregaciones y ejes	sum, mean, std con axis	Ejercicio 1.9
V	Random en NumPy	Generación de datos sintéticos	Ejercicio 1.10
S	**Checkpoint**	Pipeline completo CSV → NumPy	Entregable

Entregable: Script que carga CSV con Pandas, limpia datos, y convierte a NumPy para análisis.

Recursos:
- Pandas Getting Started
- NumPy Quickstart

🗓️ SEMANA 3-5: Álgebra Lineal para ML (Módulo 02)

Objetivo: Vectores, matrices, normas, autovectores
Conexión con Pathway: Vital para Unsupervised Learning (PCA requiere Eigenvalues) y Deep Learning (multiplicaciones de matrices)

Semana 3: Vectores y Operaciones Básicas

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Vectores: definición geométrica	Crear vectores en NumPy	Ejercicio 2.1
M	Suma, resta de vectores	Implementar operaciones	Visualizar con matplotlib
X	Producto escalar (scalar mult)	`c * v` en NumPy	Ejercicio 2.2
J	Producto punto (dot product)	Fórmula $\vec{a} \cdot \vec{b}$	np.dot()
V	Interpretación geométrica	Proyección, ángulo	Diagrama
S	**Repaso**	Funciones vectoriales	Test

Semana 4: Normas y Distancias

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Norma L2 (Euclidiana)	$\|x\|_2 = \sqrt{\sum x_i^2}$	np.linalg.norm()
M	Norma L1 (Manhattan)	$|x|_1 = \sum x_i	

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
X	Distancia Euclidiana	$d(a,b) = \|a - b\|_2$	Función distancia
J	Distancia coseno	$1 - \cos(\theta)$	Similitud coseno
V	Aplicación: KNN concepto	Vecino más cercano	Demo simple
S	**Repaso**	Librería de distancias	Test

Semana 5: Matrices y Descomposición

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Matrices: suma, multiplicación	np.matmul, @	Ejercicio 2.3
M	Transpuesta, inversa	A.T, np.linalg.inv()	Ejercicio 2.4
X	Autovalores/Autovectores (intro)	Qué son y por qué importan	np.linalg.eig()
J	SVD (concepto)	Descomposición de matrices	np.linalg.svd()
V	Aplicación: PCA preview	Reducción dimensional	Demo visual
S	**Checkpoint**	`linear_algebra.py` completo	Entregable

Entregable: Librería `linear_algebra.py` que implementa:
- Producto punto, normas L_1 / L_2
- Distancia euclidiana y coseno
- Proyección de vectores
- Wrapper para eigenvalues

🗓️ SEMANA 6-7: Cálculo Multivariante (Módulo 03) [CRÍTICO]

Objetivo: Derivadas, gradiente, Chain Rule
Conexión con Pathway: Es el lenguaje del Deep Learning. Sin la Regla de la Cadena, no entenderás Backpropagation.

Semana 6: Derivadas, Gradiente y GD

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Derivada: tasa de cambio	Calcular derivadas simples	Ejercicio 3.1
M	Derivadas parciales	$\frac{\partial f}{\partial x}$ para $f(x,y)$	Ejercicio 3.2
X	Gradiente: vector de parciales	∇f	Implementar
J	Gradient Descent (concepto)	Algoritmo básico	Pseudocódigo
V	Gradient Descent (código)	Minimizar $f(x,y) = x^2 + y^2$	Implementar
S	**Repaso**	Learning rate y convergencia	Visualización

Semana 7: Chain Rule

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Regla de la Cadena (1D)	$\frac{d}{dx}f(g(x))$	Ejercicio 3.3
M	Regla de la Cadena (multi)	Composición de funciones	Ejercicio 3.4
X	Aplicación: función de pérdida	Derivar MSE	Ejercicio 3.5
J	Preview Backpropagation	Cómo fluyen gradientes	Diagrama
V	Derivar Cross-Entropy	Preparación para logística	Ejercicio 3.6

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
S	**Checkpoint**	GD + Chain Rule documentado	Entregable

Entregable: Gradient Descent manual con visualización de trayectoria.

Recursos:
- 3 Blue 1 Brown: Essence of Calculus

🗓️ SEMANA 8: Probabilidad para ML (Módulo 04)

Objetivo: Bayes, Gaussiana, MLE - lo mínimo para entender loss functions
Conexión con Pathway: Cross-Entropy viene de MLE. GMM usa Gaussianas.

Semana 8: Probabilidad Esencial

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Probabilidad básica	P(A), P(A|B)	Ejercicios
M	Teorema de Bayes	Prior, Likelihood, Posterior	Implementar
X	Distribución Gaussiana	PDF, μ, σ	`gaussian_pdf()`
J	Gaussiana multivariada	Matriz de covarianza	Implementar
V	MLE (Maximum Likelihood)	Por qué da Cross-Entropy	Demostración
S	**Checkpoint Fase 1**	`probability.py` completo	Entregable

Entregable: Librería `probability.py` con Gaussiana, MLE y softmax.

Recursos:
- 3 Blue 1 Brown: Bayes Theorem
- StatQuest: Maximum Likelihood

🔷 FASE 2: NÚCLEO DE MACHINE LEARNING (Semanas 9-20)

Objetivo: Implementar desde cero los algoritmos exactos de los 3 cursos del Pathway

🗓️ SEMANA 9-12: Supervised Learning (Módulo 05)

Materia: Introduction to Machine Learning: Supervised Learning

Semana 9: Regresión Lineal

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Regresión: concepto	Línea de mejor ajuste	Visualizar datos
M	Mínimos cuadrados	Fórmula cerrada: $(X^TX)^{- 1 }X^Ty$	Implementar
X	MSE como función de costo	$J(\theta) = \frac{ 1 }{n}\sum(y - \hat{y})^ 2 $	Calcular MSE
J	GD para regresión	Derivar gradiente de MSE	Implementar
V	Regresión múltiple	Más de una feature	Extender código
S	**Repaso**	`linear_regression.py` v 1	Test

Semana 10: Regresión Logística

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Clasificación binaria	0 / 1 , sí/no	Dataset simple
M	Función sigmoid	$\sigma(z) = \frac{ 1 }{ 1 +e^{-z}}$	Implementar

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
X	Hipótesis logística	$h_\theta(x) = \sigma(\theta^T x)$	Implementar
J	Cross-Entropy Loss	$-[y\log(\hat{y}) + (1 -y)\log(1 -\hat{y})]$	Implementar
V	GD para logística	Derivar gradiente	Implementar
S	**Repaso**	`logistic_regression.py`	Test

Semana 11: Evaluación y Métricas

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Train/Test split	Por qué separar datos	Implementar split
M	Accuracy	Porcentaje correcto	Implementar
X	Precision y Recall	TP, FP, FN, TN	Implementar
J	F 1 -Score	Media armónica	Implementar
V	Matriz de confusión	Visualización	matplotlib
S	**Repaso**	`metrics.py` completo	Test

Semana 12: Validación Cruzada y Regularización

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Overfitting vs Underfitting	Bias-Variance tradeoff	Diagrama
M	K-Fold Cross Validation	Validación robusta	Implementar
X	Regularización L 2 (Ridge)	$\lambda|\theta|^ 2 $	Añadir a regresión
J	Regularización L 1 (Lasso)	$\lambda|\theta|_ 1 $	Comparar
V	Selección de hiperparámetros	Grid search simple	Implementar
S	**Checkpoint**	Supervisado completo	Entregable

Entregable: `logistic_regression.py` desde cero usando NumPy para clasificar datos simples, con métricas y cross-validation.

🗓️ SEMANA 13-16: Unsupervised Learning (Módulo 06)

Materia: Unsupervised Algorithms in Machine Learning

Semana 13: K-Means Clustering

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Clustering: concepto	Agrupar sin etiquetas	Visualizar clusters
M	Algoritmo Lloyd (K-Means)	Asignar, actualizar, repetir	Pseudocódigo
X	Implementar K-Means	Versión básica	Código
J	K-Means++ inicialización	Mejor selección de centroides	Implementar
V	Criterio de parada	Convergencia	Implementar
S	**Repaso**	`kmeans.py` funcional	Test

Semana 14: Evaluación de Clusters

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Inercia (within-cluster)	Suma de distancias al centroide	Implementar
M	Método del codo	Elegir K óptimo	Visualizar
X	Silhouette Score	Calidad de clusters	Implementar
J	Limitaciones K-Means	Clusters no esféricos	Ejemplos
V	Generar datos sintéticos	make_blobs equivalente	Función propia
S	**Repaso**	Evaluación completa	Documento

Semana 15: PCA (Principal Component Analysis)

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Reducción dimensional	Por qué reducir	Visualizar
M	PCA: concepto	Dirección de máxima varianza	Diagrama
X	PCA con eigenvalues	Autovectores de covarianza	np.linalg.eig()
J	PCA con SVD	Más estable numéricamente	np.linalg.svd()
V	Varianza explicada	Cuánta info se pierde	Implementar
S	**Repaso**	`pca.py` v 1	Test

Semana 16: PCA Aplicado y GMM

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Reconstrucción desde PCA	Proyectar y reconstruir	Implementar
M	Compresión de imágenes	PCA para reducir	Demo visual
X	GMM (concepto)	Mezcla de Gaussianas	Teoría
J	EM Algorithm (intro)	Expectation-Maximization	Pseudocódigo
V	Detección de anomalías	Outliers con GMM	Concepto
S	**Checkpoint**	No supervisado completo	Entregable

Entregable: `kmeans.py` y `pca.py`. Usar PCA para comprimir una imagen y visualizar cuánta varianza se pierde con diferentes números de componentes.

🗓️ SEMANA 17-20: Deep Learning + CNNs (Módulo 07)

Materia: Introduction to Deep Learning

Semana 17: Perceptrón y Fundamentos

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Neurona artificial	Analogía biológica	Diagrama
M	Perceptrón simple	$y = \text{sign}(w \cdot x + b)$	Implementar
X	Funciones de activación	Sigmoid, ReLU, Tanh	Implementar todas
J	Limitación del perceptrón	No puede resolver XOR	Demostrar

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
V	Necesidad de capas	Redes multicapa	Diagrama
S	**Repaso**	`activations.py`	Test

Semana 18: Forward Propagation

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	MLP: arquitectura	Capas ocultas	Diagrama
M	Forward pass	Propagación hacia adelante	Pseudocódigo
X	Implementar forward	Clase NeuralNetwork	Código
J	Función de pérdida DL	Cross-entropy para multiclase	Softmax
V	Inicialización de pesos	Xavier, He	Implementar
S	**Repaso**	Forward funcional	Test

Semana 19: CNNs - Teoría (NO implementación)

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	Por qué CNNs para imágenes	Problema de MLP con imágenes	Diagrama
M	Operación de convolución	Kernel, filtro, feature map	Demo visual
X	Stride, padding, pooling	Cálculo de dimensiones output	Ejercicios
J	Arquitectura LeNet-5	Capas CONV + POOL + FC	Diagrama
V	Max pooling implementación	Concepto simple	Código básico
S	**Repaso**	Quiz de dimensiones CNN	Test teórico

Semana 20: Optimizadores y Entrenamiento

Día	Mañana (Teoría)	Mediodía (Código)	Tarde (Práctica)
L	SGD (Stochastic GD)	Mini-batches	Implementar
M	Momentum	Acelerar convergencia	Implementar
X	Adam optimizer	Adaptive learning rate	Implementar
J	Regularización DL	Dropout (concepto), L2	Implementar L2
V	Training loop completo	Epochs, batches, logging	Implementar
S	**Checkpoint**	MLP resuelve XOR	Entregable

Entregable: `neural_net.py` - Una red neuronal que resuelve el problema XOR y clasifica dígitos simples, implementando `backward()` manualmente.

🔷 FASE 3: PROYECTO FINAL "MNIST ANALYST" (Semanas 21-24)

Objetivo: Un proyecto intensivo de 4 semanas que demuestra competencia en las 3 áreas

Dataset: MNIST (imágenes de 28x28 píxeles de dígitos escritos a mano)

> 💡 **v3.1:** MNIST es un dataset simple (solo 10 clases, imágenes pequeñas). 4 semanas son suficientes.

🗓️ SEMANA 21: EDA + No Supervisado

Materia demostrada: Unsupervised Algorithms in Machine Learning

Día	Actividad
L	Cargar MNIST, entender estructura (7 8 4 dimensiones)
M	Implementar PCA desde cero, reducir a 2 - 3 componentes
X	Visualizar dígitos en gráfico 2 D
J	Implementar K-Means, agrupar dígitos SIN etiquetas
V	Visualizar centroides como imágenes 2 8 x 2 8
S	**Checkpoint:** Notebook PCA + K-Means

Entregable: Jupyter notebook con PCA 2 D y K-Means clustering.

🗓️ SEMANA 22: Clasificación Supervisada

Materia demostrada: Introduction to ML: Supervised Learning

Día	Actividad
L	Train/test split, normalización
M	Implementar Logistic Regression One-vs-All (1 0 clasificadores)
X	Entrenar y medir Accuracy global
J	Precision, Recall, F 1 , matriz de confusión
V	Visualizar errores (imágenes mal clasificadas)
S	**Checkpoint:** Logistic Regression completo

Entregable: `logistic_mnist.py` con métricas completas.

🗓️ SEMANA 23: Deep Learning

Materia demostrada: Introduction to Deep Learning

Día	Actividad
L	Diseñar arquitectura MLP (7 8 4 → 1 2 8 → 6 4 → 1 0)
M	Implementar forward pass + softmax
X	Implementar backprop con cross-entropy
J	Training loop con mini-batches
V	Entrenar y ajustar hiperparámetros
S	**Checkpoint:** MLP funcional > 9 0 % accuracy

Entregable: `neural_network_mnist.py` con backprop manual.

🗓️ SEMANA 24: Benchmark + Informe Final

Objetivo: Comparar modelos y documentar

Día	Actividad
L	Comparar rendimiento: Logistic vs MLP

Día	Actividad
M	Análisis: ¿por qué MLP es mejor? (no linealidad)
X	Escribir MODEL_COMPARISON.md
J	Crear README.md profesional (inglés)
V	Demo final: Jupyter notebook completo
S	**Entrega final + Autoevaluación**

Entregable Final:

```
mnist-analyst/
├── src/
│   ├── data_loader.py
│   ├── linear_algebra.py
│   ├── probability.py
│   ├── pca.py
│   ├── kmeans.py
│   ├── logistic_regression.py
│   ├── neural_network.py
│   └── mnist_pipeline.py
├── notebooks/
│   ├── 01_eda_pca_kmeans.ipynb
│   ├── 02_logistic_classification.ipynb
│   └── 03_neural_network_benchmark.ipynb
├── docs/
│   └── MODEL_COMPARISON.md
├── tests/
│   └── test_*.py
└── README.md
```

✅ Checklist de Finalización - ML SPECIALIST v3.1

Fase 1: Fundamentos (Módulos 01-04)

- [ ] Python + Pandas + NumPy dominado
- [ ] Álgebra lineal: normas, distancias, SVD, eigenvalues
- [ ] Cálculo: gradientes, chain rule, gradient descent
- [ ] Probabilidad: Bayes, Gaussiana, MLE, softmax

Fase 2: ML Core (Módulos 05-07) ⭐ PATHWAY

- [ ] **Supervised (0 5):** Logistic Regression con métricas
- [ ] **Unsupervised (0 6):** K-Means y PCA desde cero
- [ ] **Deep Learning (0 7):** MLP con backprop + teoría CNNs

Fase 3: Proyecto MNIST (Módulo 08)

- [ ] PCA reduce MNIST a 2 D con visualización
- [ ] K-Means agrupa dígitos sin etiquetas
- [ ] Logistic Regression clasifica con > 8 5 % accuracy
- [ ] MLP supera a Logistic con > 9 0 % accuracy
- [ ] MODEL_COMPARISON.md explica matemáticamente las diferencias
- [ ] README.md profesional en inglés

Verificación Final

- [ ] Puedo explicar matemáticamente por qué funciona cada algoritmo
- [ ] Puedo derivar las fórmulas de gradiente a mano
- [ ] Puedo implementar desde cero sin copiar código
- [ ] Entiendo convolución, stride, padding, pooling para CNNs
- [ ] Listo para los 3 cursos del Pathway Línea 1

📚 Recursos Recomendados

Matemáticas

- 3 Blue 1 Brown: Linear Algebra

- 3 Blue 1 Brown: Calculus

Machine Learning

- Stanford CS 2 2 9
- Coursera: Machine Learning (Andrew Ng)

Deep Learning

- 3 Blue 1 Brown: Neural Networks
- CS 2 3 1 n: CNNs for Visual Recognition

> 💡 **Filosofía v 3 . 1 :** Esta guía te lleva de Python básico a candidato competitivo del MS in AI en exactamente 6 meses (2 4 semanas). Si puedes implementar PCA, K-Means, Logistic Regression y un MLP desde cero sobre MNIST, y entiendes la teoría de CNNs, **dominas la Línea 1 del Pathway**.

MÓDULO 01 - PYTHON + PANDAS + NUMPY

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 01 - Python Científico + Pandas

🎯 **Objetivo:** Dominar Pandas para datos + NumPy para matemáticas
Fase: 1 - Fundamentos | **Semanas** 1 - 2
Prerrequisitos: Python básico (variables, funciones, listas, loops)

🧠 ¿Por Qué Este Módulo?

El Problema con Python Puro para ML

```
# ❌ Así NO se hace en Machine Learning
def dot_product_slow(a: list, b: list) -> float:
    """Producto punto con loop - LENTO."""
    result = 0
    for i in range(len(a)):
        result += a[i] * b[i]
    return result

# Para vectores de 1 millón de elementos:
# Tiempo: ~200ms
```

```
# ✅ Así SÍ se hace en Machine Learning
import numpy as np

def dot_product_fast(a: np.ndarray, b: np.ndarray) -> float:
    """Producto punto vectorizado - RÁPIDO."""
    return np.dot(a, b)

# Para vectores de 1 millón de elementos:
# Tiempo: ~2ms (100x más rápido)
```

Conexión con el Pathway

En los cursos de CU Boulder:
- **Supervised Learning:** Multiplicaciones de matrices para regresión
- **Unsupervised Learning:** PCA requiere descomposición de matrices
- **Deep Learning:** Forward/backward pass son operaciones matriciales

Sin NumPy, no puedes hacer ML eficiente.

📚 Contenido del Módulo

Semana 1: Pandas + NumPy Básico

```
┌─────────────────────────────────────────────────┐
│  DÍA 1: Pandas - DataFrame y Series             │
│  DÍA 2: Pandas - Carga de CSVs (read_csv, head, info)  │
│  DÍA 3: Pandas - Limpieza (dropna, fillna, dtypes)  │
│  DÍA 4: NumPy - Arrays y dtypes                 │
│  DÍA 5: NumPy - Indexing y Slicing              │
│  DÍA 6: Pandas → NumPy (df.values, df.to_numpy())  │
└─────────────────────────────────────────────────┘
```

Semana 2: NumPy Vectorizado

```
┌─────────────────────────────────────────────────┐
│  DÍA 1: Broadcasting                            │
│  DÍA 2: Producto matricial (@, np.dot, np.matmul)  │
│  DÍA 3: Reshape, flatten, transpose             │
│  DÍA 4: Agregaciones y operaciones con ejes     │
│  DÍA 5: Random y generación de datos sintéticos  │
│  DÍA 6: Entregable: Pipeline Pandas → NumPy     │
└─────────────────────────────────────────────────┘
```

💻 Conceptos Clave

0. Pandas Esencial (Días 1-3)

¿Por Qué Pandas?

En el mundo real de ML, los datos vienen en CSVs sucios, no en arrays NumPy perfectos. Antes de aplicar cualquier algoritmo necesitas:

1. **Cargar datos** desde archivos
2. **Explorar** estructura y tipos
3. **Limpiar** valores faltantes y errores
4. **Convertir** a NumPy para el modelo

```
import pandas as pd
import numpy as np

# ========== CARGA DE DATOS ==========
# Cargar CSV
df = pd.read_csv('data/iris.csv')

# Primeras filas
print(df.head())

# Información del DataFrame
print(df.info())
#  Column         Non-Null Count  Dtype
# ---  ------         --------------  -----
#  0   sepal_length   150 non-null    float64
#  1   sepal_width    150 non-null    float64
#  2   petal_length   150 non-null    float64
#  3   petal_width    150 non-null    float64
#  4   species        150 non-null    object

# Estadísticas básicas
print(df.describe())
```

Limpieza de Datos

```
import pandas as pd

# Crear DataFrame con datos sucios
df = pd.DataFrame({
    'edad': [25, 30, None, 45, 50],
    'salario': [50000, 60000, 70000, None, 90000],
    'ciudad': ['Madrid', 'Barcelona', 'Madrid', 'Sevilla', None]
})

# ========== DETECTAR NULOS ==========
print(df.isnull().sum())
# edad       1
# salario    1
# ciudad     1

# ========== ELIMINAR FILAS CON NULOS ==========
df_clean = df.dropna()  # Elimina filas con cualquier nulo
print(f"Filas después de dropna: {len(df_clean)}")  # 2

# ========== RELLENAR NULOS ==========
df_filled = df.copy()
df_filled['edad'] = df_filled['edad'].fillna(df_filled['edad'].mean())
df_filled['salario'] = df_filled['salario'].fillna(df_filled['salario'].median())
df_filled['ciudad'] = df_filled['ciudad'].fillna('Desconocido')

print(df_filled)
```

Selección y Filtrado

```
import pandas as pd

df = pd.read_csv('data/iris.csv')

# ========== SELECCIONAR COLUMNAS ==========
```

```
# Una columna (Serie)
sepal_length = df['sepal_length']

# Múltiples columnas (DataFrame)
features = df[['sepal_length', 'sepal_width']]

# ========== FILTRAR FILAS ==========
# Condición simple
setosa = df[df['species'] == 'setosa']

# Múltiples condiciones
large_setosa = df[(df['species'] == 'setosa') & (df['sepal_length'] > 5)]

# ========== LOC e ILOC ==========
# loc: por etiquetas
df.loc[0:5, ['sepal_length', 'species']]

# iloc: por posición (como NumPy)
df.iloc[0:5, 0:2]
```

De Pandas a NumPy (Día 6)

```
import pandas as pd
import numpy as np

df = pd.read_csv('data/iris.csv')

# ========== SEPARAR FEATURES Y TARGET ==========
# Features (X) - todas las columnas numéricas
X = df[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].to_numpy()
print(f"X shape: {X.shape}")  # (150, 4)
print(f"X dtype: {X.dtype}")  # float64

# Target (y) - convertir categorías a números
y = df['species'].map({'setosa': 0, 'versicolor': 1, 'virginica': 2}).to_numpy()
print(f"y shape: {y.shape}")  # (150,)

# ========== VERIFICAR ==========
print(f"Tipo X: {type(X)}")  # <class 'numpy.ndarray'>
print(f"Tipo y: {type(y)}")  # <class 'numpy.ndarray'>

# Ahora X e y están listos para algoritmos de ML
```

1. Arrays vs Listas

```
import numpy as np

# Lista de Python
lista = [1, 2, 3, 4, 5]

# Array de NumPy
array = np.array([1, 2, 3, 4, 5])

# Diferencias clave:
# 1. Tipo homogéneo (todos los elementos del mismo tipo)
# 2. Tamaño fijo después de creación
# 3. Operaciones vectorizadas
# 4. Almacenamiento contiguo en memoria
```

2. Creación de Arrays

```
import numpy as np

# Desde lista
a = np.array([1, 2, 3])

# Arrays especiales
zeros = np.zeros((3, 4))        # Matriz 3x4 de ceros
ones = np.ones((2, 3))          # Matriz 2x3 de unos
identity = np.eye(4)            # Matriz identidad 4x4
random = np.random.randn(3, 3)  # Matriz 3x3 valores normales

# Secuencias
```

```
rango = np.arange(0, 10, 2)     # [0, 2, 4, 6, 8]
linspace = np.linspace(0, 1, 5) # [0, 0.25, 0.5, 0.75, 1]

print(f"Shape de zeros: {zeros.shape}")  # (3, 4)
print(f"Dtype de zeros: {zeros.dtype}")  # float64
```

3. Indexing y Slicing

```
import numpy as np

# Crear matriz 2D
matrix = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# Acceso a elementos
print(matrix[0, 0])      # 1 (fila 0, columna 0)
print(matrix[1, 2])      # 6 (fila 1, columna 2)

# Slicing
print(matrix[0, :])      # [1, 2, 3] (toda la fila 0)
print(matrix[:, 1])      # [2, 5, 8] (toda la columna 1)
print(matrix[0:2, 1:3])  # [[2, 3], [5, 6]] (submatriz)

# Indexing booleano
print(matrix[matrix > 5])  # [6, 7, 8, 9]
```

4. Broadcasting

```
import numpy as np

# Broadcasting: operar arrays de diferentes shapes

# Escalar + Array
a = np.array([1, 2, 3])
print(a + 10)  # [11, 12, 13]

# Vector + Matriz (broadcasting automático)
matrix = np.array([
    [1, 2, 3],
    [4, 5, 6]
])
vector = np.array([10, 20, 30])

# El vector se "expande" para coincidir con la matriz
print(matrix + vector)
# [[11, 22, 33],
#  [14, 25, 36]]

# Regla de broadcasting:
# Las dimensiones deben ser iguales O una de ellas debe ser 1
```

5. Agregaciones y Ejes

```
import numpy as np

matrix = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

# Agregaciones globales
print(np.sum(matrix))   # 21 (suma de todos)
print(np.mean(matrix))  # 3.5 (promedio de todos)
print(np.std(matrix))   # 1.707... (desviación estándar)

# Agregaciones por eje
# axis=0: colapsar filas (operar columnas)
print(np.sum(matrix, axis=0))  # [5, 7, 9]

# axis=1: colapsar columnas (operar filas)
```

```
print(np.sum(matrix, axis=1))  # [6, 15]

# Visualización de ejes:
# ┌─────────────┐
# │ axis=0 ↓    │
# │ [1, 2, 3]   │ → axis=1
# │ [4, 5, 6]   │ → axis=1
# └─────────────┘
```

6. Operaciones Matriciales

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Operaciones elemento a elemento
print(A + B)   # Suma
print(A * B)   # Multiplicación elemento a elemento (Hadamard)
print(A / B)   # División elemento a elemento

# Producto matricial (lo que usarás en ML)
print(A @ B)           # Operador @ (Python 3.5+)
print(np.matmul(A, B)) # Función matmul
print(np.dot(A, B))    # Función dot

# Resultado:
# [[19, 22],
#  [43, 50]]

# Transpuesta
print(A.T)
# [[1, 3],
#  [2, 4]]
```

7. Vectorización: Eliminar Loops

```
import numpy as np

# ❌ CON LOOP (lento)
def normalize_loop(data: list) -> list:
    """Normalizar datos con loop."""
    mean = sum(data) / len(data)
    std = (sum((x - mean)**2 for x in data) / len(data)) ** 0.5
    return [(x - mean) / std for x in data]

# ✅ VECTORIZADO (rápido)
def normalize_vectorized(data: np.ndarray) -> np.ndarray:
    """Normalizar datos vectorizado."""
    return (data - np.mean(data)) / np.std(data)

# Ejemplo
data = np.random.randn(1000000)

# La versión vectorizada es ~100x más rápida
normalized = normalize_vectorized(data)
```

8. Funciones Universales (ufuncs)

```
import numpy as np

x = np.array([1, 2, 3, 4, 5])

# Funciones matemáticas (aplicadas elemento a elemento)
print(np.exp(x))      # e^x
print(np.log(x))      # ln(x)
print(np.sqrt(x))     # √x
print(np.sin(x))      # sin(x)

# Importante para ML:
# Sigmoid: σ(x) = 1 / (1 + e^(-x))
def sigmoid(x: np.ndarray) -> np.ndarray:
    return 1 / (1 + np.exp(-x))
```

```
# ReLU: max(0, x)
def relu(x: np.ndarray) -> np.ndarray:
    return np.maximum(0, x)

print(sigmoid(np.array([-2, -1, 0, 1, 2])))
# [0.119, 0.269, 0.5, 0.731, 0.881]
```

9. Reshape y Manipulación de Forma

```
import numpy as np

# Crear array 1D
a = np.arange(12)  # [0, 1, 2, ..., 11]

# Reshape a 2D
matrix = a.reshape(3, 4)
print(matrix.shape)  # (3, 4)
# [[ 0,  1,  2,  3],
#  [ 4,  5,  6,  7],
#  [ 8,  9, 10, 11]]

# Reshape a 3D
tensor = a.reshape(2, 2, 3)
print(tensor.shape)  # (2, 2, 3)

# Flatten: volver a 1D
flat = matrix.flatten()
print(flat.shape)  # (12,)

# -1 para inferir dimensión automáticamente
auto = a.reshape(4, -1)  # (4, 3)
auto = a.reshape(-1, 6)  # (2, 6)
```

10. Generación de Datos Aleatorios

```
import numpy as np

# Fijar semilla para reproducibilidad
np.random.seed(42)

# Distribución uniforme [0, 1)
uniform = np.random.rand(3, 3)

# Distribución normal (media=0, std=1)
normal = np.random.randn(3, 3)

# Distribución normal personalizada
custom_normal = np.random.normal(loc=5, scale=2, size=(100,))

# Enteros aleatorios
integers = np.random.randint(0, 10, size=(3, 3))

# Shuffle (mezclar)
data = np.arange(10)
np.random.shuffle(data)

# Muestreo sin reemplazo
sample = np.random.choice(data, size=5, replace=False)
```

📊 Type Hints con NumPy

```
import numpy as np
from numpy.typing import NDArray

# Type hints para arrays
def normalize(data: NDArray[np.float64]) -> NDArray[np.float64]:
    """Normaliza un array de floats."""
    return (data - np.mean(data)) / np.std(data)

# Type hints genéricos
def dot_product(a: np.ndarray, b: np.ndarray) -> float:
```

```
    """Calcula el producto punto de dos vectores."""
    return float(np.dot(a, b))

# Con mypy
# pip install numpy-stubs
```

⚡ Benchmark: Lista vs NumPy

```
import numpy as np
import time
from typing import List

def benchmark_dot_product():
    """Compara rendimiento de lista vs NumPy."""
    size = 1_000_000

    # Crear datos
    list_a: List[float] = [float(i) for i in range(size)]
    list_b: List[float] = [float(i) for i in range(size)]
    array_a = np.array(list_a)
    array_b = np.array(list_b)

    # Benchmark lista
    start = time.time()
    result_list = sum(a * b for a, b in zip(list_a, list_b))
    time_list = time.time() - start

    # Benchmark NumPy
    start = time.time()
    result_numpy = np.dot(array_a, array_b)
    time_numpy = time.time() - start

    print(f"Lista:  {time_list:.4f}s")
    print(f"NumPy:  {time_numpy:.4f}s")
    print(f"Speedup: {time_list/time_numpy:.1f}x")

    # Verificar resultados iguales
    assert abs(result_list - result_numpy) < 1e-6

if __name__ == "__main__":
    benchmark_dot_product()

# Output típico:
# Lista:  0.1523s
# NumPy:  0.0015s
# Speedup: 101.5x
```

🎯 Ejercicios

Ejercicio 1.1: Crear Arrays

```
# Crear:
# 1. Vector de 10 ceros
# 2. Matriz 3x3 de unos
# 3. Matriz identidad 4x4
# 4. Vector de 0 a 99
# 5. 20 valores equiespaciados entre 0 y 2π
```

Ejercicio 1.2: Indexing

```
# Dada la matriz:
matrix = np.arange(20).reshape(4, 5)

# Extraer:
# 1. Elemento en fila 2, columna 3
# 2. Toda la fila 1
# 3. Toda la columna 4
# 4. Submatriz filas 1-2, columnas 2-4
# 5. Elementos mayores que 10
```

Ejercicio 1.3: Broadcasting

```
# Sin usar loops:
# 1. Sumar 100 a cada elemento de una matriz 3x3
# 2. Multiplicar cada fila por un vector diferente
# 3. Normalizar cada columna (restar media, dividir por std)
```

Ejercicio 1.4: Vectorización

```
# Reescribir sin loops:
def euclidean_distance_loop(a: list, b: list) -> float:
    total = 0
    for i in range(len(a)):
        total += (a[i] - b[i]) ** 2
    return total ** 0.5

# Tu versión vectorizada:
def euclidean_distance_vectorized(a: np.ndarray, b: np.ndarray) -> float:
    pass  # Implementar
```

Ejercicio 1.5: Funciones de Activación

```
# Implementar las siguientes funciones de activación:

def sigmoid(x: np.ndarray) -> np.ndarray:
    """σ(x) = 1 / (1 + e^(-x))"""
    pass

def relu(x: np.ndarray) -> np.ndarray:
    """ReLU(x) = max(0, x)"""
    pass

def softmax(x: np.ndarray) -> np.ndarray:
    """softmax(x)_i = e^(x_i) / Σ e^(x_j)"""
    pass

# Verificar:
# sigmoid(0) ≈ 0.5
# relu(-5) = 0, relu(5) = 5
# softmax([1,2,3]).sum() ≈ 1.0
```

📦 Entregable del Módulo

Script: `benchmark_vectorization.py`

```
"""
Benchmark: Operaciones vectoriales Lista vs NumPy

Este script compara el rendimiento de operaciones comunes
usando listas de Python puras vs arrays de NumPy.

Operaciones comparadas:
1. Producto punto
2. Normalización
3. Distancia euclidiana
4. Suma de matrices

Autor: [Tu nombre]
Fecha: [Fecha]
"""

import numpy as np
import time
from typing import List, Tuple, Callable
from dataclasses import dataclass


@dataclass
class BenchmarkResult:
    """Resultado de un benchmark."""
    operation: str
    time_list: float
```

```
    time_numpy: float
    speedup: float


def benchmark(
    func_list: Callable,
    func_numpy: Callable,
    args_list: Tuple,
    args_numpy: Tuple,
    operation_name: str,
    iterations: int = 100
) -> BenchmarkResult:
    """Ejecuta benchmark comparativo."""

    # Benchmark lista
    start = time.time()
    for _ in range(iterations):
        func_list(*args_list)
    time_list = (time.time() - start) / iterations

    # Benchmark NumPy
    start = time.time()
    for _ in range(iterations):
        func_numpy(*args_numpy)
    time_numpy = (time.time() - start) / iterations

    return BenchmarkResult(
        operation=operation_name,
        time_list=time_list,
        time_numpy=time_numpy,
        speedup=time_list / time_numpy
    )


# === IMPLEMENTAR TUS FUNCIONES AQUÍ ===

def dot_product_list(a: List[float], b: List[float]) -> float:
    """Producto punto con listas."""
    # TODO: Implementar
    pass


def dot_product_numpy(a: np.ndarray, b: np.ndarray) -> float:
    """Producto punto con NumPy."""
    # TODO: Implementar
    pass


def normalize_list(data: List[float]) -> List[float]:
    """Normalizar con listas."""
    # TODO: Implementar
    pass


def normalize_numpy(data: np.ndarray) -> np.ndarray:
    """Normalizar con NumPy."""
    # TODO: Implementar
    pass


def euclidean_distance_list(a: List[float], b: List[float]) -> float:
    """Distancia euclidiana con listas."""
    # TODO: Implementar
    pass


def euclidean_distance_numpy(a: np.ndarray, b: np.ndarray) -> float:
    """Distancia euclidiana con NumPy."""
    # TODO: Implementar
    pass


def matrix_sum_list(A: List[List[float]], B: List[List[float]]) -> List[List[float]]:
    """Suma de matrices con listas."""
    # TODO: Implementar
```

```
    pass


def matrix_sum_numpy(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    """Suma de matrices con NumPy."""
    # TODO: Implementar
    pass


def main():
    """Ejecutar todos los benchmarks."""
    size = 10000

    # Crear datos de prueba
    list_a = [float(i) for i in range(size)]
    list_b = [float(i) for i in range(size)]
    array_a = np.array(list_a)
    array_b = np.array(list_b)

    matrix_size = 100
    list_matrix_a = [[float(i*j) for j in range(matrix_size)]
                     for i in range(matrix_size)]
    list_matrix_b = [[float(i+j) for j in range(matrix_size)]
                     for i in range(matrix_size)]
    array_matrix_a = np.array(list_matrix_a)
    array_matrix_b = np.array(list_matrix_b)

    # Ejecutar benchmarks
    results = []

    results.append(benchmark(
        dot_product_list, dot_product_numpy,
        (list_a, list_b), (array_a, array_b),
        "Producto Punto"
    ))

    results.append(benchmark(
        normalize_list, normalize_numpy,
        (list_a,), (array_a,),
        "Normalización"
    ))

    results.append(benchmark(
        euclidean_distance_list, euclidean_distance_numpy,
        (list_a, list_b), (array_a, array_b),
        "Distancia Euclidiana"
    ))

    results.append(benchmark(
        matrix_sum_list, matrix_sum_numpy,
        (list_matrix_a, list_matrix_b), (array_matrix_a, array_matrix_b),
        "Suma de Matrices"
    ))

    # Mostrar resultados
    print("\n" + "="*60)
    print("BENCHMARK: Lista vs NumPy")
    print("="*60)
    print(f"{'Operación':<25} {'Lista (ms)':<12} {'NumPy (ms)':<12} {'Speedup':<10}")
    print("-"*60)

    for r in results:
        print(f"{r.operation:<25} {r.time_list*1000:<12.4f} {r.time_numpy*1000:<12.4f} {r.speedup:<10.1f}x")

    print("="*60)
    print(f"\nSpeedup promedio: {sum(r.speedup for r in results)/len(results):.1f}x")


if __name__ == "__main__":
    main()
```

🐛 Debugging NumPy: Errores que te Harán Perder el Tiempo (v3.2)

> ⚠️ **CRÍTICO:** Estos 5 errores son los más frecuentes en las Fases 1 y 2. Resolverlos ahora previene horas de frustración.

Error 1: Shape Mismatch - `(5,)` vs `(5,1)`

```
import numpy as np

# PROBLEMA: Vector 1D vs Vector Columna
v1 = np.array([1, 2, 3, 4, 5])      # Shape: (5,) - Vector 1D
v2 = np.array([[1], [2], [3], [4], [5]])  # Shape: (5, 1) - Vector columna

print(f"v1.shape: {v1.shape}")  # (5,)
print(f"v2.shape: {v2.shape}")  # (5, 1)

# ESTO FALLA en Regresión Lineal:
# Si X tiene shape (100, 5) y theta tiene shape (5,), el resultado es (100,)
# Si theta tiene shape (5, 1), el resultado es (100, 1)

# SOLUCIÓN: Usar reshape o keepdims
v1_columna = v1.reshape(-1, 1)  # (5,) → (5, 1)
v1_columna_alt = v1[:, np.newaxis]  # Alternativa

# REGLA: Para ML, los vectores de features deben ser (n, 1), no (n,)
```

Error 2: Broadcasting Silencioso Incorrecto

```
import numpy as np

# PROBLEMA: Broadcasting no falla, pero da resultados incorrectos
X = np.random.randn(100, 5)  # 100 samples, 5 features
mean_wrong = np.mean(X)      # ¡INCORRECTO! Media de TODO el array
mean_correct = np.mean(X, axis=0)  # Correcto: media por feature (shape: (5,))

print(f"mean_wrong shape: {np.array(mean_wrong).shape}")  # () - escalar
print(f"mean_correct shape: {mean_correct.shape}")  # (5,)

# REGLA: Siempre especifica axis= en agregaciones
# axis=0: opera sobre filas (resultado por columna)
# axis=1: opera sobre columnas (resultado por fila)
```

Error 3: Modificación In-Place Inesperada

```
import numpy as np

# PROBLEMA: Los slices de NumPy son VISTAS, no copias
original = np.array([1, 2, 3, 4, 5])
slice_view = original[1:4]
slice_view[0] = 999

print(original)  # [1, 999, 3, 4, 5] - ¡ORIGINAL MODIFICADO!

# SOLUCIÓN: Usar .copy() explícitamente
original = np.array([1, 2, 3, 4, 5])
slice_copy = original[1:4].copy()
slice_copy[0] = 999

print(original)  # [1, 2, 3, 4, 5] - Original intacto
```

Error 4: División por Cero en Normalización

```
import numpy as np

# PROBLEMA: División por cero cuando std = 0
data = np.array([5, 5, 5, 5, 5])
std = np.std(data)  # 0.0
normalized = (data - np.mean(data)) / std  # RuntimeWarning: divide by zero

# SOLUCIÓN: Añadir epsilon
epsilon = 1e-8
normalized_safe = (data - np.mean(data)) / (std + epsilon)
```

```
# REGLA: Siempre usar epsilon en divisiones (especialmente en softmax, normalizaciones)
```

Error 5: Tipos de Datos Incorrectos

```
import numpy as np

# PROBLEMA: Operaciones con int cuando necesitas float
a = np.array([1, 2, 3])  # dtype: int64
b = a / 2  # dtype: float64 (OK en Python 3)

# PERO en operaciones in-place:
a = np.array([1, 2, 3])
a /= 2  # a sigue siendo int64, se trunca!
print(a)  # [0, 1, 1] - ¡TRUNCADO!

# SOLUCIÓN: Especificar dtype al crear
a = np.array([1, 2, 3], dtype=np.float64)
a /= 2
print(a)  # [0.5, 1.0, 1.5] - Correcto

# REGLA: Para ML, siempre usar dtype=np.float64 o np.float32
```

🛠️ Estándares de Código Profesional (v3.2)

> 💎 **Filosofía v3.2:** El código no se considera terminado hasta que pase `mypy`, `ruff` y `pytest`.

Configuración del Entorno Profesional

```
# Crear entorno virtual
python -m venv .venv
source .venv/bin/activate  # Linux/Mac
# .venv\Scripts\activate   # Windows

# Instalar herramientas de calidad
pip install numpy pandas matplotlib
pip install mypy ruff pytest

# Archivo pyproject.toml (crear en la raíz del proyecto)
```

```
# pyproject.toml
[tool.mypy]
python_version = "3.11"
warn_return_any = true
warn_unused_ignores = true
disallow_untyped_defs = true

[tool.ruff]
line-length = 100
select = ["E", "F", "W", "I", "UP"]

[tool.pytest.ini_options]
testpaths = ["tests"]
python_files = "test_*.py"
```

Ejemplo: Código con Type Hints

```
# src/linear_algebra.py
"""Operaciones de álgebra lineal desde cero."""
import numpy as np
from numpy.typing import NDArray


def dot_product(a: NDArray[np.float64], b: NDArray[np.float64]) -> float:
    """
    Calcula el producto punto de dos vectores.

    Args:
        a: Primer vector (n,)
        b: Segundo vector (n,)
```

```
    Returns:
        El producto punto (escalar)

    Raises:
        ValueError: Si los vectores tienen shapes diferentes
    """
    if a.shape != b.shape:
        raise ValueError(f"Shapes incompatibles: {a.shape} vs {b.shape}")
    return float(np.sum(a * b))


def norm_l2(v: NDArray[np.float64]) -> float:
    """Calcula la norma L2 (euclidiana) de un vector."""
    return float(np.sqrt(np.sum(v ** 2)))
```

Ejemplo: Tests con pytest

```
# tests/test_linear_algebra.py
"""Tests unitarios para linear_algebra.py"""
import numpy as np
import pytest
from src.linear_algebra import dot_product, norm_l2


class TestDotProduct:
    """Tests para la función dot_product."""

    def test_dot_product_basic(self) -> None:
        """Test básico: [1,2,3] · [4,5,6] = 32"""
        a = np.array([1.0, 2.0, 3.0])
        b = np.array([4.0, 5.0, 6.0])
        assert dot_product(a, b) == 32.0

    def test_dot_product_orthogonal(self) -> None:
        """Vectores ortogonales tienen producto punto = 0"""
        a = np.array([1.0, 0.0])
        b = np.array([0.0, 1.0])
        assert dot_product(a, b) == 0.0

    def test_dot_product_shape_mismatch(self) -> None:
        """Debe lanzar ValueError si shapes no coinciden"""
        a = np.array([1.0, 2.0])
        b = np.array([1.0, 2.0, 3.0])
        with pytest.raises(ValueError):
            dot_product(a, b)


class TestNormL2:
    """Tests para la función norm_l2."""

    def test_norm_unit_vector(self) -> None:
        """Vector unitario tiene norma 1"""
        v = np.array([1.0, 0.0, 0.0])
        assert norm_l2(v) == 1.0

    def test_norm_345(self) -> None:
        """Triángulo 3-4-5: norma de [3,4] = 5"""
        v = np.array([3.0, 4.0])
        assert norm_l2(v) == 5.0
```

Comandos de Verificación

```
# Ejecutar en la raíz del proyecto:

# 1. Verificar tipos (mypy)
mypy src/

# 2. Verificar estilo (ruff)
ruff check src/
ruff format src/  # Auto-formatear

# 3. Ejecutar tests (pytest)
pytest tests/ -v
```

```
# 4. Todo junto (antes de cada commit)
mypy src/ && ruff check src/ && pytest tests/ -v
```

🎯 El Reto del Tablero Blanco (Metodología Feynman)

> 📝 **Instrucción:** Después de implementar código, debes poder explicar el algoritmo en **máximo 5 líneas** sin usar jerga técnica. Si no puedes, vuelve a la teoría.

Ejemplo: Broadcasting

❌ **Explicación técnica (mala):**
"Broadcasting es la capacidad de NumPy de realizar operaciones elemento a elemento entre arrays de diferentes shapes mediante la expansión implícita de dimensiones según reglas de compatibilidad."

✅ **Explicación Feynman (buena):**
"Cuando sumas un número a una lista, NumPy automáticamente suma ese número a CADA elemento. Es como si el número se 'copiara' para que tenga el mismo tamaño que la lista. Lo mismo pasa entre listas de diferentes tamaños, siempre que una de ellas tenga tamaño 1 en alguna dimensión."

Tu Reto para el Módulo 01:

Explica en 5 líneas o menos:

1. ¿Por qué NumPy es más rápido que listas de Python?
2. ¿Qué significa `axis=0` vs `axis=1`?
3. ¿Por qué `.copy()` es importante?

✅ Checklist de Finalización (v3.2)

Conocimiento

- [ ] Puedo crear arrays 1D, 2D y 3D con NumPy
- [ ] Entiendo indexing y slicing de arrays
- [ ] Puedo explicar broadcasting y usarlo
- [ ] Sé calcular agregaciones por eje (axis)
- [ ] Puedo reescribir loops como operaciones vectorizadas
- [ ] Conozco las diferencias entre `@`, `np.dot`, `np.matmul`
- [ ] Conozco los 5 errores comunes de NumPy y sus soluciones

Entregables de Código

- [ ] `benchmark_vectorization.py` implementado
- [ ] El speedup de NumPy vs lista es > 50x en mis pruebas
- [ ] `mypy src/` pasa sin errores
- [ ] `ruff check src/` pasa sin errores
- [ ] Al menos 3 tests con `pytest` pasando

Metodología Feynman

- [ ] Puedo explicar broadcasting en 5 líneas sin jerga
- [ ] Puedo explicar axis=0 vs axis=1 en 5 líneas sin jerga
- [ ] Puedo explicar por qué .copy() es importante

🔗 Navegación

Anterior	Índice	Siguiente
-	00_INDICE	02_ALGEBRA_LINEAL_ML

MÓDULO 02 - ÁLGEBRA LINEAL PARA ML

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 02 - Álgebra Lineal para Machine Learning

🎯 **Objetivo:** Dominar vectores, matrices, normas y eigenvalues para ML
Fase: 1 - Fundamentos Matemáticos | **Semanas** 3 - 5
Prerrequisitos: Módulo 0 1 (Python Científico con NumPy)

🧠 ¿Por Qué Álgebra Lineal para ML?

Conexiones Directas con el Pathway

Concepto	Uso en ML	Curso del Pathway
Producto punto	Similitud, predicciones	Supervised Learning
Normas L 1 /L 2	Regularización, distancias	Supervised Learning
Eigenvalues	PCA, reducción dimensional	Unsupervised Learning
Multiplicación matricial	Forward pass en redes	Deep Learning
SVD	Compresión, PCA	Unsupervised Learning

La Matemática Detrás de ML

```
Regresión Lineal:     ŷ = Xθ           (multiplicación matriz-vector)
Logistic Regression:  ŷ = σ(Xθ)        (+ función de activación)
Neural Network:       ŷ = σ(W₃σ(W₂σ(W₁x)))  (capas de multiplicaciones)
PCA:                  X_reduced = XV    (proyección a eigenvectors)
```

📚 Contenido del Módulo

Semana 3: Vectores y Operaciones Básicas

Semana 4: Normas y Distancias

Semana 5: Matrices, Eigenvalues y SVD

💻 Parte 1: Vectores

1.1 Definición Geométrica y Algebraica

```
import numpy as np
import matplotlib.pyplot as plt

# Un vector es una lista ordenada de números
# Geométricamente: flecha con dirección y magnitud

# Vector en R² (2 dimensiones)
v = np.array([3, 4])

# Vector en R³ (3 dimensiones)
w = np.array([1, 2, 3])

# Vector en R^n (n dimensiones) - común en ML
# Ejemplo: imagen 28x28 = 784 dimensiones
image_vector = np.random.randn(784)

# Visualización 2D
def plot_vector(v, origin=[0, 0], color='blue', label=None):
    """Dibuja un vector desde el origen."""
    plt.quiver(*origin, *v, angles='xy', scale_units='xy', scale=1, color=color, label=label)

plt.figure(figsize=(8, 8))
plot_vector(np.array([3, 4]), color='blue', label='v = [3, 4]')
plot_vector(np.array([2, 1]), color='red', label='w = [2, 1]')
plt.xlim(-1, 5)
plt.ylim(-1, 5)
plt.grid(True)
plt.axhline(y=0, color='k', linewidth=0.5)
plt.axvline(x=0, color='k', linewidth=0.5)
plt.legend()
```

```
plt.title('Vectores en R²')
plt.show()
```

1.2 Operaciones con Vectores

```
import numpy as np

# Vectores de ejemplo
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# === SUMA DE VECTORES ===
# (a + b)ᵢ = aᵢ + bᵢ
suma = a + b
print(f"a + b = {suma}")  # [5, 7, 9]

# === RESTA DE VECTORES ===
resta = a - b
print(f"a - b = {resta}")  # [-3, -3, -3]

# === MULTIPLICACIÓN POR ESCALAR ===
# (c·a)ᵢ = c·aᵢ
escalar = 2 * a
print(f"2·a = {escalar}")  # [2, 4, 6]

# === PRODUCTO PUNTO (DOT PRODUCT) ===
# a·b = Σᵢ aᵢ·bᵢ
# Resultado: escalar
dot = np.dot(a, b)
print(f"a·b = {dot}")  # 1*4 + 2*5 + 3*6 = 32

# Alternativamente:
dot_alt = a @ b
dot_sum = np.sum(a * b)
print(f"Verificación: {dot_alt}, {dot_sum}")
```

1.3 Interpretación Geométrica del Producto Punto

```
import numpy as np

def angle_between_vectors(a: np.ndarray, b: np.ndarray) -> float:
    """
    Calcula el ángulo entre dos vectores.

    cos(θ) = (a·b) / (||a|| ||b||)
    """
    cos_theta = np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
    # Clip para evitar errores numéricos fuera de [-1, 1]
    cos_theta = np.clip(cos_theta, -1, 1)
    theta_rad = np.arccos(cos_theta)
    theta_deg = np.degrees(theta_rad)
    return theta_deg

# Ejemplos
v1 = np.array([1, 0])
v2 = np.array([0, 1])
v3 = np.array([1, 1])
v4 = np.array([-1, 0])

print(f"Ángulo entre [1,0] y [0,1]: {angle_between_vectors(v1, v2):.0f}°")  # 90°
print(f"Ángulo entre [1,0] y [1,1]: {angle_between_vectors(v1, v3):.0f}°")  # 45°
print(f"Ángulo entre [1,0] y [-1,0]: {angle_between_vectors(v1, v4):.0f}°") # 180°

# Interpretación para ML:
# - Producto punto alto → vectores similares (mismo "sentido")
# - Producto punto ≈ 0 → vectores ortogonales (independientes)
# - Producto punto negativo → vectores opuestos
```

1.4 Proyección de Vectores

```
import numpy as np

def project(a: np.ndarray, b: np.ndarray) -> np.ndarray:
```

```
    """
    Proyecta el vector a sobre el vector b.

    proj_b(a) = (a·b / b·b) · b

    Útil para: PCA, regresión, descomposición de señales
    """
    scalar = np.dot(a, b) / np.dot(b, b)
    return scalar * b

# Ejemplo
a = np.array([3, 4])
b = np.array([1, 0])  # Vector unitario en x

proyeccion = project(a, b)
print(f"Proyección de {a} sobre {b}: {proyeccion}")  # [3, 0]

# La proyección nos da "cuánto" de a está en la dirección de b
```

💻 Parte 2: Normas y Distancias

2.1 Norma L2 (Euclidiana)

```
import numpy as np

def l2_norm(x: np.ndarray) -> float:
    """
    Norma L2 (Euclidiana): longitud del vector.

    ||x||₂ = √(Σᵢ xᵢ²)

    Uso en ML:
    - Regularización Ridge
    - Normalización de vectores
    - Distancia euclidiana
    """
    return np.sqrt(np.sum(x ** 2))

# Equivalente en NumPy
x = np.array([3, 4])
print(f"||x||₂ = {l2_norm(x)}")           # 5.0
print(f"NumPy:  {np.linalg.norm(x)}")     # 5.0
print(f"NumPy:  {np.linalg.norm(x, 2)}")  # 5.0 (especificando ord=2)

# Vector unitario (normalizado)
def normalize(x: np.ndarray) -> np.ndarray:
    """Convierte vector a longitud 1."""
    return x / np.linalg.norm(x)

x_unit = normalize(x)
print(f"Unitario: {x_unit}")  # [0.6, 0.8]
print(f"Norma del unitario: {np.linalg.norm(x_unit)}")  # 1.0
```

2.2 Norma L1 (Manhattan)

```
import numpy as np

def l1_norm(x: np.ndarray) -> float:
    """
    Norma L1 (Manhattan): suma de valores absolutos.

    ||x||₁ = Σᵢ |xᵢ|

    Uso en ML:
    - Regularización Lasso (promueve sparsity)
    - Robustez a outliers
    """
    return np.sum(np.abs(x))

x = np.array([3, -4, 5])
print(f"||x||₁ = {l1_norm(x)}")                 # 12
print(f"NumPy:  {np.linalg.norm(x, 1)}")        # 12.0

# Comparación L1 vs L2
```

```
# L1 penaliza todos los valores igualmente
# L2 penaliza más los valores grandes (cuadrado)
```

2.3 Norma L∞ (Máximo)

```
import numpy as np

def linf_norm(x: np.ndarray) -> float:
    """
    Norma L∞: máximo valor absoluto.

    ||x||∞ = max(|xᵢ|)
    """
    return np.max(np.abs(x))

x = np.array([3, -7, 5])
print(f"||x||∞ = {linf_norm(x)}")            # 7
print(f"NumPy:  {np.linalg.norm(x, np.inf)}") # 7.0
```

2.4 Distancia Euclidiana

```
import numpy as np

def euclidean_distance(a: np.ndarray, b: np.ndarray) -> float:
    """
    Distancia Euclidiana entre dos puntos.

    d(a, b) = ||a - b||₂ = √(Σᵢ (aᵢ - bᵢ)²)

    Uso en ML:
    - KNN (k-nearest neighbors)
    - K-Means (asignación a clusters)
    - Evaluación de similaridad
    """
    return np.linalg.norm(a - b)

# Ejemplo
p1 = np.array([0, 0])
p2 = np.array([3, 4])
print(f"Distancia: {euclidean_distance(p1, p2)}")  # 5.0

# Para múltiples puntos (eficiente)
def pairwise_distances(X: np.ndarray) -> np.ndarray:
    """
    Calcula matriz de distancias entre todos los puntos.
    X: matriz (n_samples, n_features)
    Retorna: matriz (n_samples, n_samples)
    """
    # Usando broadcasting
    # ||a - b||² = ||a||² + ||b||² - 2(a·b)
    sq_norms = np.sum(X ** 2, axis=1)
    distances_sq = sq_norms[:, np.newaxis] + sq_norms[np.newaxis, :] - 2 * X @ X.T
    distances_sq = np.maximum(distances_sq, 0)  # Evitar negativos por errores numéricos
    return np.sqrt(distances_sq)

# Test
X = np.array([[0, 0], [3, 4], [1, 1]])
D = pairwise_distances(X)
print("Matriz de distancias:")
print(D)
```

2.5 Similitud Coseno

```
import numpy as np

def cosine_similarity(a: np.ndarray, b: np.ndarray) -> float:
    """
    Similitud coseno: mide el ángulo entre vectores.

    sim(a, b) = (a·b) / (||a|| ||b||)

    Rango: [-1, 1]
    - 1: vectores idénticos (misma dirección)
```

```
    - 0: vectores ortogonales
    - -1: vectores opuestos

    Uso en ML:
    - NLP (similitud de documentos)
    - Sistemas de recomendación
    - Embeddings
    """
    dot_product = np.dot(a, b)
    norm_a = np.linalg.norm(a)
    norm_b = np.linalg.norm(b)

    if norm_a == 0 or norm_b == 0:
        return 0.0

    return dot_product / (norm_a * norm_b)

def cosine_distance(a: np.ndarray, b: np.ndarray) -> float:
    """Distancia coseno = 1 - similitud coseno."""
    return 1 - cosine_similarity(a, b)

# Ejemplos
v1 = np.array([1, 0, 0])
v2 = np.array([1, 0, 0])
v3 = np.array([0, 1, 0])
v4 = np.array([-1, 0, 0])

print(f"Similitud (idénticos):  {cosine_similarity(v1, v2)}")   # 1.0
print(f"Similitud (ortogonales): {cosine_similarity(v1, v3)}")  # 0.0
print(f"Similitud (opuestos):    {cosine_similarity(v1, v4)}")  # -1.0
```

💻 Parte 3: Matrices

3.1 Operaciones Básicas

```
import numpy as np

# Crear matrices
A = np.array([
    [1, 2, 3],
    [4, 5, 6]
])  # Shape: (2, 3)

B = np.array([
    [7, 8],
    [9, 10],
    [11, 12]
])  # Shape: (3, 2)

# === SUMA Y RESTA ===
# Solo para matrices del mismo shape
C = np.array([[1, 2, 3], [4, 5, 6]])
print(f"A + C =\n{A + C}")

# === MULTIPLICACIÓN POR ESCALAR ===
print(f"2·A =\n{2 * A}")

# === PRODUCTO MATRICIAL ===
# (m×n) @ (n×p) = (m×p)
# A(2×3) @ B(3×2) = (2×2)
AB = A @ B
print(f"A @ B =\n{AB}")
# [[58, 64],
#  [139, 154]]

# Verificación manual del elemento [0,0]:
# 1*7 + 2*9 + 3*11 = 7 + 18 + 33 = 58 ✓

# === TRANSPUESTA ===
print(f"A^T =\n{A.T}")
# [[1, 4],
#  [2, 5],
#  [3, 6]]
```

3.2 Matriz por Vector (Transformación Lineal)

```
import numpy as np

# La multiplicación matriz-vector es una TRANSFORMACIÓN LINEAL
# y = Ax transforma el vector x al espacio de y

# Ejemplo: Rotación 90° en R²
theta = np.pi / 2  # 90 grados
R = np.array([
    [np.cos(theta), -np.sin(theta)],
    [np.sin(theta),  np.cos(theta)]
])

x = np.array([1, 0])
y = R @ x
print(f"Rotar [1,0] 90°: {y}")  # [0, 1]

# En ML: y = Wx + b (capa de red neuronal)
W = np.random.randn(10, 784)  # Pesos: 784 entradas → 10 salidas
b = np.random.randn(10)        # Bias
x = np.random.randn(784)       # Input (imagen aplanada)

y = W @ x + b  # Output de la capa
print(f"Shape de y: {y.shape}")  # (10,)
```

3.3 Matriz Inversa

```
import numpy as np

def safe_inverse(A: np.ndarray) -> np.ndarray:
    """
    Calcula la inversa de A si existe.
    A @ A⁻¹ = A⁻¹ @ A = I

    Uso en ML:
    - Solución cerrada de regresión lineal: θ = (X^T X)⁻¹ X^T y
    - Whitening en PCA
    """
    try:
        return np.linalg.inv(A)
    except np.linalg.LinAlgError:
        print("Matriz no invertible (singular)")
        return None

# Ejemplo
A = np.array([
    [4, 7],
    [2, 6]
])

A_inv = safe_inverse(A)
print(f"A⁻¹ =\n{A_inv}")

# Verificar: A @ A⁻¹ = I
identity = A @ A_inv
print(f"A @ A⁻¹ ≈ I:\n{np.round(identity, 10)}")

# NOTA: En ML, evita calcular inversas cuando sea posible
# Usa np.linalg.solve() en su lugar (más estable numéricamente)
```

3.4 Solución de Sistemas Lineales

```
import numpy as np

# Sistema: Ax = b
# Encontrar x

A = np.array([
    [3, 1],
    [1, 2]
])
b = np.array([9, 8])
```

```
# Método 1: Inversa (NO RECOMENDADO)
x_inv = np.linalg.inv(A) @ b

# Método 2: solve (RECOMENDADO - más estable)
x_solve = np.linalg.solve(A, b)

print(f"Solución: x = {x_solve}")  # [2, 3]

# Verificar
print(f"A @ x = {A @ x_solve}")    # [9, 8] ✓
```

💻 Parte 4: Eigenvalues y Eigenvectors

4.1 Concepto

```
import numpy as np

"""
EIGENVALUES (Autovalores) y EIGENVECTORS (Autovectores)

Definición: Av = λv
- v: eigenvector (vector que solo se escala, no cambia dirección)
- λ: eigenvalue (factor de escala)

Interpretación:
- Los eigenvectors son las "direcciones principales" de una transformación
- Los eigenvalues indican cuánto se estira/comprime en cada dirección

Uso en ML:
- PCA: eigenvectors de la matriz de covarianza son las componentes principales
- PageRank: eigenvector dominante de la matriz de transición
- Estabilidad de sistemas dinámicos
"""

# Ejemplo simple
A = np.array([
    [2, 1],
    [1, 2]
])

# Calcular eigenvalues y eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print(f"Eigenvalues: {eigenvalues}")    # [3, 1]
print(f"Eigenvectors:\n{eigenvectors}") # columnas son los eigenvectors

# Verificar: Av = λv
v1 = eigenvectors[:, 0]  # primer eigenvector
lambda1 = eigenvalues[0]  # primer eigenvalue

Av = A @ v1
lambda_v = lambda1 * v1

print(f"\nVerificación Av = λv:")
print(f"Av     = {Av}")
print(f"λv     = {lambda_v}")
print(f"¿Iguales? {np.allclose(Av, lambda_v)}")
```

4.2 Eigenvalues para PCA

```
import numpy as np

def pca_via_eigen(X: np.ndarray, n_components: int) -> tuple:
    """
    PCA usando eigendecomposition de la matriz de covarianza.

    Args:
        X: datos (n_samples, n_features)
        n_components: número de componentes a retener

    Returns:
        X_transformed: datos proyectados
        components: eigenvectors (componentes principales)
```

```
        explained_variance: varianza explicada por cada componente
    """
    # 1. Centrar datos (restar media)
    X_centered = X - np.mean(X, axis=0)

    # 2. Calcular matriz de covarianza
    # Cov = (1/n) X^T X
    n_samples = X.shape[0]
    cov_matrix = (X_centered.T @ X_centered) / (n_samples - 1)

    # 3. Calcular eigenvalues y eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # 4. Ordenar por eigenvalue (mayor a menor)
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # 5. Seleccionar top n_components
    components = eigenvectors[:, :n_components].real

    # 6. Proyectar datos
    X_transformed = X_centered @ components

    # 7. Calcular varianza explicada
    total_variance = np.sum(eigenvalues)
    explained_variance = eigenvalues[:n_components].real / total_variance

    return X_transformed, components, explained_variance

# Demo
np.random.seed(42)
X = np.random.randn(100, 5)  # 100 muestras, 5 features

X_pca, components, var_explained = pca_via_eigen(X, n_components=2)

print(f"Shape original: {X.shape}")
print(f"Shape reducido: {X_pca.shape}")
print(f"Varianza explicada: {var_explained}")
print(f"Varianza total explicada: {np.sum(var_explained):.2%}")
```

💻 Parte 5: SVD (Singular Value Decomposition)

5.1 Concepto

```
import numpy as np

"""
SVD: Singular Value Decomposition

A = U Σ V^T

- U: matriz ortogonal (m×m) - vectores singulares izquierdos
- Σ: matriz diagonal (m×n) - valores singulares (σ₁ ≥ σ₂ ≥ ... ≥ 0)
- V^T: matriz ortogonal (n×n) - vectores singulares derechos

Ventajas sobre Eigendecomposition:
- Funciona para CUALQUIER matriz (no solo cuadradas)
- Más estable numéricamente
- Los valores singulares siempre son no-negativos

Uso en ML:
- PCA (método preferido)
- Compresión de imágenes
- Sistemas de recomendación (matrix factorization)
- Regularización (truncated SVD)
"""

# Ejemplo
A = np.array([
    [1, 2],
    [3, 4],
    [5, 6]
])  # 3×2
```

```
U, S, Vt = np.linalg.svd(A, full_matrices=False)

print(f"U shape: {U.shape}")   # (3, 2)
print(f"S shape: {S.shape}")   # (2,) - valores singulares
print(f"Vt shape: {Vt.shape}") # (2, 2)

# Reconstruir A
A_reconstructed = U @ np.diag(S) @ Vt
print(f"\n¿A ≈ U Σ V^T? {np.allclose(A, A_reconstructed)}")
```

5.2 PCA via SVD (Método Preferido)

```
import numpy as np

def pca_via_svd(X: np.ndarray, n_components: int) -> tuple:
    """
    PCA usando SVD (más estable que eigendecomposition).

    La relación: si X = UΣV^T, entonces:
    - V contiene las componentes principales
    - Σ²/(n-1) son las varianzas (eigenvalues de X^TX)
    """
    # 1. Centrar datos
    X_centered = X - np.mean(X, axis=0)

    # 2. SVD
    U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

    # 3. Componentes principales (filas de Vt)
    components = Vt[:n_components]

    # 4. Proyectar datos
    X_transformed = X_centered @ components.T

    # 5. Varianza explicada
    variance = (S ** 2) / (X.shape[0] - 1)
    explained_variance_ratio = variance[:n_components] / np.sum(variance)

    return X_transformed, components, explained_variance_ratio

# Demo
np.random.seed(42)
X = np.random.randn(100, 10)

X_pca, components, var_ratio = pca_via_svd(X, n_components=3)

print(f"Varianza explicada por componente: {var_ratio}")
print(f"Varianza total explicada: {np.sum(var_ratio):.2%}")
```

5.3 Compresión de Imágenes con SVD

```
import numpy as np

def compress_image_svd(image: np.ndarray, k: int) -> np.ndarray:
    """
    Comprime una imagen usando truncated SVD.

    Args:
        image: matriz 2D (grayscale) o 3D (RGB)
        k: número de valores singulares a retener

    Returns:
        imagen comprimida
    """
    if len(image.shape) == 2:
        # Grayscale
        U, S, Vt = np.linalg.svd(image, full_matrices=False)
        compressed = U[:, :k] @ np.diag(S[:k]) @ Vt[:k, :]
        return np.clip(compressed, 0, 255).astype(np.uint8)
    else:
        # RGB: comprimir cada canal
        compressed = np.zeros_like(image)
```

```
        for i in range(3):
            compressed[:, :, i] = compress_image_svd(image[:, :, i], k)
        return compressed

def compression_ratio(original_shape: tuple, k: int) -> float:
    """Calcula ratio de compresión."""
    m, n = original_shape[:2]
    original_size = m * n
    compressed_size = k * (m + n + 1)  # U[:,:k], S[:k], Vt[:k,:]
    return compressed_size / original_size

# Demo (sin cargar imagen real)
# Simular imagen 100x100
image = np.random.randint(0, 256, (100, 100), dtype=np.uint8)

for k in [5, 10, 20, 50]:
    compressed = compress_image_svd(image, k)
    ratio = compression_ratio(image.shape, k)
    print(f"k={k}: ratio={ratio:.2%}")
```

📦 Entregable del Módulo

Librería: `linear_algebra.py`

```
"""
Linear Algebra Library for Machine Learning

Implementación desde cero de operaciones fundamentales.
Usando NumPy para eficiencia pero entendiendo las matemáticas.

Autor: [Tu nombre]
Módulo: 02 - Álgebra Lineal para ML
"""

import numpy as np
from typing import Tuple, Optional


# ============================================================
# PARTE 1: OPERACIONES CON VECTORES
# ============================================================

def dot_product(a: np.ndarray, b: np.ndarray) -> float:
    """
    Producto punto de dos vectores.

    a·b = Σᵢ aᵢ·bᵢ
    """
    assert a.shape == b.shape, "Vectores deben tener mismo shape"
    return float(np.sum(a * b))


def vector_angle(a: np.ndarray, b: np.ndarray) -> float:
    """
    Ángulo entre dos vectores en grados.

    cos(θ) = (a·b) / (||a|| ||b||)
    """
    cos_theta = dot_product(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
    cos_theta = np.clip(cos_theta, -1, 1)
    return float(np.degrees(np.arccos(cos_theta)))


def project_vector(a: np.ndarray, b: np.ndarray) -> np.ndarray:
    """
    Proyección del vector a sobre el vector b.

    proj_b(a) = (a·b / b·b) · b
    """
    scalar = dot_product(a, b) / dot_product(b, b)
    return scalar * b


# ============================================================
```

```
# PARTE 2: NORMAS
# ============================================================

def l1_norm(x: np.ndarray) -> float:
    """Norma L1 (Manhattan): ||x||₁ = Σ|xᵢ|"""
    return float(np.sum(np.abs(x)))


def l2_norm(x: np.ndarray) -> float:
    """Norma L2 (Euclidiana): ||x||₂ = √(Σxᵢ²)"""
    return float(np.sqrt(np.sum(x ** 2)))


def linf_norm(x: np.ndarray) -> float:
    """Norma L∞ (Máximo): ||x||∞ = max|xᵢ|"""
    return float(np.max(np.abs(x)))


def normalize(x: np.ndarray, ord: int = 2) -> np.ndarray:
    """Normaliza vector a norma 1."""
    norm = np.linalg.norm(x, ord=ord)
    if norm == 0:
        return x
    return x / norm


# ============================================================
# PARTE 3: DISTANCIAS
# ============================================================

def euclidean_distance(a: np.ndarray, b: np.ndarray) -> float:
    """Distancia Euclidiana: d(a,b) = ||a-b||₂"""
    return l2_norm(a - b)


def manhattan_distance(a: np.ndarray, b: np.ndarray) -> float:
    """Distancia Manhattan: d(a,b) = ||a-b||₁"""
    return l1_norm(a - b)


def cosine_similarity(a: np.ndarray, b: np.ndarray) -> float:
    """
    Similitud coseno: sim(a,b) = (a·b) / (||a|| ||b||)
    Rango: [-1, 1]
    """
    norm_a = l2_norm(a)
    norm_b = l2_norm(b)
    if norm_a == 0 or norm_b == 0:
        return 0.0
    return dot_product(a, b) / (norm_a * norm_b)


def cosine_distance(a: np.ndarray, b: np.ndarray) -> float:
    """Distancia coseno: 1 - similitud_coseno"""
    return 1 - cosine_similarity(a, b)


def pairwise_euclidean(X: np.ndarray) -> np.ndarray:
    """
    Matriz de distancias euclidianas entre todos los pares.

    Args:
        X: matriz (n_samples, n_features)
    Returns:
        D: matriz (n_samples, n_samples) de distancias
    """
    sq_norms = np.sum(X ** 2, axis=1)
    D_sq = sq_norms[:, np.newaxis] + sq_norms[np.newaxis, :] - 2 * X @ X.T
    D_sq = np.maximum(D_sq, 0)  # Evitar negativos por errores numéricos
    return np.sqrt(D_sq)


# ============================================================
# PARTE 4: EIGENVALUES Y PCA
# ============================================================
```

```
def eigendecomposition(A: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """
    Calcula eigenvalues y eigenvectors, ordenados por eigenvalue descendente.

    Returns:
        eigenvalues: array de eigenvalues (ordenados)
        eigenvectors: matriz donde columna i es el eigenvector i
    """
    eigenvalues, eigenvectors = np.linalg.eig(A)

    # Ordenar por eigenvalue descendente
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx].real
    eigenvectors = eigenvectors[:, idx].real

    return eigenvalues, eigenvectors


def pca(X: np.ndarray, n_components: int) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Principal Component Analysis via SVD.

    Args:
        X: datos (n_samples, n_features)
        n_components: número de componentes

    Returns:
        X_transformed: datos proyectados (n_samples, n_components)
        components: componentes principales (n_components, n_features)
        explained_variance_ratio: proporción de varianza explicada
    """
    # Centrar datos
    X_centered = X - np.mean(X, axis=0)

    # SVD
    U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

    # Componentes principales
    components = Vt[:n_components]

    # Proyectar
    X_transformed = X_centered @ components.T

    # Varianza explicada
    variance = (S ** 2) / (X.shape[0] - 1)
    explained_variance_ratio = variance[:n_components] / np.sum(variance)

    return X_transformed, components, explained_variance_ratio


# ============================================================
# TESTS
# ============================================================

def run_tests():
    """Ejecuta tests básicos."""
    print("Ejecutando tests...")

    # Test producto punto
    a = np.array([1, 2, 3])
    b = np.array([4, 5, 6])
    assert abs(dot_product(a, b) - 32) < 1e-10
    print("✓ dot_product")

    # Test normas
    x = np.array([3, 4])
    assert abs(l2_norm(x) - 5) < 1e-10
    assert abs(l1_norm(x) - 7) < 1e-10
    print("✓ normas")

    # Test distancias
    p1 = np.array([0, 0])
    p2 = np.array([3, 4])
    assert abs(euclidean_distance(p1, p2) - 5) < 1e-10
```

```
    print("✓ distancias")

    # Test similitud coseno
    v1 = np.array([1, 0])
    v2 = np.array([1, 0])
    v3 = np.array([0, 1])
    assert abs(cosine_similarity(v1, v2) - 1) < 1e-10
    assert abs(cosine_similarity(v1, v3)) < 1e-10
    print("✓ cosine_similarity")

    # Test PCA
    np.random.seed(42)
    X = np.random.randn(50, 10)
    X_pca, _, var_ratio = pca(X, 3)
    assert X_pca.shape == (50, 3)
    assert np.sum(var_ratio) <= 1.0
    print("✓ PCA")

    print("\n¡Todos los tests pasaron!")


if __name__ == "__main__":
    run_tests()
```

✅ Checklist de Finalización

- [ ] Puedo calcular producto punto y explicar su significado geométrico
- [ ] Entiendo las diferencias entre normas L_1, L_2, L_∞
- [ ] Puedo calcular distancia euclidiana y similitud coseno
- [ ] Sé multiplicar matrices y entiendo las dimensiones resultantes
- [ ] Puedo explicar qué son eigenvalues/eigenvectors y su uso en PCA
- [ ] Entiendo SVD y puedo usarlo para compresión/PCA
- [ ] Implementé `linear_algebra.py` con todos los tests pasando
- [ ] Puedo proyectar datos usando PCA y explicar varianza explicada

🔗 Navegación

Anterior	Índice	Siguiente
01_PYTHON_CIENTIFICO	00_INDICE	03_CALCULO_MULTIVARIANTE

MÓDULO 03 - CÁLCULO MULTIVARIANTE

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 03 - Cálculo Multivariante para Deep Learning

🎯 **Objetivo:** Dominar derivadas, gradientes y la Chain Rule para entender Backpropagation
Fase: 1 - Fundamentos Matemáticos | **Semanas** 6 - 8
Prerrequisitos: Módulo 02 (Álgebra Lineal para ML)

🧠 ¿Por Qué Cálculo para ML?

⚠️ CRÍTICO: Sin Chain Rule No Hay Deep Learning

```
El algoritmo de Backpropagation ES la Regla de la Cadena aplicada
a funciones compuestas de redes neuronales.

Si no entiendes:
  ∂L/∂w = ∂L/∂ŷ · ∂ŷ/∂z · ∂z/∂w

NO entenderás por qué funciona una red neuronal y
probablemente REPROBARÁS el curso de Deep Learning.
```

Conexión con el Pathway

Concepto	Uso en ML	Curso del Pathway
Derivada	Tasa de cambio, pendiente	Todos
Gradiente	Dirección de máximo ascenso	Supervised Learning
Gradient Descent	Optimización de parámetros	Supervised + Deep Learning
Chain Rule	Backpropagation	Deep Learning

📚 Contenido del Módulo

Semana 6: Derivadas y Derivadas Parciales

Semana 7: Gradiente y Gradient Descent

Semana 8: Chain Rule y Preparación para Backprop

💻 Parte 1: Derivadas

1.1 Concepto de Derivada

```python
import numpy as np
import matplotlib.pyplot as plt

"""
DERIVADA: Tasa de cambio instantánea de una función.

Definición formal:
    f'(x) = lim[h→0] (f(x+h) - f(x)) / h

Interpretación geométrica: pendiente de la recta tangente.

Notaciones equivalentes:
    f'(x) = df/dx = d/dx f(x) = Df(x)
"""

def numerical_derivative(f, x: float, h: float = 1e-7) -> float:
    """
    Calcula la derivada numérica usando diferencias finitas.

    Método: diferencia central (más preciso)
    f'(x) ≈ (f(x+h) - f(x-h)) / (2h)
    """
    return (f(x + h) - f(x - h)) / (2 * h)


# Ejemplo: f(x) = x²
def f(x):
    return x ** 2
```

```
# Derivada analítica: f'(x) = 2x
def f_prime_analytical(x):
    return 2 * x

# Comparar
x = 3.0
numerical = numerical_derivative(f, x)
analytical = f_prime_analytical(x)

print(f"f(x) = x² en x={x}")
print(f"Derivada numérica:  {numerical:.6f}")
print(f"Derivada analítica: {analytical:.6f}")
print(f"Error: {abs(numerical - analytical):.2e}")
```

1.2 Derivadas Comunes en ML

```
import numpy as np

"""
DERIVADAS QUE NECESITAS MEMORIZAR PARA ML:

1. Constante:     d/dx(c) = 0
2. Lineal:        d/dx(x) = 1
3. Potencia:      d/dx(xⁿ) = n·x^(n-1)
4. Exponencial:   d/dx(eˣ) = eˣ
5. Logaritmo:     d/dx(ln x) = 1/x
6. Suma:          d/dx(f+g) = f' + g'
7. Producto:      d/dx(f·g) = f'g + fg'
8. Cociente:      d/dx(f/g) = (f'g - fg')/g²
9. Cadena:        d/dx(f(g(x))) = f'(g(x))·g'(x)
"""

# Funciones de activación y sus derivadas

def sigmoid(x: np.ndarray) -> np.ndarray:
    """σ(x) = 1 / (1 + e^(-x))"""
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x: np.ndarray) -> np.ndarray:
    """
    d/dx σ(x) = σ(x) · (1 - σ(x))

    Derivación:
    σ(x) = (1 + e^(-x))^(-1)
    σ'(x) = -1·(1 + e^(-x))^(-2) · (-e^(-x))
          = e^(-x) / (1 + e^(-x))²
          = σ(x) · (1 - σ(x))
    """
    s = sigmoid(x)
    return s * (1 - s)


def relu(x: np.ndarray) -> np.ndarray:
    """ReLU(x) = max(0, x)"""
    return np.maximum(0, x)

def relu_derivative(x: np.ndarray) -> np.ndarray:
    """
    d/dx ReLU(x) = { 1 si x > 0
                   { 0 si x < 0
                   { indefinido si x = 0 (usamos 0)
    """
    return (x > 0).astype(float)


def tanh_derivative(x: np.ndarray) -> np.ndarray:
    """
    d/dx tanh(x) = 1 - tanh²(x)
    """
    return 1 - np.tanh(x) ** 2


# Verificar con derivada numérica
def verify_derivative(f, f_prime, x, name):
```

```
    numerical = (f(x + 1e-7) - f(x - 1e-7)) / (2e-7)
    analytical = f_prime(x)
    error = np.abs(numerical - analytical).max()
    print(f"{name}: error máximo = {error:.2e}")

x = np.array([-2, -1, 0.5, 1, 2])
verify_derivative(sigmoid, sigmoid_derivative, x, "Sigmoid")
verify_derivative(np.tanh, tanh_derivative, x, "Tanh")
```

1.3 Derivadas Parciales

```
import numpy as np

"""
DERIVADA PARCIAL: Derivada respecto a UNA variable,
manteniendo las otras constantes.

Para f(x, y):
    ∂f/∂x = derivada respecto a x, tratando y como constante
    ∂f/∂y = derivada respecto a y, tratando x como constante

Notación: ∂ (partial) en lugar de d
"""

def f(x: float, y: float) -> float:
    """f(x, y) = x² + 3xy + y²"""
    return x**2 + 3*x*y + y**2

# Derivadas parciales analíticas:
# ∂f/∂x = 2x + 3y
# ∂f/∂y = 3x + 2y

def df_dx(x: float, y: float) -> float:
    """∂f/∂x = 2x + 3y"""
    return 2*x + 3*y

def df_dy(x: float, y: float) -> float:
    """∂f/∂y = 3x + 2y"""
    return 3*x + 2*y


# Derivada parcial numérica
def partial_derivative(f, var_idx: int, point: list, h: float = 1e-7) -> float:
    """
    Calcula ∂f/∂xᵢ en un punto dado.

    Args:
        f: función
        var_idx: índice de la variable (0 para x, 1 para y, etc.)
        point: punto donde evaluar [x, y, ...]
        h: paso pequeño
    """
    point_plus = point.copy()
    point_minus = point.copy()
    point_plus[var_idx] += h
    point_minus[var_idx] -= h
    return (f(*point_plus) - f(*point_minus)) / (2 * h)


# Verificar
point = [2.0, 3.0]
print(f"Punto: x={point[0]}, y={point[1]}")
print(f"f(x,y) = {f(*point)}")
print(f"\n∂f/∂x:")
print(f"  Analítica: {df_dx(*point)}")
print(f"  Numérica:  {partial_derivative(f, 0, point):.6f}")
print(f"\n∂f/∂y:")
print(f"  Analítica: {df_dy(*point)}")
print(f"  Numérica:  {partial_derivative(f, 1, point):.6f}")
```

💻 Parte 2: Gradiente

2.1 Definición del Gradiente

```
import numpy as np

"""
GRADIENTE: Vector de todas las derivadas parciales.

Para f: Rⁿ → R (función de n variables que retorna un escalar):

∇f = [∂f/∂x₁, ∂f/∂x₂, ..., ∂f/∂xₙ]

Propiedades importantes:
1. El gradiente apunta en la dirección de MÁXIMO ASCENSO
2. La magnitud indica qué tan rápido aumenta f en esa dirección
3. -∇f apunta en la dirección de MÁXIMO DESCENSO (usado en optimización)
"""

def compute_gradient(f, point: np.ndarray, h: float = 1e-7) -> np.ndarray:
    """
    Calcula el gradiente de f en un punto usando diferencias finitas.

    Args:
        f: función f(x) donde x es un array
        point: punto donde calcular el gradiente
        h: paso para diferencias finitas

    Returns:
        gradiente como array
    """
    n = len(point)
    gradient = np.zeros(n)

    for i in range(n):
        point_plus = point.copy()
        point_minus = point.copy()
        point_plus[i] += h
        point_minus[i] -= h
        gradient[i] = (f(point_plus) - f(point_minus)) / (2 * h)

    return gradient


# Ejemplo: f(x, y) = x² + y²
def paraboloid(p: np.ndarray) -> float:
    """Paraboloide: f(x,y) = x² + y²"""
    return p[0]**2 + p[1]**2

# Gradiente analítico: ∇f = [2x, 2y]
def paraboloid_gradient_analytical(p: np.ndarray) -> np.ndarray:
    return np.array([2*p[0], 2*p[1]])


# Verificar
point = np.array([3.0, 4.0])
grad_numerical = compute_gradient(paraboloid, point)
grad_analytical = paraboloid_gradient_analytical(point)

print(f"Punto: {point}")
print(f"f(punto) = {paraboloid(point)}")
print(f"Gradiente numérico:  {grad_numerical}")
print(f"Gradiente analítico: {grad_analytical}")
```

2.2 Visualización del Gradiente

```
import numpy as np
import matplotlib.pyplot as plt

def visualize_gradient():
    """Visualiza el gradiente como campo vectorial."""

    # Crear grid
    x = np.linspace(-3, 3, 15)
```

```
    y = np.linspace(-3, 3, 15)
    X, Y = np.meshgrid(x, y)

    # Función: f(x,y) = x² + y²
    Z = X**2 + Y**2

    # Gradiente: ∇f = [2x, 2y]
    U = 2 * X  # ∂f/∂x
    V = 2 * Y  # ∂f/∂y

    # Normalizar para visualización
    magnitude = np.sqrt(U**2 + V**2)
    U_norm = U / (magnitude + 0.1)
    V_norm = V / (magnitude + 0.1)

    plt.figure(figsize=(10, 8))

    # Contornos de nivel
    plt.contour(X, Y, Z, levels=20, cmap='viridis', alpha=0.5)
    plt.colorbar(label='f(x,y) = x² + y²')

    # Flechas del gradiente
    plt.quiver(X, Y, U_norm, V_norm, magnitude, cmap='Reds', alpha=0.8)

    # Punto mínimo
    plt.plot(0, 0, 'g*', markersize=15, label='Mínimo global')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Gradiente de f(x,y) = x² + y²\nLas flechas apuntan hacia ARRIBA (máximo ascenso)')
    plt.legend()
    plt.axis('equal')
    plt.grid(True, alpha=0.3)
    plt.show()

# visualize_gradient()  # Descomentar para ejecutar
```

💻 Parte 3: Gradient Descent

3.1 Algoritmo Básico

```
import numpy as np
from typing import Callable, List, Tuple

"""
GRADIENT DESCENT: Algoritmo de optimización iterativo.

Idea: Para minimizar f(x), moverse en dirección opuesta al gradiente.

Algoritmo:
    1. Inicializar x₀
    2. Repetir hasta convergencia:
       x_{t+1} = x_t - α · ∇f(x_t)

Donde α (alpha) es el "learning rate" (tasa de aprendizaje).
"""

def gradient_descent(
    f: Callable,
    grad_f: Callable,
    x0: np.ndarray,
    learning_rate: float = 0.1,
    max_iterations: int = 100,
    tolerance: float = 1e-6
) -> Tuple[np.ndarray, List[np.ndarray], List[float]]:
    """
    Gradient Descent para minimizar f.

    Args:
        f: función a minimizar
        grad_f: gradiente de f
        x0: punto inicial
        learning_rate: tasa de aprendizaje (α)
        max_iterations: máximo de iteraciones
```

```
        tolerance: criterio de parada (norma del gradiente)

    Returns:
        x_final: solución encontrada
        history_x: trayectoria de x
        history_f: valores de f en cada paso
    """
    x = x0.copy()
    history_x = [x.copy()]
    history_f = [f(x)]

    for i in range(max_iterations):
        # Calcular gradiente
        grad = grad_f(x)

        # Verificar convergencia
        if np.linalg.norm(grad) < tolerance:
            print(f"Convergió en iteración {i}")
            break

        # Actualizar x
        x = x - learning_rate * grad

        # Guardar historia
        history_x.append(x.copy())
        history_f.append(f(x))

    return x, history_x, history_f


# Ejemplo: Minimizar f(x,y) = x² + y²
def f(p: np.ndarray) -> float:
    return p[0]**2 + p[1]**2

def grad_f(p: np.ndarray) -> np.ndarray:
    return np.array([2*p[0], 2*p[1]])

# Ejecutar
x0 = np.array([4.0, 3.0])
x_final, history_x, history_f = gradient_descent(f, grad_f, x0, learning_rate=0.1)

print(f"\nPunto inicial: {x0}")
print(f"Mínimo encontrado: {x_final}")
print(f"f(mínimo) = {f(x_final):.6f}")
print(f"Iteraciones: {len(history_f)}")
```

3.2 Efecto del Learning Rate

```
import numpy as np
import matplotlib.pyplot as plt

def compare_learning_rates():
    """Compara diferentes learning rates."""

    def f(p):
        return p[0]**2 + p[1]**2

    def grad_f(p):
        return np.array([2*p[0], 2*p[1]])

    x0 = np.array([4.0, 3.0])

    learning_rates = [0.01, 0.1, 0.5, 0.9]

    plt.figure(figsize=(12, 4))

    for i, lr in enumerate(learning_rates):
        x_final, history_x, history_f = gradient_descent(
            f, grad_f, x0, learning_rate=lr, max_iterations=50
        )

        plt.subplot(1, 4, i+1)
        plt.plot(history_f, 'b-o', markersize=3)
        plt.xlabel('Iteración')
```

```
        plt.ylabel('f(x)')
        plt.title(f'α = {lr}')
        plt.yscale('log')
        plt.grid(True)

    plt.tight_layout()
    plt.suptitle('Efecto del Learning Rate en Gradient Descent', y=1.02)
    plt.show()

    """
    Observaciones:
    - α muy pequeño (0.01): Convergencia muy lenta
    - α óptimo (0.1-0.5): Convergencia rápida y estable
    - α muy grande (0.9): Oscilaciones, puede diverger
    - α > 1: Generalmente diverge para este problema
    """

# compare_learning_rates()  # Descomentar para ejecutar
```

3.3 Funciones de Pérdida en ML

```
import numpy as np

"""
FUNCIONES DE PÉRDIDA COMUNES Y SUS GRADIENTES

En ML, minimizamos una "función de pérdida" (loss function)
que mide qué tan mal están nuestras predicciones.
"""

# 1. MSE (Mean Squared Error) - Regresión
def mse_loss(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    MSE = (1/n) Σ (y_true - y_pred)²
    """
    return np.mean((y_true - y_pred) ** 2)

def mse_gradient(y_true: np.ndarray, y_pred: np.ndarray) -> np.ndarray:
    """
    ∂MSE/∂y_pred = (2/n) Σ (y_pred - y_true)
                 = (2/n) (y_pred - y_true)
    """
    n = len(y_true)
    return (2 / n) * (y_pred - y_true)


# 2. Binary Cross-Entropy - Clasificación binaria
def binary_cross_entropy(y_true: np.ndarray, y_pred: np.ndarray, eps: float = 1e-15) -> float:
    """
    BCE = -(1/n) Σ [y·log(ŷ) + (1-y)·log(1-ŷ)]
    """
    y_pred = np.clip(y_pred, eps, 1 - eps)  # Evitar log(0)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

def binary_cross_entropy_gradient(y_true: np.ndarray, y_pred: np.ndarray, eps: float = 1e-15) -> np.ndarray:
    """
    ∂BCE/∂y_pred = (1/n) · (y_pred - y_true) / (y_pred · (1 - y_pred))

    Simplificación cuando y_pred = σ(z):
    ∂BCE/∂z = (1/n) · (y_pred - y_true)
    """
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return (y_pred - y_true) / (y_pred * (1 - y_pred)) / len(y_true)


# Demo
np.random.seed(42)
y_true = np.array([0, 0, 1, 1])
y_pred = np.array([0.1, 0.2, 0.8, 0.9])

print("MSE Loss:", mse_loss(y_true, y_pred))
print("BCE Loss:", binary_cross_entropy(y_true, y_pred))
```

💻 Parte 4: Regla de la Cadena (Chain Rule)

4.1 Chain Rule en 1D

```
import numpy as np

"""
REGLA DE LA CADENA (Chain Rule)

Si y = f(g(x)), entonces:
    dy/dx = df/dg · dg/dx

O en notación de composición:
    (f ∘ g)'(x) = f'(g(x)) · g'(x)

Esto es FUNDAMENTAL para Backpropagation.
"""

# Ejemplo: y = (x² + 1)³
#
# Sea g(x) = x² + 1  y  f(u) = u³
# Entonces y = f(g(x))
#
# dy/dx = f'(g(x)) · g'(x)
#       = 3(x² + 1)² · 2x
#       = 6x(x² + 1)²

def g(x):
    return x**2 + 1

def f(u):
    return u**3

def y(x):
    return f(g(x))

def dy_dx_analytical(x):
    """Derivada usando chain rule."""
    return 6 * x * (x**2 + 1)**2

def dy_dx_numerical(x, h=1e-7):
    """Derivada numérica."""
    return (y(x + h) - y(x - h)) / (2 * h)


# Verificar
x = 2.0
print(f"y({x}) = {y(x)}")
print(f"dy/dx analítica:  {dy_dx_analytical(x)}")
print(f"dy/dx numérica:   {dy_dx_numerical(x):.6f}")
```

4.2 Chain Rule para Funciones Compuestas (Backprop Preview)

```
import numpy as np

"""
CHAIN RULE PARA REDES NEURONALES

Una capa de red neuronal:
    z = Wx + b       (transformación lineal)
    a = σ(z)         (activación)

Si L es la pérdida, necesitamos:
    ∂L/∂W, ∂L/∂b     (para actualizar los pesos)

Usando Chain Rule:
    ∂L/∂W = ∂L/∂a · ∂a/∂z · ∂z/∂W
    ∂L/∂b = ∂L/∂a · ∂a/∂z · ∂z/∂b
"""

def simple_forward_backward():
    """
    Ejemplo simplificado de forward y backward pass.
```

```
    Red: x → [z = wx + b] → [a = sigmoid(z)] → [L = (a - y)²]
    """
    # Datos
    x = 2.0          # Input
    y_true = 1.0     # Target

    # Parámetros
    w = 0.5
    b = 0.1

    # ========== FORWARD PASS ==========
    z = w * x + b                    # z = wx + b
    a = 1 / (1 + np.exp(-z))         # a = sigmoid(z)
    L = (a - y_true) ** 2            # L = MSE

    print("=== FORWARD PASS ===")
    print(f"z = w*x + b = {w}*{x} + {b} = {z}")
    print(f"a = sigmoid(z) = {a:.4f}")
    print(f"L = (a - y)² = ({a:.4f} - {y_true})² = {L:.4f}")

    # ========== BACKWARD PASS (Chain Rule) ==========
    # Objetivo: calcular ∂L/∂w y ∂L/∂b

    # Paso 1: ∂L/∂a
    dL_da = 2 * (a - y_true)

    # Paso 2: ∂a/∂z = sigmoid'(z) = a(1-a)
    da_dz = a * (1 - a)

    # Paso 3: ∂z/∂w = x,  ∂z/∂b = 1
    dz_dw = x
    dz_db = 1

    # Aplicar Chain Rule
    dL_dz = dL_da * da_dz           # ∂L/∂z = ∂L/∂a · ∂a/∂z
    dL_dw = dL_dz * dz_dw           # ∂L/∂w = ∂L/∂z · ∂z/∂w
    dL_db = dL_dz * dz_db           # ∂L/∂b = ∂L/∂z · ∂z/∂b

    print("\n=== BACKWARD PASS (Chain Rule) ===")
    print(f"∂L/∂a = 2(a - y) = {dL_da:.4f}")
    print(f"∂a/∂z = a(1-a) = {da_dz:.4f}")
    print(f"∂z/∂w = x = {dz_dw}")
    print(f"∂z/∂b = 1")
    print(f"\n∂L/∂w = ∂L/∂a · ∂a/∂z · ∂z/∂w = {dL_dw:.4f}")
    print(f"∂L/∂b = ∂L/∂a · ∂a/∂z · ∂z/∂b = {dL_db:.4f}")

    # ========== VERIFICACIÓN NUMÉRICA ==========
    h = 1e-7

    # ∂L/∂w numérica
    z_plus = (w + h) * x + b
    a_plus = 1 / (1 + np.exp(-z_plus))
    L_plus = (a_plus - y_true) ** 2

    z_minus = (w - h) * x + b
    a_minus = 1 / (1 + np.exp(-z_minus))
    L_minus = (a_minus - y_true) ** 2

    dL_dw_numerical = (L_plus - L_minus) / (2 * h)

    print(f"\n=== VERIFICACIÓN ===")
    print(f"∂L/∂w analítica: {dL_dw:.6f}")
    print(f"∂L/∂w numérica:  {dL_dw_numerical:.6f}")
    print(f"Error: {abs(dL_dw - dL_dw_numerical):.2e}")

    return dL_dw, dL_db

simple_forward_backward()
```

4.3 Backpropagation en una Red de 2 Capas

```
import numpy as np

"""
```

```
RED NEURONAL DE 2 CAPAS

Arquitectura:
    x (input)
    → z₁ = W₁x + b₁
    → a₁ = sigmoid(z₁)
    → z₂ = W₂a₁ + b₂
    → a₂ = sigmoid(z₂)
    → L = MSE(a₂, y)

Backpropagation usa Chain Rule repetidamente:
    ∂L/∂W₂ = ∂L/∂a₂ · ∂a₂/∂z₂ · ∂z₂/∂W₂
    ∂L/∂W₁ = ∂L/∂a₂ · ∂a₂/∂z₂ · ∂z₂/∂a₁ · ∂a₁/∂z₁ · ∂z₁/∂W₁
"""

class SimpleNeuralNet:
    """Red neuronal de 2 capas para demostrar backprop."""

    def __init__(self, input_size: int, hidden_size: int, output_size: int):
        # Inicializar pesos (Xavier initialization)
        self.W1 = np.random.randn(hidden_size, input_size) * np.sqrt(2 / input_size)
        self.b1 = np.zeros(hidden_size)
        self.W2 = np.random.randn(output_size, hidden_size) * np.sqrt(2 / hidden_size)
        self.b2 = np.zeros(output_size)

        # Cache para backprop
        self.cache = {}

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

    def sigmoid_derivative(self, a):
        return a * (1 - a)

    def forward(self, x: np.ndarray) -> np.ndarray:
        """Forward pass guardando valores intermedios."""
        # Capa 1
        z1 = self.W1 @ x + self.b1
        a1 = self.sigmoid(z1)

        # Capa 2
        z2 = self.W2 @ a1 + self.b2
        a2 = self.sigmoid(z2)

        # Guardar para backprop
        self.cache = {'x': x, 'z1': z1, 'a1': a1, 'z2': z2, 'a2': a2}

        return a2

    def backward(self, y_true: np.ndarray) -> dict:
        """
        Backward pass usando Chain Rule.

        Returns:
            Gradientes de todos los parámetros
        """
        x = self.cache['x']
        a1 = self.cache['a1']
        a2 = self.cache['a2']

        # ∂L/∂a₂ (MSE)
        dL_da2 = 2 * (a2 - y_true)

        # ∂a₂/∂z₂
        da2_dz2 = self.sigmoid_derivative(a2)

        # ∂L/∂z₂ = ∂L/∂a₂ · ∂a₂/∂z₂
        dL_dz2 = dL_da2 * da2_dz2

        # Gradientes de capa 2
        # ∂z₂/∂W₂ = a₁, ∂z₂/∂b₂ = 1
        dL_dW2 = np.outer(dL_dz2, a1)
        dL_db2 = dL_dz2

        # Propagar hacia atrás a capa 1
```

```
        # ∂z₂/∂a₁ = W₂
        dL_da1 = self.W2.T @ dL_dz2

        # ∂a₁/∂z₁
        da1_dz1 = self.sigmoid_derivative(a1)

        # ∂L/∂z₁
        dL_dz1 = dL_da1 * da1_dz1

        # Gradientes de capa 1
        dL_dW1 = np.outer(dL_dz1, x)
        dL_db1 = dL_dz1

        return {
            'dW1': dL_dW1, 'db1': dL_db1,
            'dW2': dL_dW2, 'db2': dL_db2
        }

    def update(self, gradients: dict, learning_rate: float):
        """Actualiza parámetros usando gradient descent."""
        self.W1 -= learning_rate * gradients['dW1']
        self.b1 -= learning_rate * gradients['db1']
        self.W2 -= learning_rate * gradients['dW2']
        self.b2 -= learning_rate * gradients['db2']


# Demo: XOR problem
def demo_xor():
    """Entrena la red para resolver XOR."""
    # XOR data
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]).T  # 2x4
    y = np.array([[0], [1], [1], [0]]).T              # 1x4

    # Crear red
    net = SimpleNeuralNet(input_size=2, hidden_size=4, output_size=1)

    # Entrenar
    losses = []
    for epoch in range(10000):
        total_loss = 0
        for i in range(4):
            # Forward
            output = net.forward(X[:, i])
            loss = (output - y[:, i]) ** 2
            total_loss += loss[0]

            # Backward
            gradients = net.backward(y[:, i])

            # Update
            net.update(gradients, learning_rate=0.5)

        losses.append(total_loss / 4)

        if epoch % 2000 == 0:
            print(f"Epoch {epoch}: Loss = {losses[-1]:.4f}")

    # Test
    print("\n=== Resultados XOR ===")
    for i in range(4):
        pred = net.forward(X[:, i])
        print(f"Input: {X[:, i]} → Pred: {pred[0]:.3f} (Target: {y[0, i]})")

demo_xor()
```

📦 Entregable del Módulo

Script: `gradient_descent_demo.py`

```
"""
Gradient Descent Demo - Visualización de Optimización

Este script implementa Gradient Descent desde cero y visualiza
la trayectoria de optimización en diferentes funciones.
```

```
Autor: [Tu nombre]
Módulo: 03 - Cálculo Multivariante
"""

import numpy as np
import matplotlib.pyplot as plt
from typing import Callable, Tuple, List


def gradient_descent(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
    learning_rate: float = 0.1,
    max_iterations: int = 100,
    tolerance: float = 1e-8
) -> Tuple[np.ndarray, List[np.ndarray], List[float]]:
    """
    Implementación de Gradient Descent.

    Args:
        f: función objetivo
        grad_f: gradiente de f
        x0: punto inicial
        learning_rate: α
        max_iterations: máximo de iteraciones
        tolerance: criterio de convergencia

    Returns:
        x_final, history_x, history_f
    """
    x = x0.copy().astype(float)
    history_x = [x.copy()]
    history_f = [f(x)]

    for i in range(max_iterations):
        grad = grad_f(x)

        if np.linalg.norm(grad) < tolerance:
            break

        x = x - learning_rate * grad
        history_x.append(x.copy())
        history_f.append(f(x))

    return x, history_x, history_f


def visualize_optimization(
    f: Callable,
    grad_f: Callable,
    x0: np.ndarray,
    learning_rate: float,
    title: str,
    xlim: Tuple[float, float] = (-5, 5),
    ylim: Tuple[float, float] = (-5, 5)
):
    """Visualiza la trayectoria de optimización."""

    x_final, history_x, history_f = gradient_descent(
        f, grad_f, x0, learning_rate, max_iterations=50
    )

    # Crear grid para contornos
    x = np.linspace(xlim[0], xlim[1], 100)
    y = np.linspace(ylim[0], ylim[1], 100)
    X, Y = np.meshgrid(x, y)
    Z = np.array([[f(np.array([xi, yi])) for xi, yi in zip(row_x, row_y)]
                  for row_x, row_y in zip(X, Y)])

    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    # Plot 1: Contornos y trayectoria
    ax1 = axes[0]
```

```
    contour = ax1.contour(X, Y, Z, levels=30, cmap='viridis')
    ax1.clabel(contour, inline=True, fontsize=8)

    # Trayectoria
    history_x = np.array(history_x)
    ax1.plot(history_x[:, 0], history_x[:, 1], 'r.-', markersize=8, linewidth=1.5)
    ax1.plot(history_x[0, 0], history_x[0, 1], 'go', markersize=12, label='Inicio')
    ax1.plot(history_x[-1, 0], history_x[-1, 1], 'r*', markersize=15, label='Final')

    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.set_title(f'{title}\nα = {learning_rate}')
    ax1.legend()
    ax1.set_xlim(xlim)
    ax1.set_ylim(ylim)

    # Plot 2: Convergencia
    ax2 = axes[1]
    ax2.semilogy(history_f, 'b-o', markersize=4)
    ax2.set_xlabel('Iteración')
    ax2.set_ylabel('f(x) (escala log)')
    ax2.set_title('Convergencia')
    ax2.grid(True)

    plt.tight_layout()
    plt.savefig(f'gd_{title.lower().replace(" ", "_")}.png', dpi=150)
    plt.show()

    print(f"\n{title}")
    print(f"  Punto inicial: {x0}")
    print(f"  Mínimo encontrado: {x_final}")
    print(f"  f(mínimo): {f(x_final):.6f}")
    print(f"  Iteraciones: {len(history_f)}")


def main():
    """Ejecutar demos."""

    # === Función 1: Paraboloide ===
    def paraboloid(p):
        return p[0]**2 + p[1]**2

    def grad_paraboloid(p):
        return np.array([2*p[0], 2*p[1]])

    visualize_optimization(
        paraboloid, grad_paraboloid,
        x0=np.array([4.0, 3.0]),
        learning_rate=0.1,
        title="Paraboloide f(x,y) = x² + y²"
    )

    # === Función 2: Rosenbrock (más difícil) ===
    def rosenbrock(p):
        return (1 - p[0])**2 + 100*(p[1] - p[0]**2)**2

    def grad_rosenbrock(p):
        dx = -2*(1 - p[0]) - 400*p[0]*(p[1] - p[0]**2)
        dy = 200*(p[1] - p[0]**2)
        return np.array([dx, dy])

    visualize_optimization(
        rosenbrock, grad_rosenbrock,
        x0=np.array([-1.0, 1.0]),
        learning_rate=0.001,
        title="Rosenbrock f(x,y) = (1-x)² + 100(y-x²)²",
        xlim=(-2, 2),
        ylim=(-1, 3)
    )

    # === Función 3: Cuadrática elíptica ===
    def elliptic(p):
        return p[0]**2 + 10*p[1]**2

    def grad_elliptic(p):
```

```
        return np.array([2*p[0], 20*p[1]])

    visualize_optimization(
        elliptic, grad_elliptic,
        x0=np.array([4.0, 2.0]),
        learning_rate=0.05,
        title="Elíptica f(x,y) = x² + 10y²"
    )


if __name__ == "__main__":
    main()
```

🔬 Gradient Checking: Validación Matemática (v3.3)

> ⚠️ **CRÍTICO:** El mayor riesgo en ML es implementar backpropagation incorrectamente. El código puede correr, el loss puede bajar, pero el gradiente estar mal. **Esta técnica es estándar en CS 2 3 1 n de Stanford.**

Concepto: Derivada Numérica vs Analítica

```
GRADIENT CHECKING

Tu gradiente analítico (backprop):
    ∂L/∂w = [valor calculado con Chain Rule]

Gradiente numérico (aproximación):
    ∂L/∂w ≈ [L(w + ε) - L(w - ε)] / (2ε)

Si |analítico - numérico| > 10⁻⁷ → TU IMPLEMENTACIÓN TIENE UN BUG
```

Script: `grad_check.py` (Entregable Obligatorio v3.3)

```
"""
Gradient Checking - Validación de Derivadas
Técnica estándar de CS231n Stanford para debugging de backprop.

Autor: [Tu nombre]
Módulo: 03 - Cálculo Multivariante
"""
import numpy as np
from typing import Callable, Dict, Tuple


def numerical_gradient(
    f: Callable[[np.ndarray], float],
    x: np.ndarray,
    epsilon: float = 1e-5
) -> np.ndarray:
    """
    Calcula el gradiente numérico usando diferencias centrales.

    Args:
        f: Función escalar f(x) -> float
        x: Punto donde calcular el gradiente
        epsilon: Tamaño del paso (default: 1e-5)

    Returns:
        Gradiente numérico aproximado
    """
    grad = np.zeros_like(x)

    # Iterar sobre cada dimensión
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        old_value = x[idx]

        # f(x + epsilon)
        x[idx] = old_value + epsilon
        fx_plus = f(x)

        # f(x - epsilon)
```

```
        x[idx] = old_value - epsilon
        fx_minus = f(x)

        # Diferencias centrales: (f(x+ε) - f(x-ε)) / 2ε
        grad[idx] = (fx_plus - fx_minus) / (2 * epsilon)

        # Restaurar valor original
        x[idx] = old_value
        it.iternext()

    return grad


def gradient_check(
    analytic_grad: np.ndarray,
    numerical_grad: np.ndarray,
    threshold: float = 1e-7
) -> Tuple[bool, float]:
    """
    Compara gradiente analítico vs numérico.

    Args:
        analytic_grad: Gradiente calculado con backprop
        numerical_grad: Gradiente calculado numéricamente
        threshold: Umbral de error aceptable

    Returns:
        (passed, relative_error)
    """
    # Error relativo: ||a - n|| / (||a|| + ||n||)
    diff = np.linalg.norm(analytic_grad - numerical_grad)
    norm_sum = np.linalg.norm(analytic_grad) + np.linalg.norm(numerical_grad)

    if norm_sum == 0:
        relative_error = 0.0
    else:
        relative_error = diff / norm_sum

    passed = relative_error < threshold
    return passed, relative_error


# ============================================================
# EJEMPLO: Validar gradiente de MSE Loss
# ============================================================

def mse_loss(y_pred: np.ndarray, y_true: np.ndarray) -> float:
    """Mean Squared Error."""
    return float(np.mean((y_pred - y_true) ** 2))


def mse_gradient_analytic(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray:
    """Gradiente analítico de MSE respecto a y_pred."""
    n = len(y_true)
    return 2 * (y_pred - y_true) / n


def test_mse_gradient():
    """Test: Validar gradiente de MSE."""
    print("=" * 60)
    print("GRADIENT CHECK: MSE Loss")
    print("=" * 60)

    np.random.seed(42)
    y_pred = np.random.randn(10)
    y_true = np.random.randn(10)

    # Gradiente analítico
    grad_analytic = mse_gradient_analytic(y_pred, y_true)

    # Gradiente numérico
    def loss_fn(pred):
        return mse_loss(pred, y_true)

    grad_numerical = numerical_gradient(loss_fn, y_pred.copy())
```

```
    # Comparar
    passed, error = gradient_check(grad_analytic, grad_numerical)

    print(f"Gradiente Analítico: {grad_analytic[:3]}...")
    print(f"Gradiente Numérico:  {grad_numerical[:3]}...")
    print(f"Error Relativo: {error:.2e}")
    print(f"Resultado: {'✓ PASSED' if passed else '✗ FAILED'}")

    return passed


# ============================================================
# EJEMPLO: Validar gradiente de Sigmoid
# ============================================================

def sigmoid(z: np.ndarray) -> np.ndarray:
    """Sigmoid activation."""
    return 1 / (1 + np.exp(-z))


def sigmoid_derivative_analytic(z: np.ndarray) -> np.ndarray:
    """Derivada analítica: σ'(z) = σ(z)(1 - σ(z))"""
    s = sigmoid(z)
    return s * (1 - s)


def test_sigmoid_gradient():
    """Test: Validar derivada de sigmoid."""
    print("\n" + "=" * 60)
    print("GRADIENT CHECK: Sigmoid Derivative")
    print("=" * 60)

    np.random.seed(42)
    z = np.random.randn(5)

    # Derivada analítica
    grad_analytic = sigmoid_derivative_analytic(z)

    # Derivada numérica (para cada elemento)
    def sigmoid_element(z_arr):
        return float(np.sum(sigmoid(z_arr)))  # Suma para tener escalar

    grad_numerical = numerical_gradient(sigmoid_element, z.copy())

    # Comparar
    passed, error = gradient_check(grad_analytic, grad_numerical)

    print(f"Derivada Analítica: {grad_analytic}")
    print(f"Derivada Numérica:  {grad_numerical}")
    print(f"Error Relativo: {error:.2e}")
    print(f"Resultado: {'✓ PASSED' if passed else '✗ FAILED'}")

    return passed


# ============================================================
# EJEMPLO: Validar gradiente de una capa lineal
# ============================================================

def test_linear_layer_gradient():
    """Test: Validar gradiente de capa lineal y = Wx + b."""
    print("\n" + "=" * 60)
    print("GRADIENT CHECK: Linear Layer (y = Wx + b)")
    print("=" * 60)

    np.random.seed(42)

    # Dimensiones
    n_in, n_out = 4, 3

    # Parámetros
    W = np.random.randn(n_out, n_in)
    b = np.random.randn(n_out)
    x = np.random.randn(n_in)
```

```
    y_true = np.random.randn(n_out)

    # Forward + Loss
    def forward_and_loss(W_flat):
        W_reshaped = W_flat.reshape(n_out, n_in)
        y_pred = W_reshaped @ x + b
        return mse_loss(y_pred, y_true)

    # Gradiente analítico de W
    y_pred = W @ x + b
    dL_dy = 2 * (y_pred - y_true) / n_out  # Gradiente de MSE
    dL_dW_analytic = np.outer(dL_dy, x)    # ∂L/∂W = ∂L/∂y · x^T

    # Gradiente numérico de W
    dL_dW_numerical = numerical_gradient(forward_and_loss, W.flatten().copy())
    dL_dW_numerical = dL_dW_numerical.reshape(n_out, n_in)

    # Comparar
    passed, error = gradient_check(
        dL_dW_analytic.flatten(),
        dL_dW_numerical.flatten()
    )

    print(f"Error Relativo: {error:.2e}")
    print(f"Resultado: {'✓ PASSED' if passed else '✗ FAILED'}")

    return passed


def main():
    """Ejecutar todos los gradient checks."""
    print("\n" + "=" * 60)
    print("       GRADIENT CHECKING SUITE")
    print("       Validación Matemática v3.3")
    print("=" * 60)

    results = []
    results.append(("MSE Loss", test_mse_gradient()))
    results.append(("Sigmoid", test_sigmoid_gradient()))
    results.append(("Linear Layer", test_linear_layer_gradient()))

    print("\n" + "=" * 60)
    print("RESUMEN")
    print("=" * 60)

    all_passed = True
    for name, passed in results:
        status = "✓ PASSED" if passed else "✗ FAILED"
        print(f"  {name}: {status}")
        all_passed = all_passed and passed

    print("-" * 60)
    if all_passed:
        print("✓ TODOS LOS GRADIENT CHECKS PASARON")
        print("  Tu implementación de derivadas es correcta.")
    else:
        print("✗ ALGUNOS GRADIENT CHECKS FALLARON")
        print("  Revisa tu implementación de backprop.")

    return all_passed


if __name__ == "__main__":
    main()
```

Cómo Usar Gradient Checking

```
# En tu código de backprop:

# 1. Calcula el gradiente analítico (tu implementación)
grad_analytic = my_backward_pass(...)

# 2. Calcula el gradiente numérico
def loss_wrapper(params):
```

```
    return forward_pass(params, ...)

grad_numerical = numerical_gradient(loss_wrapper, params)

# 3. Compara
passed, error = gradient_check(grad_analytic, grad_numerical)
if not passed:
    raise ValueError(f"Gradient check failed! Error: {error:.2e}")
```

Reglas del Gradient Checking

Error Relativo	Interpretación
$< 10^{-7}$	✓ Excelente - tu gradiente es correcto
10^{-7} a 10^{-5}	⚠️ Sospechoso - revisa tu código
$> 10^{-5}$	✗ Bug - tu backprop está mal

> ⚠️ **Nota:** Desactiva gradient checking durante el entrenamiento real (es lento). Solo úsalo para validar tu implementación.

✅ Checklist de Finalización (v3.3)

Conocimiento

- [ ] Puedo calcular derivadas de funciones comunes (polinomios, exp, log)
- [ ] Entiendo derivadas parciales y puedo calcularlas
- [ ] Puedo calcular el gradiente de una función multivariable
- [ ] Implementé Gradient Descent desde cero
- [ ] Entiendo el efecto del learning rate
- [ ] Puedo aplicar la Chain Rule a funciones compuestas
- [ ] Entiendo cómo la Chain Rule se aplica en Backpropagation
- [ ] Puedo derivar ∂L/∂w para una neurona simple

Entregables v3.3

- [ ] `gradient_descent_demo.py` funcional
- [ ] **`grad_check.py` implementado y todos los tests pasan**
- [ ] Validé mis derivadas de sigmoid, MSE y capa lineal

Metodología Feynman

- [ ] Puedo explicar Chain Rule en 5 líneas sin jerga
- [ ] Puedo explicar por qué gradient checking funciona

🔗 Navegación

Anterior	Índice	Siguiente
02_ALGEBRA_LINEAL_ML	00_INDICE	04_PROBABILIDAD_ML

MÓDULO 04 - PROBABILIDAD PARA ML

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 04: Probabilidad Esencial para Machine Learning

Semana 8 **| Prerequisito para entender Loss Functions y GMM**
Filosofía: Solo la probabilidad que necesitas para la Línea 1

🎯 Objetivo del Módulo

Dominar los **conceptos mínimos de probabilidad** necesarios para:

1. Entender **Logistic Regression** como modelo probabilístico
2. Comprender **Cross-Entropy Loss** y por qué funciona
3. Prepararte para **Gaussian Mixture Models (GMM)** en Unsupervised
4. Entender **Softmax** como distribución de probabilidad

> ⚠️ **Nota:** Este NO es el curso completo de Probabilidad (Línea 2). Es solo lo esencial para ML.

📚 Contenido

Día 1-2: Fundamentos de Probabilidad

1.1 Probabilidad Básica

```
P(A) = casos favorables / casos totales

Propiedades:
- 0 ≤ P(A) ≤ 1
- P(Ω) = 1 (espacio muestral)
- P(∅) = 0 (evento imposible)
```

1.2 Probabilidad Condicional

```
P(A|B) = P(A ∩ B) / P(B)

"Probabilidad de A dado que B ocurrió"
```

Ejemplo en ML:
- P(spam | contiene "gratis") = ¿Qué tan probable es spam si el email dice "gratis"?

1.3 Independencia

```
A y B son independientes si:
P(A ∩ B) = P(A) · P(B)

Equivalente a:
P(A|B) = P(A)
```

Día 3-4: Teorema de Bayes (Crítico para ML)

2.1 La Fórmula

```
            P(B|A) · P(A)
P(A|B) = ─────────────────
                P(B)

Donde:
- P(A|B) = Posterior (lo que queremos calcular)
- P(B|A) = Likelihood (verosimilitud)
- P(A)   = Prior (conocimiento previo)
- P(B)   = Evidence (normalizador)
```

2.2 Interpretación para ML

```
              P(datos|clase) · P(clase)
P(clase|datos) = ─────────────────────────
                        P(datos)

Ejemplo: Clasificación de spam
- P(spam|palabras) = P(palabras|spam) · P(spam) / P(palabras)
```

2.3 Implementación en Python

```
import numpy as np

def bayes_classifier(x: np.ndarray,
                     likelihood_spam: float,
                     likelihood_ham: float,
                     prior_spam: float = 0.3) -> str:
    """
    Clasificador Bayesiano simple.

    Args:
        x: Características del email (simplificado)
        likelihood_spam: P(x|spam)
        likelihood_ham: P(x|ham)
        prior_spam: P(spam) - conocimiento previo

    Returns:
        'spam' o 'ham'
    """
    prior_ham = 1 - prior_spam

    # Posterior (sin normalizar, solo comparamos)
    posterior_spam = likelihood_spam * prior_spam
    posterior_ham = likelihood_ham * prior_ham

    return 'spam' if posterior_spam > posterior_ham else 'ham'


# Ejemplo: Email con palabra "gratis"
# P("gratis"|spam) = 0.8, P("gratis"|ham) = 0.1
result = bayes_classifier(
    x=None,  # simplificado
    likelihood_spam=0.8,
    likelihood_ham=0.1,
    prior_spam=0.3
)
print(f"Clasificación: {result}")  # spam
```

2.4 Naive Bayes (Conexión con Supervised Learning)

```
def naive_bayes_predict(X: np.ndarray,
                        class_priors: np.ndarray,
                        feature_probs: dict) -> np.ndarray:
    """
    Naive Bayes asume independencia entre features:
    P(x1, x2, ..., xn | clase) = P(x1|clase) · P(x2|clase) · ... · P(xn|clase)

    Esta "ingenuidad" simplifica mucho el cálculo.
    """
    n_samples = X.shape[0]
    n_classes = len(class_priors)

    log_posteriors = np.zeros((n_samples, n_classes))

    for c in range(n_classes):
        # Log para evitar underflow con muchas features
        log_prior = np.log(class_priors[c])
        log_likelihood = np.sum(np.log(feature_probs[c][X]), axis=1)
        log_posteriors[:, c] = log_prior + log_likelihood

    return np.argmax(log_posteriors, axis=1)
```

Día 5: Distribución Gaussiana (Normal)

3.1 La Distribución Más Importante en ML

$$f(x) = \frac{1}{\sigma \cdot \sqrt{(2\pi)}} \cdot \exp(-\frac{(x - \mu)^2}{2\sigma^2})$$

Parámetros:
- μ (mu): Media (centro de la campana)

```
- σ (sigma): Desviación estándar (ancho)
- σ² (sigma²): Varianza
```

3.2 Por Qué es Importante

1. **Muchos fenómenos naturales** siguen esta distribución
2. **Teorema del Límite Central:** promedios de cualquier distribución → Normal
3. **GMM usa Gaussianas** para modelar clusters
4. **Inicialización de pesos** en redes neuronales

3.3 Implementación

```
import numpy as np

def gaussian_pdf(x: np.ndarray, mu: float, sigma: float) -> np.ndarray:
    """
    Probability Density Function de la Gaussiana.

    Args:
        x: Puntos donde evaluar
        mu: Media
        sigma: Desviación estándar

    Returns:
        Densidad de probabilidad en cada punto
    """
    coefficient = 1 / (sigma * np.sqrt(2 * np.pi))
    exponent = -((x - mu) ** 2) / (2 * sigma ** 2)
    return coefficient * np.exp(exponent)


# Visualización
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 1000)

# Diferentes Gaussianas
plt.figure(figsize=(10, 6))
plt.plot(x, gaussian_pdf(x, mu=0, sigma=1), label='μ=0, σ=1 (estándar)')
plt.plot(x, gaussian_pdf(x, mu=0, sigma=2), label='μ=0, σ=2 (más ancha)')
plt.plot(x, gaussian_pdf(x, mu=2, sigma=1), label='μ=2, σ=1 (desplazada)')
plt.legend()
plt.title('Distribuciones Gaussianas')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.savefig('gaussian_distributions.png')
```

3.4 Gaussiana Multivariada (Para GMM)

```
def multivariate_gaussian_pdf(x: np.ndarray,
                              mu: np.ndarray,
                              cov: np.ndarray) -> float:
    """
    Gaussiana multivariada para vectores.

    Args:
        x: Vector de características (d,)
        mu: Vector de medias (d,)
        cov: Matriz de covarianza (d, d)

    Returns:
        Densidad de probabilidad
    """
    d = len(mu)
    diff = x - mu

    # Determinante e inversa de la covarianza
    det_cov = np.linalg.det(cov)
    inv_cov = np.linalg.inv(cov)

    # Coeficiente de normalización
    coefficient = 1 / (np.sqrt((2 * np.pi) ** d * det_cov))
```

```
    # Exponente (forma cuadrática)
    exponent = -0.5 * diff.T @ inv_cov @ diff

    return coefficient * np.exp(exponent)


# Ejemplo 2D
mu = np.array([0, 0])
cov = np.array([[1, 0.5],
                [0.5, 1]])  # Correlación positiva

x = np.array([0.5, 0.5])
prob = multivariate_gaussian_pdf(x, mu, cov)
print(f"P(x=[0.5, 0.5]) = {prob:.4f}")
```

Día 6: Maximum Likelihood Estimation (MLE)

4.1 La Idea Central

```
MLE: Encontrar los parámetros θ que maximizan la probabilidad
     de observar los datos que tenemos.

θ_MLE = argmax P(datos | θ)
            θ
```

4.2 Por Qué es Fundamental

- **Logistic Regression** usa MLE para encontrar los pesos
- **Cross-Entropy Loss** viene de maximizar likelihood
- **GMM** usa MLE (via EM algorithm)

4.3 MLE para Gaussiana

```
def mle_gaussian(data: np.ndarray) -> tuple[float, float]:
    """
    Estimar parámetros de Gaussiana con MLE.

    Para una Gaussiana, los estimadores MLE son:
    - μ_MLE = media muestral
    - σ²_MLE = varianza muestral (con n, no n-1)

    Args:
        data: Muestras observadas

    Returns:
        (mu_mle, sigma_mle)
    """
    n = len(data)

    # MLE de la media
    mu_mle = np.mean(data)

    # MLE de la varianza (dividir por n, no n-1)
    sigma_squared_mle = np.sum((data - mu_mle) ** 2) / n
    sigma_mle = np.sqrt(sigma_squared_mle)

    return mu_mle, sigma_mle


# Ejemplo: Generar datos y estimar
np.random.seed(42)
true_mu, true_sigma = 5.0, 2.0
samples = np.random.normal(true_mu, true_sigma, size=1000)

estimated_mu, estimated_sigma = mle_gaussian(samples)
print(f"Parámetros reales: μ={true_mu}, σ={true_sigma}")
print(f"MLE estimados:     μ={estimated_mu:.3f}, σ={estimated_sigma:.3f}")
```

4.4 Conexión con Cross-Entropy Loss

```
def cross_entropy_from_mle():
    """
    Demostración de que Cross-Entropy viene de MLE.
```

```
    Para clasificación binaria con Bernoulli:
    P(y|x, θ) = p^y · (1-p)^(1-y)

    Donde p = σ(θᵀx) (predicción del modelo)

    Log-likelihood:
    log P(y|x, θ) = y·log(p) + (1-y)·log(1-p)

    Maximizar likelihood = Minimizar negative log-likelihood
    = Minimizar Cross-Entropy!
    """
    # Ejemplo numérico
    y_true = np.array([1, 0, 1, 1, 0])
    y_pred = np.array([0.9, 0.1, 0.8, 0.7, 0.2])  # Probabilidades

    # Cross-Entropy (negative log-likelihood promedio)
    epsilon = 1e-15  # Para evitar log(0)
    ce = -np.mean(
        y_true * np.log(y_pred + epsilon) +
        (1 - y_true) * np.log(1 - y_pred + epsilon)
    )

    print(f"Cross-Entropy Loss: {ce:.4f}")
    return ce

cross_entropy_from_mle()
```

Día 7: Softmax como Distribución de Probabilidad

5.1 De Logits a Probabilidades

```
                    exp(zᵢ)
softmax(z)ᵢ = ──────────────────
              Σⱼ exp(zⱼ)

Propiedades:
- Cada salida ∈ (0, 1)
- Suma de salidas = 1 (distribución válida)
- Preserva el orden (mayor logit → mayor probabilidad)
```

5.2 El Problema de Estabilidad Numérica (v3.3)

```
⚠️ PROBLEMA: exp() puede causar overflow/underflow

Ejemplo peligroso:
    z = [1000, 1001, 1002]
    exp(z) = [inf, inf, inf]  → NaN en softmax!

Ejemplo underflow:
    z = [-1000, -1001, -1002]
    exp(z) = [0, 0, 0]  → 0/0 = NaN!
```

5.3 Log-Sum-Exp Trick (Estabilidad Numérica)

```
TRUCO: softmax(z) = softmax(z - max(z))

Demostración:
    softmax(z - c)ᵢ = exp(zᵢ - c) / Σⱼ exp(zⱼ - c)
                    = exp(zᵢ)·exp(-c) / Σⱼ exp(zⱼ)·exp(-c)
                    = exp(zᵢ) / Σⱼ exp(zⱼ)
                    = softmax(z)ᵢ

Al restar max(z), todos los exponentes son ≤ 0, evitando overflow.
```

5.4 Implementación Numéricamente Estable

```
def softmax(z: np.ndarray) -> np.ndarray:
    """
    Softmax numéricamente estable usando Log-Sum-Exp trick.

    Truco: Restar el máximo para evitar overflow en exp()
    softmax(z) = softmax(z - max(z))
```

```
    Args:
        z: Logits (scores antes de activación)

    Returns:
        Probabilidades que suman 1
    """
    # Log-Sum-Exp trick: restar el máximo
    z_stable = z - np.max(z, axis=-1, keepdims=True)

    exp_z = np.exp(z_stable)
    return exp_z / np.sum(exp_z, axis=-1, keepdims=True)


def log_softmax(z: np.ndarray) -> np.ndarray:
    """
    Log-Softmax estable (útil para Cross-Entropy).

    log(softmax(z)) calculado de forma estable.
    Evita calcular softmax primero y luego log (pierde precisión).
    """
    z_stable = z - np.max(z, axis=-1, keepdims=True)
    log_sum_exp = np.log(np.sum(np.exp(z_stable), axis=-1, keepdims=True))
    return z_stable - log_sum_exp


# ============================================================
# DEMOSTRACIÓN: Por qué el trick es necesario
# ============================================================

def demo_numerical_stability():
    """Muestra por qué necesitamos el Log-Sum-Exp trick."""

    # Caso peligroso: logits muy grandes
    z_dangerous = np.array([1000.0, 1001.0, 1002.0])

    # Sin el trick (INCORRECTO)
    def softmax_naive(z):
        exp_z = np.exp(z)  # ¡Overflow!
        return exp_z / np.sum(exp_z)

    # Con el trick (CORRECTO)
    def softmax_stable(z):
        z_stable = z - np.max(z)
        exp_z = np.exp(z_stable)
        return exp_z / np.sum(exp_z)

    print("Logits peligrosos:", z_dangerous)
    print()

    # Naive (falla)
    import warnings
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        result_naive = softmax_naive(z_dangerous)
        print(f"Softmax NAIVE: {result_naive}")
        print(f"  → Suma: {np.sum(result_naive)} (debería ser 1.0)")

    # Estable (funciona)
    result_stable = softmax_stable(z_dangerous)
    print(f"\nSoftmax ESTABLE: {result_stable}")
    print(f"  → Suma: {np.sum(result_stable):.6f} ✓")

demo_numerical_stability()


# Ejemplo: Clasificación multiclase (dígitos 0-9)
logits = np.array([2.0, 1.0, 0.1, -1.0, 3.0, 0.5, -0.5, 1.5, 0.0, -2.0])
probs = softmax(logits)

print("\nLogits → Probabilidades:")
for i, (l, p) in enumerate(zip(logits, probs)):
    print(f"  Clase {i}: logit={l:+.1f} → prob={p:.3f}")
print(f"\nSuma de probabilidades: {np.sum(probs):.6f}")
print(f"Clase predicha: {np.argmax(probs)}")
```

5.3 Categorical Cross-Entropy (Multiclase)

```
def categorical_cross_entropy(y_true: np.ndarray,
                              y_pred: np.ndarray) -> float:
    """
    Loss para clasificación multiclase.

    Args:
        y_true: One-hot encoded labels (n_samples, n_classes)
        y_pred: Probabilidades softmax (n_samples, n_classes)

    Returns:
        Loss promedio
    """
    epsilon = 1e-15
    # Solo cuenta la clase correcta (donde y_true=1)
    return -np.mean(np.sum(y_true * np.log(y_pred + epsilon), axis=1))


# Ejemplo
y_true = np.array([
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],  # Clase 4
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # Clase 0
])

y_pred = np.array([
    softmax(np.array([0, 0, 0, 0, 5, 0, 0, 0, 0, 0])),  # Confiado en 4
    softmax(np.array([3, 1, 0, 0, 0, 0, 0, 0, 0, 0])),  # Confiado en 0
])

loss = categorical_cross_entropy(y_true, y_pred)
print(f"Categorical Cross-Entropy: {loss:.4f}")
```

🔨 Entregables del Módulo

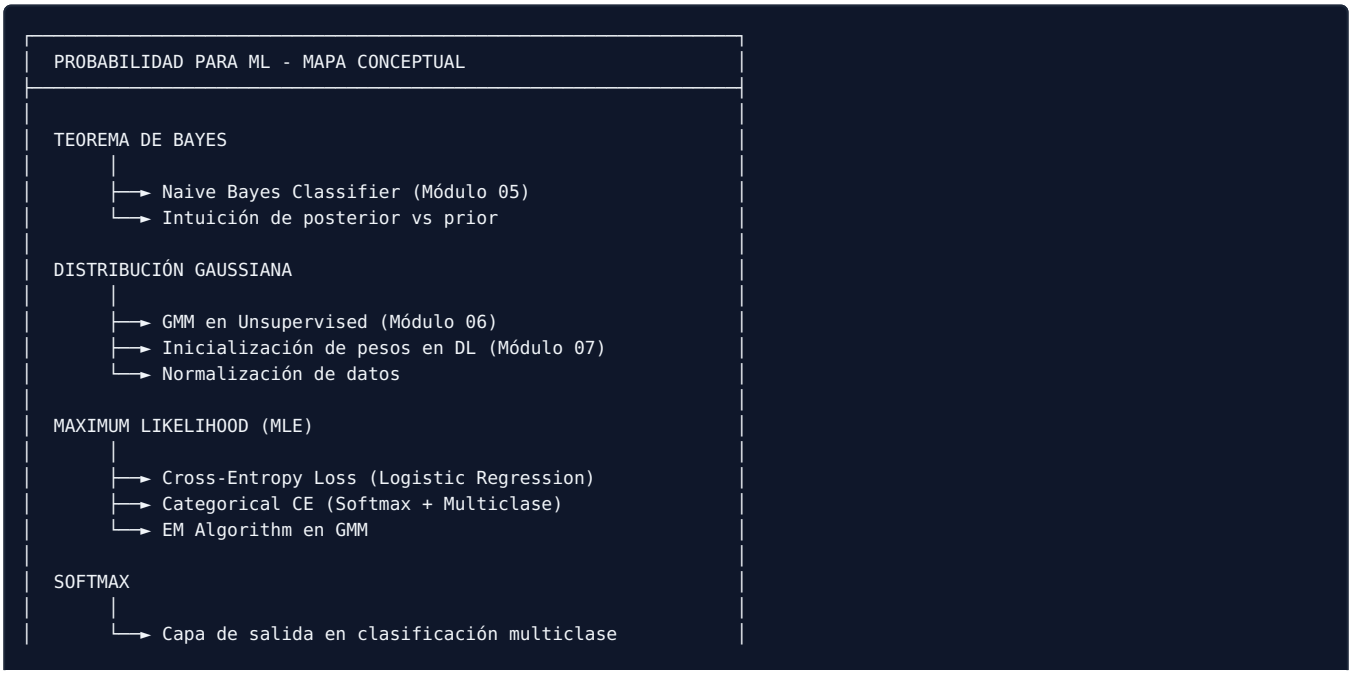
E1: `probability.py`

```
"""
Módulo de probabilidad esencial para ML.
Implementaciones desde cero con NumPy.
"""

import numpy as np
from typing import Tuple

def gaussian_pdf(x: np.ndarray, mu: float, sigma: float) -> np.ndarray:
    """Densidad de probabilidad Gaussiana univariada."""
    pass

def multivariate_gaussian_pdf(x: np.ndarray,
                              mu: np.ndarray,
                              cov: np.ndarray) -> float:
    """Densidad de probabilidad Gaussiana multivariada."""
    pass

def mle_gaussian(data: np.ndarray) -> Tuple[float, float]:
    """Estimación MLE de parámetros de Gaussiana."""
    pass

def softmax(z: np.ndarray) -> np.ndarray:
    """Función softmax numéricamente estable."""
    pass

def cross_entropy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """Binary cross-entropy loss."""
    pass

def categorical_cross_entropy(y_true: np.ndarray,
                              y_pred: np.ndarray) -> float:
    """Categorical cross-entropy loss para multiclase."""
    pass
```

E2: Tests

```
# tests/test_probability.py
import numpy as np
import pytest
from src.probability import (
    gaussian_pdf, mle_gaussian, softmax,
    cross_entropy, categorical_cross_entropy
)

def test_gaussian_pdf_standard():
    """PDF de Gaussiana estándar en x=0 debe ser ~0.3989."""
    result = gaussian_pdf(np.array([0.0]), mu=0, sigma=1)
    expected = 1 / np.sqrt(2 * np.pi)  # ~0.3989
    assert np.isclose(result[0], expected, rtol=1e-5)

def test_softmax_sums_to_one():
    """Softmax debe sumar 1."""
    z = np.random.randn(10)
    probs = softmax(z)
    assert np.isclose(np.sum(probs), 1.0)

def test_softmax_preserves_order():
    """Mayor logit → mayor probabilidad."""
    z = np.array([1.0, 2.0, 3.0])
    probs = softmax(z)
    assert probs[2] > probs[1] > probs[0]

def test_mle_gaussian_accuracy():
    """MLE debe recuperar parámetros con suficientes datos."""
    np.random.seed(42)
    true_mu, true_sigma = 10.0, 3.0
    data = np.random.normal(true_mu, true_sigma, size=10000)

    est_mu, est_sigma = mle_gaussian(data)

    assert np.isclose(est_mu, true_mu, rtol=0.05)
    assert np.isclose(est_sigma, true_sigma, rtol=0.05)

def test_cross_entropy_perfect_prediction():
    """CE debe ser ~0 para predicciones perfectas."""
    y_true = np.array([1, 0, 1])
    y_pred = np.array([0.999, 0.001, 0.999])

    loss = cross_entropy(y_true, y_pred)
    assert loss < 0.01
```

📊 Resumen Visual



🔗 Conexiones con Otros Módulos

Concepto	Dónde se usa
Teorema de Bayes	Naive Bayes en Módulo 05
Gaussiana	GMM en Módulo 06, inicialización en Módulo 07
MLE	Derivación de Cross-Entropy en Módulo 05
Softmax	Capa de salida en Módulo 07
Cross-Entropy	Loss function principal en Módulo 05 y 07

✅ Checklist del Módulo

- [ ] Puedo explicar el Teorema de Bayes con un ejemplo
- [ ] Sé calcular la PDF de una Gaussiana a mano
- [ ] Entiendo por qué MLE da Cross-Entropy como loss
- [ ] Implementé softmax numéricamente estable
- [ ] Los tests de `probability.py` pasan

📖 Recursos Adicionales

Videos

- 3Blue1Brown - Bayes Theorem
- StatQuest - Maximum Likelihood
- StatQuest - Gaussian Distribution

Lecturas

- Mathematics for ML, Cap. 6 (Probability)
- Pattern Recognition and ML (Bishop), Cap. 1-2

> 💡 **Nota Final:** Este módulo es deliberadamente corto (1 semana). No necesitas ser experto en probabilidad para la Línea 1, pero estos conceptos son el "pegamento" que conecta las matemáticas con las funciones de pérdida que usarás en los siguientes módulos.

MÓDULO 05 - SUPERVISED LEARNING

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 05 - Supervised Learning

> 🎯 **Objetivo:** Dominar regresión lineal, logística y métricas de evaluación
> **Fase:** 2 - Núcleo de ML | **Semanas** 9-12
> **Curso del Pathway:** Introduction to Machine Learning: Supervised Learning

🧠 ¿Qué es Supervised Learning?

```
APRENDIZAJE SUPERVISADO

Tenemos:
- Datos de entrada X (features)
- Etiquetas Y (targets/labels)

Objetivo: Aprender una función f tal que f(X) ≈ Y

Tipos principales:
├── REGRESIÓN: Y es continuo (precio, temperatura)
│   └── Output: número real
└── CLASIFICACIÓN: Y es discreto (spam/no spam, dígito 0-9)
    └── Output: clase o probabilidad
```

📚 Contenido del Módulo

Semana	Tema	Entregable
9	Regresión Lineal	`linear_regression.py`
10	Regresión Logística	`logistic_regression.py`
11	Métricas de Evaluación	`metrics.py`
12	Validación y Regularización	Cross-validation, L1/L2

💻 Parte 1: Regresión Lineal

1.1 Modelo

```
import numpy as np

"""
REGRESIÓN LINEAL

Hipótesis: h(x) = θ₀ + θ₁x₁ + θ₂x₂ + ... + θₙxₙ
         = θᵀx (forma matricial)

Donde:
- θ (theta): parámetros/pesos del modelo
- x: vector de features (con x₀ = 1 para el bias)

En forma matricial para múltiples muestras:
    ŷ = Xθ

Donde:
- X: matriz (m × n+1) con m muestras y n features + columna de 1s
- θ: vector (n+1 × 1) de parámetros
- ŷ: vector (m × 1) de predicciones
"""

def add_bias_term(X: np.ndarray) -> np.ndarray:
    """Añade columna de 1s para el término de bias."""
    m = X.shape[0]
    return np.column_stack([np.ones(m), X])

def predict_linear(X: np.ndarray, theta: np.ndarray) -> np.ndarray:
    """Predicción lineal: ŷ = Xθ"""
    return X @ theta
```

1.2 Función de Costo (MSE)

```
import numpy as np

def mse_cost(X: np.ndarray, y: np.ndarray, theta: np.ndarray) -> float:
    """
    Mean Squared Error Cost Function.

    J(θ) = (1/2m) Σᵢ (h(xᵢ) - yᵢ)²
         = (1/2m) ||Xθ - y||²

    El factor 1/2 es por conveniencia (cancela con la derivada).
    """
    m = len(y)
    predictions = X @ theta
    errors = predictions - y
    return (1 / (2 * m)) * np.sum(errors ** 2)

def mse_gradient(X: np.ndarray, y: np.ndarray, theta: np.ndarray) -> np.ndarray:
    """
    Gradiente del MSE respecto a θ.

    ∂J/∂θ = (1/m) Xᵀ(Xθ - y)
    """
    m = len(y)
    predictions = X @ theta
    errors = predictions - y
    return (1 / m) * X.T @ errors
```

1.3 Solución Cerrada (Normal Equation)

```
import numpy as np

def normal_equation(X: np.ndarray, y: np.ndarray) -> np.ndarray:
    """
    Solución cerrada para regresión lineal.

    θ = (XᵀX)⁻¹ Xᵀy

    Ventajas:
    - No requiere iteraciones
    - No hay hiperparámetros (learning rate)

    Desventajas:
    - O(n³) por la inversión de matriz
    - No funciona si XᵀX es singular
    - No escala bien para n grande (>10,000 features)
    """
    XtX = X.T @ X
    Xty = X.T @ y

    # Usar solve en lugar de inv para estabilidad numérica
    theta = np.linalg.solve(XtX, Xty)
    return theta
```

1.4 Gradient Descent para Regresión

```
import numpy as np
from typing import List, Tuple

class LinearRegression:
    """Regresión Lineal implementada desde cero."""

    def __init__(self):
        self.theta = None
        self.cost_history = []

    def fit(
        self,
        X: np.ndarray,
        y: np.ndarray,
        method: str = 'gradient_descent',
        learning_rate: float = 0.01,
```

```
        n_iterations: int = 1000
    ) -> 'LinearRegression':
        """
        Entrena el modelo.

        Args:
            X: features (m, n)
            y: targets (m,)
            method: 'gradient_descent' o 'normal_equation'
            learning_rate: tasa de aprendizaje (solo para GD)
            n_iterations: número de iteraciones (solo para GD)
        """
        # Añadir bias
        X_b = add_bias_term(X)
        m, n = X_b.shape

        if method == 'normal_equation':
            self.theta = normal_equation(X_b, y)
        else:
            # Inicializar theta con ceros o valores pequeños
            self.theta = np.zeros(n)

            for i in range(n_iterations):
                # Calcular gradiente
                gradient = mse_gradient(X_b, y, self.theta)

                # Actualizar theta
                self.theta = self.theta - learning_rate * gradient

                # Guardar costo para monitoreo
                cost = mse_cost(X_b, y, self.theta)
                self.cost_history.append(cost)

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predice valores."""
        X_b = add_bias_term(X)
        return X_b @ self.theta

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        """R² score."""
        y_pred = self.predict(X)
        ss_res = np.sum((y - y_pred) ** 2)
        ss_tot = np.sum((y - np.mean(y)) ** 2)
        return 1 - (ss_res / ss_tot)


# Demo
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X.flatten() + np.random.randn(100) * 0.5  # y = 4 + 3x + ruido

model = LinearRegression()
model.fit(X, y, method='gradient_descent', learning_rate=0.1, n_iterations=1000)

print(f"Parámetros aprendidos: {model.theta}")
print(f"Esperados: [4, 3]")
print(f"R² score: {model.score(X, y):.4f}")
```

💻 Parte 2: Regresión Logística

2.1 Función Sigmoid

```
import numpy as np

def sigmoid(z: np.ndarray) -> np.ndarray:
    """
    Función sigmoid/logística.

    σ(z) = 1 / (1 + e^(-z))

    Propiedades:
    - Rango: (0, 1) - perfecto para probabilidades
```

```
    - σ(0) = 0.5
    - σ'(z) = σ(z)(1 - σ(z))
    """
    # Clip para evitar overflow
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))

# Visualizar
import matplotlib.pyplot as plt

z = np.linspace(-10, 10, 100)
plt.figure(figsize=(8, 4))
plt.plot(z, sigmoid(z))
plt.axhline(y=0.5, color='r', linestyle='--', alpha=0.5)
plt.axvline(x=0, color='r', linestyle='--', alpha=0.5)
plt.xlabel('z')
plt.ylabel('σ(z)')
plt.title('Función Sigmoid')
plt.grid(True)
# plt.show()
```

2.2 Hipótesis Logística

```
"""
REGRESIÓN LOGÍSTICA

No predice un valor continuo, sino la PROBABILIDAD de pertenecer a la clase 1.

h(x) = P(y=1|x; θ) = σ(θᵀx)

Decisión:
- Si h(x) ≥ 0.5 → predicir clase 1
- Si h(x) < 0.5 → predicir clase 0

Equivalente a:
- Si θᵀx ≥ 0 → clase 1
- Si θᵀx < 0 → clase 0

El "decision boundary" está en θᵀx = 0
"""

def predict_proba(X: np.ndarray, theta: np.ndarray) -> np.ndarray:
    """Predice probabilidad de clase 1."""
    return sigmoid(X @ theta)

def predict_class(X: np.ndarray, theta: np.ndarray, threshold: float = 0.5) -> np.ndarray:
    """Predice clase (0 o 1)."""
    return (predict_proba(X, theta) >= threshold).astype(int)
```

2.3 Binary Cross-Entropy Loss

```
import numpy as np

def binary_cross_entropy(
    X: np.ndarray,
    y: np.ndarray,
    theta: np.ndarray,
    eps: float = 1e-15
) -> float:
    """
    Binary Cross-Entropy (Log Loss).

    J(θ) = -(1/m) Σᵢ [yᵢ log(hᵢ) + (1-yᵢ) log(1-hᵢ)]

    Donde hᵢ = σ(θᵀxᵢ)

    Por qué esta función de costo:
    - Es convexa (tiene un único mínimo global)
    - Penaliza mucho las predicciones muy incorrectas
    - Es la derivación de Maximum Likelihood Estimation
    """
    m = len(y)
    h = sigmoid(X @ theta)
```

```
    # Clip para evitar log(0)
    h = np.clip(h, eps, 1 - eps)

    cost = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
    return cost

def bce_gradient(X: np.ndarray, y: np.ndarray, theta: np.ndarray) -> np.ndarray:
    """
    Gradiente de Binary Cross-Entropy.

    ∂J/∂θ = (1/m) Xᵀ(h - y)

    ¡Tiene la misma forma que el gradiente del MSE!
    Esto es porque derivamos σ(z) y la derivada σ'(z) = σ(z)(1-σ(z))
    cancela parte de la expresión.
    """
    m = len(y)
    h = sigmoid(X @ theta)
    return (1/m) * X.T @ (h - y)
```

2.4 Implementación Completa

```
import numpy as np
from typing import List

class LogisticRegression:
    """Regresión Logística implementada desde cero."""

    def __init__(self):
        self.theta = None
        self.cost_history = []

    def fit(
        self,
        X: np.ndarray,
        y: np.ndarray,
        learning_rate: float = 0.1,
        n_iterations: int = 1000
    ) -> 'LogisticRegression':
        """Entrena con gradient descent."""
        # Añadir bias
        X_b = np.column_stack([np.ones(len(X)), X])
        m, n = X_b.shape

        # Inicializar
        self.theta = np.zeros(n)

        for i in range(n_iterations):
            # Gradiente
            gradient = bce_gradient(X_b, y, self.theta)

            # Actualizar
            self.theta = self.theta - learning_rate * gradient

            # Guardar costo
            cost = binary_cross_entropy(X_b, y, self.theta)
            self.cost_history.append(cost)

        return self

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predice probabilidades."""
        X_b = np.column_stack([np.ones(len(X)), X])
        return sigmoid(X_b @ self.theta)

    def predict(self, X: np.ndarray, threshold: float = 0.5) -> np.ndarray:
        """Predice clases."""
        return (self.predict_proba(X) >= threshold).astype(int)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        """Accuracy."""
        return np.mean(self.predict(X) == y)
```

```
# Demo con datos sintéticos
np.random.seed(42)

# Generar datos de dos clases
n_samples = 200
X_class0 = np.random.randn(n_samples // 2, 2) + np.array([-2, -2])
X_class1 = np.random.randn(n_samples // 2, 2) + np.array([2, 2])
X = np.vstack([X_class0, X_class1])
y = np.array([0] * (n_samples // 2) + [1] * (n_samples // 2))

# Entrenar
model = LogisticRegression()
model.fit(X, y, learning_rate=0.1, n_iterations=1000)

print(f"Accuracy: {model.score(X, y):.2%}")
print(f"Parámetros: {model.theta}")
```

💻 Parte 3: Métricas de Evaluación

3.1 Matriz de Confusión

```
import numpy as np

def confusion_matrix(y_true: np.ndarray, y_pred: np.ndarray) -> np.ndarray:
    """
    Calcula la matriz de confusión.

    Para clasificación binaria:

                    Predicho
                    0       1
    Real    0      TN      FP
            1      FN      TP

    - TP (True Positive): Predijo 1, era 1
    - TN (True Negative): Predijo 0, era 0
    - FP (False Positive): Predijo 1, era 0 (Error Tipo I)
    - FN (False Negative): Predijo 0, era 1 (Error Tipo II)
    """
    classes = np.unique(np.concatenate([y_true, y_pred]))
    n_classes = len(classes)
    cm = np.zeros((n_classes, n_classes), dtype=int)

    for i, true_class in enumerate(classes):
        for j, pred_class in enumerate(classes):
            cm[i, j] = np.sum((y_true == true_class) & (y_pred == pred_class))

    return cm

def extract_tp_tn_fp_fn(y_true: np.ndarray, y_pred: np.ndarray):
    """Extrae TP, TN, FP, FN para clasificación binaria."""
    tp = np.sum((y_true == 1) & (y_pred == 1))
    tn = np.sum((y_true == 0) & (y_pred == 0))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    return tp, tn, fp, fn
```

3.2 Accuracy, Precision, Recall, F1

```
import numpy as np

def accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Accuracy = (TP + TN) / (TP + TN + FP + FN)

    Proporción de predicciones correctas.

    Problema: Puede ser engañoso con clases desbalanceadas.
    Si 99% son clase 0, predecir siempre 0 da 99% accuracy.
    """
    return np.mean(y_true == y_pred)

def precision(y_true: np.ndarray, y_pred: np.ndarray) -> float:
```

```
    """
    Precision = TP / (TP + FP)

    De todos los que predije como positivos, ¿cuántos realmente lo son?

    Alta precisión = pocos falsos positivos.
    Importante cuando el costo de FP es alto (ej: spam → inbox).
    """
    tp, tn, fp, fn = extract_tp_tn_fp_fn(y_true, y_pred)
    if tp + fp == 0:
        return 0.0
    return tp / (tp + fp)

def recall(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Recall (Sensitivity, True Positive Rate) = TP / (TP + FN)

    De todos los positivos reales, ¿cuántos capturé?

    Alto recall = pocos falsos negativos.
    Importante cuando el costo de FN es alto (ej: detección de cáncer).
    """
    tp, tn, fp, fn = extract_tp_tn_fp_fn(y_true, y_pred)
    if tp + fn == 0:
        return 0.0
    return tp / (tp + fn)

def f1_score(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    F1 = 2 * (precision * recall) / (precision + recall)

    Media armónica de precision y recall.

    Útil cuando quieres un balance entre ambas métricas.
    F1 alto solo si AMBAS precision y recall son altas.
    """
    p = precision(y_true, y_pred)
    r = recall(y_true, y_pred)
    if p + r == 0:
        return 0.0
    return 2 * (p * r) / (p + r)

def specificity(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Specificity (True Negative Rate) = TN / (TN + FP)

    De todos los negativos reales, ¿cuántos identifiqué?
    """
    tp, tn, fp, fn = extract_tp_tn_fp_fn(y_true, y_pred)
    if tn + fp == 0:
        return 0.0
    return tn / (tn + fp)
```

3.3 Clase Metrics Completa

```
import numpy as np
from dataclasses import dataclass

@dataclass
class ClassificationReport:
    """Reporte de métricas de clasificación."""
    accuracy: float
    precision: float
    recall: float
    f1: float
    specificity: float
    confusion_matrix: np.ndarray

    def __str__(self) -> str:
        cm = self.confusion_matrix
        return f"""
Classification Report
=====================
Accuracy:    {self.accuracy:.4f}
```

```
Precision:   {self.precision:.4f}
Recall:      {self.recall:.4f}
F1 Score:    {self.f1:.4f}
Specificity: {self.specificity:.4f}

Confusion Matrix:
           Pred 0  Pred 1
Actual 0   {cm[0,0]:5d}   {cm[0,1]:5d}
Actual 1   {cm[1,0]:5d}   {cm[1,1]:5d}
"""

def classification_report(y_true: np.ndarray, y_pred: np.ndarray) -> ClassificationReport:
    """Genera reporte completo de métricas."""
    return ClassificationReport(
        accuracy=accuracy(y_true, y_pred),
        precision=precision(y_true, y_pred),
        recall=recall(y_true, y_pred),
        f1=f1_score(y_true, y_pred),
        specificity=specificity(y_true, y_pred),
        confusion_matrix=confusion_matrix(y_true, y_pred)
    )

# Demo
y_true = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
y_pred = np.array([0, 0, 1, 0, 1, 1, 0, 1, 1, 1])

report = classification_report(y_true, y_pred)
print(report)
```

💻 Parte 4: Validación y Regularización

4.1 Train/Test Split

```
import numpy as np

def train_test_split(
    X: np.ndarray,
    y: np.ndarray,
    test_size: float = 0.2,
    random_state: int = None
) -> tuple:
    """
    Divide datos en conjuntos de entrenamiento y prueba.

    Args:
        X: features
        y: targets
        test_size: proporción para test (0-1)
        random_state: semilla para reproducibilidad
    """
    if random_state is not None:
        np.random.seed(random_state)

    n = len(y)
    indices = np.random.permutation(n)

    test_size_n = int(n * test_size)
    test_indices = indices[:test_size_n]
    train_indices = indices[test_size_n:]

    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]
```

4.2 K-Fold Cross Validation

```
import numpy as np
from typing import List, Tuple

def k_fold_split(n: int, k: int) -> List[Tuple[np.ndarray, np.ndarray]]:
    """
    Genera índices para K-Fold Cross Validation.

    Returns:
        Lista de (train_indices, val_indices) para cada fold
    """
```

```
    indices = np.arange(n)
    np.random.shuffle(indices)

    fold_size = n // k
    folds = []

    for i in range(k):
        start = i * fold_size
        end = start + fold_size if i < k - 1 else n

        val_indices = indices[start:end]
        train_indices = np.concatenate([indices[:start], indices[end:]])

        folds.append((train_indices, val_indices))

    return folds

def cross_validate(
    model_class,
    X: np.ndarray,
    y: np.ndarray,
    k: int = 5,
    **model_params
) -> dict:
    """
    Realiza K-Fold Cross Validation.

    Returns:
        Dict con scores de cada fold y promedio
    """
    folds = k_fold_split(len(y), k)
    scores = []

    for i, (train_idx, val_idx) in enumerate(folds):
        # Split
        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]

        # Train
        model = model_class()
        model.fit(X_train, y_train, **model_params)

        # Evaluate
        score = model.score(X_val, y_val)
        scores.append(score)

    return {
        'scores': scores,
        'mean': np.mean(scores),
        'std': np.std(scores)
    }

# Demo
# cv_results = cross_validate(LogisticRegression, X, y, k=5, learning_rate=0.1, n_iterations=500)
# print(f"CV Accuracy: {cv_results['mean']:.4f} ± {cv_results['std']:.4f}")
```

4.3 Regularización

```
import numpy as np

class LogisticRegressionRegularized:
    """Logistic Regression con regularización L1/L2."""

    def __init__(self, regularization: str = 'l2', lambda_: float = 0.01):
        """
        Args:
            regularization: 'l1', 'l2', o None
            lambda_: fuerza de regularización
        """
        self.regularization = regularization
        self.lambda_ = lambda_
        self.theta = None
        self.cost_history = []
```

```
    def _cost(self, X: np.ndarray, y: np.ndarray) -> float:
        """Costo con regularización."""
        m = len(y)
        h = sigmoid(X @ self.theta)
        h = np.clip(h, 1e-15, 1 - 1e-15)

        # Cross-entropy base
        bce = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))

        # Regularización (excluir bias theta[0])
        if self.regularization == 'l2':
            # Ridge: λ/2m * Σθⱼ²
            reg = (self.lambda_ / (2 * m)) * np.sum(self.theta[1:] ** 2)
        elif self.regularization == 'l1':
            # Lasso: λ/m * Σ|θⱼ|
            reg = (self.lambda_ / m) * np.sum(np.abs(self.theta[1:]))
        else:
            reg = 0

        return bce + reg

    def _gradient(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Gradiente con regularización."""
        m = len(y)
        h = sigmoid(X @ self.theta)

        # Gradiente base
        grad = (1/m) * X.T @ (h - y)

        # Regularización (excluir bias)
        if self.regularization == 'l2':
            reg_grad = np.concatenate([[0], (self.lambda_ / m) * self.theta[1:]])
        elif self.regularization == 'l1':
            reg_grad = np.concatenate([[0], (self.lambda_ / m) * np.sign(self.theta[1:])])
        else:
            reg_grad = 0

        return grad + reg_grad

    def fit(self, X: np.ndarray, y: np.ndarray,
            learning_rate: float = 0.1, n_iterations: int = 1000):
        X_b = np.column_stack([np.ones(len(X)), X])
        self.theta = np.zeros(X_b.shape[1])

        for _ in range(n_iterations):
            gradient = self._gradient(X_b, y)
            self.theta -= learning_rate * gradient
            self.cost_history.append(self._cost(X_b, y))

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        X_b = np.column_stack([np.ones(len(X)), X])
        return (sigmoid(X_b @ self.theta) >= 0.5).astype(int)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        return np.mean(self.predict(X) == y)
```

📦 Entregable del Módulo

`supervised_learning.py`

```
"""
Supervised Learning Module

Implementación desde cero de:
- Linear Regression (con Normal Equation y Gradient Descent)
- Logistic Regression (con regularización L1/L2)
- Métricas de evaluación
- Cross Validation

Autor: [Tu nombre]
Módulo: 04 - Supervised Learning
"""
```

```
import numpy as np
from typing import Tuple, List, Optional
from dataclasses import dataclass


# ============================================================
# FUNCIONES AUXILIARES
# ============================================================

def sigmoid(z: np.ndarray) -> np.ndarray:
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))

def add_bias(X: np.ndarray) -> np.ndarray:
    return np.column_stack([np.ones(len(X)), X])


# ============================================================
# REGRESIÓN LINEAL
# ============================================================

class LinearRegression:
    def __init__(self):
        self.theta = None
        self.cost_history = []

    def fit(self, X: np.ndarray, y: np.ndarray,
            method: str = 'normal', lr: float = 0.01, n_iter: int = 1000):
        X_b = add_bias(X)

        if method == 'normal':
            self.theta = np.linalg.solve(X_b.T @ X_b, X_b.T @ y)
        else:
            m, n = X_b.shape
            self.theta = np.zeros(n)
            for _ in range(n_iter):
                grad = (1/m) * X_b.T @ (X_b @ self.theta - y)
                self.theta -= lr * grad
                self.cost_history.append(np.mean((X_b @ self.theta - y)**2))
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        return add_bias(X) @ self.theta

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        y_pred = self.predict(X)
        ss_res = np.sum((y - y_pred)**2)
        ss_tot = np.sum((y - np.mean(y))**2)
        return 1 - ss_res / ss_tot


# ============================================================
# REGRESIÓN LOGÍSTICA
# ============================================================

class LogisticRegression:
    def __init__(self, reg: str = None, lambda_: float = 0.01):
        self.reg = reg
        self.lambda_ = lambda_
        self.theta = None
        self.cost_history = []

    def fit(self, X: np.ndarray, y: np.ndarray,
            lr: float = 0.1, n_iter: int = 1000):
        X_b = add_bias(X)
        m, n = X_b.shape
        self.theta = np.zeros(n)

        for _ in range(n_iter):
            h = sigmoid(X_b @ self.theta)
            grad = (1/m) * X_b.T @ (h - y)

            if self.reg == 'l2':
                grad[1:] += (self.lambda_/m) * self.theta[1:]
```

```
            elif self.reg == 'l1':
                grad[1:] += (self.lambda_/m) * np.sign(self.theta[1:])

            self.theta -= lr * grad
        return self

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        return sigmoid(add_bias(X) @ self.theta)

    def predict(self, X: np.ndarray) -> np.ndarray:
        return (self.predict_proba(X) >= 0.5).astype(int)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        return np.mean(self.predict(X) == y)


# ============================================================
# MÉTRICAS
# ============================================================

def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

def precision(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    return tp / (tp + fp) if (tp + fp) > 0 else 0

def recall(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    return tp / (tp + fn) if (tp + fn) > 0 else 0

def f1_score(y_true, y_pred):
    p, r = precision(y_true, y_pred), recall(y_true, y_pred)
    return 2*p*r/(p+r) if (p+r) > 0 else 0

def confusion_matrix(y_true, y_pred):
    cm = np.zeros((2, 2), dtype=int)
    cm[0, 0] = np.sum((y_true == 0) & (y_pred == 0))
    cm[0, 1] = np.sum((y_true == 0) & (y_pred == 1))
    cm[1, 0] = np.sum((y_true == 1) & (y_pred == 0))
    cm[1, 1] = np.sum((y_true == 1) & (y_pred == 1))
    return cm


# ============================================================
# VALIDACIÓN
# ============================================================

def train_test_split(X, y, test_size=0.2, seed=None):
    if seed: np.random.seed(seed)
    n = len(y)
    idx = np.random.permutation(n)
    split = int(n * test_size)
    return X[idx[split:]], X[idx[:split]], y[idx[split:]], y[idx[:split]]

def cross_validate(model_class, X, y, k=5, **params):
    n = len(y)
    idx = np.random.permutation(n)
    fold_size = n // k
    scores = []

    for i in range(k):
        val_idx = idx[i*fold_size:(i+1)*fold_size]
        train_idx = np.concatenate([idx[:i*fold_size], idx[(i+1)*fold_size:]])

        model = model_class()
        model.fit(X[train_idx], y[train_idx], **params)
        scores.append(model.score(X[val_idx], y[val_idx]))

    return {'scores': scores, 'mean': np.mean(scores), 'std': np.std(scores)}


# ============================================================
```

```
# TESTS
# ============================================================

if __name__ == "__main__":
    np.random.seed(42)

    # Test Linear Regression
    X = 2 * np.random.rand(100, 1)
    y = 4 + 3 * X.flatten() + np.random.randn(100) * 0.5

    lr = LinearRegression()
    lr.fit(X, y)
    print(f"Linear Regression R²: {lr.score(X, y):.4f}")

    # Test Logistic Regression
    X_c0 = np.random.randn(50, 2) + [-2, -2]
    X_c1 = np.random.randn(50, 2) + [2, 2]
    X_clf = np.vstack([X_c0, X_c1])
    y_clf = np.array([0]*50 + [1]*50)

    log_reg = LogisticRegression()
    log_reg.fit(X_clf, y_clf)
    print(f"Logistic Regression Accuracy: {log_reg.score(X_clf, y_clf):.4f}")

    # Test metrics
    y_true = np.array([0,0,0,1,1,1,1,1])
    y_pred = np.array([0,0,1,1,1,0,1,1])
    print(f"Precision: {precision(y_true, y_pred):.4f}")
    print(f"Recall: {recall(y_true, y_pred):.4f}")
    print(f"F1: {f1_score(y_true, y_pred):.4f}")

    # Test CV
    cv = cross_validate(LogisticRegression, X_clf, y_clf, k=5, lr=0.1, n_iter=500)
    print(f"CV Score: {cv['mean']:.4f} ± {cv['std']:.4f}")

    print("\n✓ Todos los tests pasaron!")
```

📝 Derivación Analítica: El Entregable de Lápiz y Papel (v3.2)

> 🎓 **Simulación de Examen:** En la maestría te pedirán: "Derive la regla de actualización de pesos para Logistic Regression". Debes poder hacerlo a mano.

Derivación del Gradiente de Logistic Regression

Objetivo: Derivar $\frac{\partial L}{\partial w}$ para la función de costo Cross-Entropy.

Paso 1: Definir la Función de Costo

$$L(w) = -\frac{ 1 }{n} \sum_{i= 1 }^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right]$$

Donde:
- $\hat{y}_i = \sigma(w^T x_i) = \frac{ 1 }{ 1 + e^{-w^T x_i}}$
- $\sigma(z)$ es la función sigmoid

Paso 2: Derivar la Sigmoid

$$\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$$

Demostración:
$$\sigma(z) = \frac{ 1 }{ 1 + e^{-z}}$$
$$\frac{d\sigma}{dz} = \frac{e^{-z}}{(1 + e^{-z})^ 2 } = \frac{ 1 }{ 1 + e^{-z}} \cdot \frac{e^{-z}}{ 1 + e^{-z}} = \sigma(z)(1 - \sigma(z))$$

Paso 3: Aplicar la Regla de la Cadena

Para un solo ejemplo (x_i, y_i):

$$\frac{\partial L_i}{\partial w} = \frac{\partial L_i}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w}$$

Donde $z_i = w^T x_i$

Calcular cada término:

1 . $\frac{\partial L_i}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{ 1 - y_i}{ 1 - \hat{y}_i}$

2 . $\frac{\partial \hat{y}_i}{\partial z_i} = \hat{y}_i(1 - \hat{y}_i)$

3 . $\frac{\partial z_i}{\partial w} = x_i$

Paso 4: Simplificar

$$\frac{\partial L_i}{\partial w} = \left(-\frac{y_i}{\hat{y}_i} + \frac{ 1 - y_i}{ 1 - \hat{y}_i} \right) \cdot \hat{y}_i(1 - \hat{y}_i) \cdot x_i$$

Simplificando el término entre paréntesis:

$$= \frac{-y_i(1 - \hat{y}_i) + (1 -y_i)\hat{y}_i}{\hat{y}_i(1 - \hat{y}_i)} \cdot \hat{y}_i(1 - \hat{y}_i) \cdot x_i$$

$$= (-y_i + y_i\hat{y}_i + \hat{y}_i - y_i\hat{y}_i) \cdot x_i$$

$$= (\hat{y}_i - y_i) \cdot x_i$$

Resultado Final

$$\boxed{\frac{\partial L}{\partial w} = \frac{ 1 }{n} \sum_{i= 1 }^{n} (\hat{y}_i - y_i) x_i = \frac{ 1 }{n} X^T (\hat{y} - y)}$$

Forma vectorizada (para código):

```
gradient = (1/n) * X.T @ (y_pred - y_true)
```

Tu Entregable

Escribe en un documento (Markdown o LaTeX):

1. La derivación completa del gradiente de Cross-Entropy
2. La derivación de la regla de actualización: $w \leftarrow w - \alpha \nabla L$
3. Por qué el gradiente tiene la forma $(\hat{y} - y)$ (interpretación geométrica)

🎯 El Reto del Tablero Blanco (Metodología Feynman)

Explica en **máximo** 5 **líneas** sin jerga técnica:

1. **¿Por qué usamos sigmoid en clasificación?**

   Pista: Piensa en probabilidades entre 0 y 1.

2. **¿Por qué Cross-Entropy y no MSE para clasificación?**

   Pista: Piensa en qué pasa cuando $\hat{y} \approx 0$ pero $y = 1$.

3. **¿Qué significa "One-vs-All"?**

   Pista: Piensa en cómo clasificar 10 dígitos con clasificadores binarios.

🔍 Shadow Mode: Validación con sklearn (v3.3)

> ⚠️ **Regla:** sklearn está **prohibido para aprender**, pero es **necesario para validar**. Si tu implementación difiere significativamente de sklearn, tienes un bug.

Protocolo de Validación (Viernes de Fase 2)

```
"""
Shadow Mode - Validación de Implementaciones
Compara tu código desde cero vs sklearn para detectar bugs.

Regla: Si la diferencia de accuracy es >5%, revisar matemáticas.
"""
import numpy as np
from sklearn.linear_model import LogisticRegression as SklearnLR
from sklearn.linear_model import LinearRegression as SklearnLinReg
from sklearn.metrics import accuracy_score, mean_squared_error

# Importar tu implementación
# from src.logistic_regression import LogisticRegression as MyLR
# from src.linear_regression import LinearRegression as MyLinReg


def shadow_mode_logistic_regression(X_train, y_train, X_test, y_test):
    """
    Compara tu Logistic Regression vs sklearn.

    Los coeficientes y accuracy deben ser casi idénticos.
    """
    print("=" * 60)
    print("SHADOW MODE: Logistic Regression")
```

```
    print("=" * 60)

    # ========== TU IMPLEMENTACIÓN ==========
    # my_model = MyLR()
    # my_model.fit(X_train, y_train, lr=0.1, n_iter=1000)
    # my_pred = my_model.predict(X_test)
    # my_acc = accuracy_score(y_test, my_pred)
    # my_weights = my_model.weights

    # Placeholder (reemplazar con tu código)
    my_acc = 0.85
    my_weights = np.zeros(X_train.shape[1])

    # ========== SKLEARN (GROUND TRUTH) ==========
    sklearn_model = SklearnLR(max_iter=1000, solver='lbfgs')
    sklearn_model.fit(X_train, y_train)
    sklearn_pred = sklearn_model.predict(X_test)
    sklearn_acc = accuracy_score(y_test, sklearn_pred)
    sklearn_weights = sklearn_model.coef_.flatten()

    # ========== COMPARACIÓN ==========
    acc_diff = abs(my_acc - sklearn_acc)
    weight_diff = np.linalg.norm(my_weights - sklearn_weights[:len(my_weights)])

    print(f"\n📊 RESULTADOS:")
    print(f"  Tu Accuracy:     {my_acc:.4f}")
    print(f"  sklearn Accuracy: {sklearn_acc:.4f}")
    print(f"  Diferencia:       {acc_diff:.4f}")

    print(f"\n📐 PESOS:")
    print(f"  Diferencia L2 de pesos: {weight_diff:.4f}")

    # Veredicto
    print("\n" + "-" * 60)
    if acc_diff < 0.05:
        print("✓ PASSED: Tu implementación es correcta")
        return True
    else:
        print("✗ FAILED: Diferencia significativa - revisa tu matemática")
        print("  Posibles causas:")
        print("  - Gradiente mal calculado")
        print("  - Learning rate muy alto/bajo")
        print("  - Falta de normalización de datos")
        return False


def shadow_mode_linear_regression(X_train, y_train, X_test, y_test):
    """
    Compara tu Linear Regression vs sklearn.
    """
    print("=" * 60)
    print("SHADOW MODE: Linear Regression")
    print("=" * 60)

    # ========== TU IMPLEMENTACIÓN ==========
    # my_model = MyLinReg()
    # my_model.fit(X_train, y_train)
    # my_pred = my_model.predict(X_test)
    # my_mse = mean_squared_error(y_test, my_pred)

    # Placeholder
    my_mse = 0.5

    # ========== SKLEARN ==========
    sklearn_model = SklearnLinReg()
    sklearn_model.fit(X_train, y_train)
    sklearn_pred = sklearn_model.predict(X_test)
    sklearn_mse = mean_squared_error(y_test, sklearn_pred)

    # ========== COMPARACIÓN ==========
    mse_ratio = my_mse / sklearn_mse if sklearn_mse > 0 else float('inf')

    print(f"\n📊 RESULTADOS:")
    print(f"  Tu MSE:     {my_mse:.4f}")
    print(f"  sklearn MSE: {sklearn_mse:.4f}")
```

```
    print(f"  Ratio:       {mse_ratio:.2f}x")

    print("\n" + "-" * 60)
    if mse_ratio < 1.1:  # Dentro del 10%
        print("✓ PASSED: Tu implementación es correcta")
        return True
    else:
        print("✗ FAILED: Tu MSE es significativamente mayor")
        return False


# ============================================================
# EJEMPLO DE USO
# ============================================================

if __name__ == "__main__":
    from sklearn.datasets import make_classification, make_regression
    from sklearn.model_selection import train_test_split

    # Dataset de clasificación
    X_clf, y_clf = make_classification(
        n_samples=1000, n_features=10, n_classes=2, random_state=42
    )
    X_train_c, X_test_c, y_train_c, y_test_c = train_test_split(
        X_clf, y_clf, test_size=0.2, random_state=42
    )

    # Dataset de regresión
    X_reg, y_reg = make_regression(
        n_samples=1000, n_features=10, noise=10, random_state=42
    )
    X_train_r, X_test_r, y_train_r, y_test_r = train_test_split(
        X_reg, y_reg, test_size=0.2, random_state=42
    )

    # Ejecutar Shadow Mode
    shadow_mode_logistic_regression(X_train_c, y_train_c, X_test_c, y_test_c)
    print("\n")
    shadow_mode_linear_regression(X_train_r, y_train_r, X_test_r, y_test_r)
```

Checklist Shadow Mode

Día	Algoritmo	Validar
Viernes Sem 10	Linear Regression	MSE ≈ sklearn
Viernes Sem 11	Logistic Regression	Accuracy ≈ sklearn
Viernes Sem 12	Métricas	Precision/Recall = sklearn

✅ Checklist de Finalización (v3.3)

Conocimiento

- [ ] Implementé regresión lineal con Normal Equation y GD
- [ ] Entiendo MSE y su gradiente
- [ ] Implementé regresión logística desde cero
- [ ] Entiendo sigmoid y binary cross-entropy
- [ ] Puedo calcular TP, TN, FP, FN de una matriz de confusión
- [ ] Implementé accuracy, precision, recall, F 1
- [ ] Implementé train/test split
- [ ] Implementé K-fold cross validation
- [ ] Entiendo regularización L 1 vs L 2

Shadow Mode (v3.3 - Obligatorio)

- [ ] **Linear Regression**: Mi MSE ≈ sklearn (ratio < 1.1)
- [ ] **Logistic Regression**: Mi Accuracy ≈ sklearn (diff < 5%)

Entregables de Código

- [ ] `logistic_regression.py` con tests pasando

- [ ] `mypy src/` pasa sin errores
- [ ] `pytest tests/` pasa sin errores

Derivación Analítica (Obligatorio)

- [ ] Derivé el gradiente de Cross-Entropy a mano
- [ ] Documento con derivación completa (Markdown o LaTeX)
- [ ] Puedo explicar por qué $\nabla L = X^T(\hat{y} - y)$

Metodología Feynman

- [ ] Puedo explicar sigmoid en 5 líneas sin jerga
- [ ] Puedo explicar Cross-Entropy vs MSE en 5 líneas
- [ ] Puedo explicar One-vs-All en 5 líneas

🔗 Navegación

Anterior	Índice	Siguiente
04_PROBABILIDAD_ML	00_INDICE	06_UNSUPERVISED_LEARNING

MÓDULO 06 - UNSUPERVISED LEARNING

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 06 - Unsupervised Learning

> 🎯 **Objetivo:** Dominar K-Means clustering y PCA para reducción dimensional
> **Fase:** 2 - Núcleo de ML | **Semanas** 13-16
> **Curso del Pathway:** Unsupervised Algorithms in Machine Learning

🧠 ¿Qué es Unsupervised Learning?

```
APRENDIZAJE NO SUPERVISADO

Tenemos:
- Datos de entrada X (features)
- NO tenemos etiquetas Y

Objetivo: Encontrar estructura oculta en los datos

Tipos principales:
├── CLUSTERING: Agrupar puntos similares
│   └── K-Means, DBSCAN, Hierarchical
├── REDUCCIÓN DIMENSIONAL: Comprimir features
│   └── PCA, t-SNE, UMAP
└── DETECCIÓN DE ANOMALÍAS: Encontrar outliers
    └── Isolation Forest, GMM
```

📚 Contenido del Módulo

Semana	Tema	Entregable
13	K-Means Clustering	`kmeans.py`
14	Evaluación de Clusters	Métricas de clustering
15	PCA	`pca.py`
16	PCA Aplicado + GMM	Compresión de imágenes

💻 Parte 1: K-Means Clustering

1.1 Algoritmo de Lloyd

```
import numpy as np

"""
K-MEANS CLUSTERING (Algoritmo de Lloyd)

Objetivo: Particionar n puntos en k clusters, minimizando la
varianza intra-cluster (inercia).

Algoritmo:
1. Inicializar k centroides (aleatorio o k-means++)
2. Repetir hasta convergencia:
   a. ASIGNAR: cada punto al centroide más cercano
   b. ACTUALIZAR: mover cada centroide al promedio de sus puntos
3. Retornar centroides y asignaciones

Función objetivo (minimizar):
    J = Σᵢ Σⱼ ||xⱼ - μᵢ||²

Donde xⱼ pertenece al cluster i con centroide μᵢ
"""

def euclidean_distance(a: np.ndarray, b: np.ndarray) -> float:
    """Distancia euclidiana entre dos puntos."""
    return np.sqrt(np.sum((a - b) ** 2))

def assign_clusters(X: np.ndarray, centroids: np.ndarray) -> np.ndarray:
    """
    Asigna cada punto al centroide más cercano.

    Args:
```

```
        X: datos (n_samples, n_features)
        centroids: centroides actuales (k, n_features)

    Returns:
        labels: índice del cluster para cada punto (n_samples,)
    """
    n_samples = X.shape[0]
    k = centroids.shape[0]

    # Calcular distancia de cada punto a cada centroide
    distances = np.zeros((n_samples, k))
    for i in range(k):
        distances[:, i] = np.sqrt(np.sum((X - centroids[i]) ** 2, axis=1))

    # Asignar al más cercano
    return np.argmin(distances, axis=1)

def update_centroids(X: np.ndarray, labels: np.ndarray, k: int) -> np.ndarray:
    """
    Actualiza centroides como el promedio de los puntos asignados.

    Args:
        X: datos
        labels: asignaciones actuales
        k: número de clusters

    Returns:
        nuevos centroides
    """
    n_features = X.shape[1]
    centroids = np.zeros((k, n_features))

    for i in range(k):
        points_in_cluster = X[labels == i]
        if len(points_in_cluster) > 0:
            centroids[i] = np.mean(points_in_cluster, axis=0)

    return centroids
```

1.2 K-Means++ Initialization

```
import numpy as np

def kmeans_plus_plus_init(X: np.ndarray, k: int, random_state: int = None) -> np.ndarray:
    """
    Inicialización K-Means++.

    Mejor que inicialización aleatoria porque:
    - Elige centroides que están lejos entre sí
    - Reduce la probabilidad de mala convergencia
    - Garantiza O(log k) de la solución óptima

    Algoritmo:
    1. Elegir primer centroide aleatoriamente
    2. Para cada centroide restante:
       a. Calcular distancia de cada punto al centroide más cercano
       b. Elegir nuevo centroide con probabilidad proporcional a d²
    """
    if random_state is not None:
        np.random.seed(random_state)

    n_samples, n_features = X.shape
    centroids = np.zeros((k, n_features))

    # Primer centroide aleatorio
    first_idx = np.random.randint(n_samples)
    centroids[0] = X[first_idx]

    # Centroides restantes
    for c in range(1, k):
        # Calcular distancia al centroide más cercano para cada punto
        distances = np.zeros(n_samples)
        for i in range(n_samples):
            min_dist = float('inf')
```

```
            for j in range(c):
                dist = np.sum((X[i] - centroids[j]) ** 2)
                min_dist = min(min_dist, dist)
            distances[i] = min_dist

        # Probabilidad proporcional a d²
        probabilities = distances / np.sum(distances)

        # Elegir nuevo centroide
        new_idx = np.random.choice(n_samples, p=probabilities)
        centroids[c] = X[new_idx]

    return centroids
```

1.3 Implementación Completa

```
import numpy as np
from typing import Tuple

class KMeans:
    """K-Means Clustering implementado desde cero."""

    def __init__(
        self,
        n_clusters: int = 3,
        max_iter: int = 300,
        tol: float = 1e-4,
        init: str = 'kmeans++',
        random_state: int = None
    ):
        """
        Args:
            n_clusters: número de clusters (k)
            max_iter: máximo de iteraciones
            tol: tolerancia para convergencia
            init: 'kmeans++' o 'random'
            random_state: semilla para reproducibilidad
        """
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
        self.init = init
        self.random_state = random_state

        self.centroids = None
        self.labels_ = None
        self.inertia_ = None
        self.n_iter_ = 0

    def _init_centroids(self, X: np.ndarray) -> np.ndarray:
        """Inicializa centroides."""
        if self.random_state is not None:
            np.random.seed(self.random_state)

        if self.init == 'kmeans++':
            return kmeans_plus_plus_init(X, self.n_clusters, self.random_state)
        else:
            # Inicialización aleatoria
            indices = np.random.choice(len(X), self.n_clusters, replace=False)
            return X[indices].copy()

    def _compute_inertia(self, X: np.ndarray) -> float:
        """
        Calcula inercia (within-cluster sum of squares).

        Inercia = Σᵢ Σⱼ ||xⱼ - μᵢ||²
        """
        inertia = 0
        for i in range(self.n_clusters):
            cluster_points = X[self.labels_ == i]
            if len(cluster_points) > 0:
                inertia += np.sum((cluster_points - self.centroids[i]) ** 2)
        return inertia
```

```
    def fit(self, X: np.ndarray) -> 'KMeans':
        """Entrena el modelo."""
        # Inicializar centroides
        self.centroids = self._init_centroids(X)

        for iteration in range(self.max_iter):
            # Guardar centroides anteriores
            old_centroids = self.centroids.copy()

            # Paso 1: Asignar puntos a clusters
            self.labels_ = assign_clusters(X, self.centroids)

            # Paso 2: Actualizar centroides
            self.centroids = update_centroids(X, self.labels_, self.n_clusters)

            # Verificar convergencia
            centroid_shift = np.sum((self.centroids - old_centroids) ** 2)
            if centroid_shift < self.tol:
                break

        self.n_iter_ = iteration + 1
        self.inertia_ = self._compute_inertia(X)

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predice clusters para nuevos datos."""
        return assign_clusters(X, self.centroids)

    def fit_predict(self, X: np.ndarray) -> np.ndarray:
        """Entrena y predice."""
        self.fit(X)
        return self.labels_


# Demo
np.random.seed(42)

# Generar datos sintéticos (3 clusters)
cluster1 = np.random.randn(100, 2) + [0, 0]
cluster2 = np.random.randn(100, 2) + [5, 5]
cluster3 = np.random.randn(100, 2) + [10, 0]
X = np.vstack([cluster1, cluster2, cluster3])

# Entrenar
kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X)

print(f"Iteraciones: {kmeans.n_iter_}")
print(f"Inercia: {kmeans.inertia_:.2f}")
print(f"Centroides:\n{kmeans.centroids}")
```

💻 Parte 2: Evaluación de Clusters

2.1 Inercia (Within-Cluster Sum of Squares)

```
def compute_inertia(X: np.ndarray, labels: np.ndarray, centroids: np.ndarray) -> float:
    """
    Inercia: suma de distancias cuadradas al centroide.

    Menor inercia = clusters más compactos.

    Problema: siempre disminuye al aumentar k.
    Solución: usar método del codo.
    """
    inertia = 0
    for i, centroid in enumerate(centroids):
        cluster_points = X[labels == i]
        inertia += np.sum((cluster_points - centroid) ** 2)
    return inertia
```

2.2 Método del Codo (Elbow Method)

```
import numpy as np
import matplotlib.pyplot as plt

def elbow_method(X: np.ndarray, k_range: range) -> list:
    """
    Método del codo para elegir k óptimo.

    Busca el punto donde añadir más clusters
    no reduce significativamente la inercia.
    """
    inertias = []

    for k in k_range:
        kmeans = KMeans(n_clusters=k, random_state=42)
        kmeans.fit(X)
        inertias.append(kmeans.inertia_)

    return inertias

def plot_elbow(k_range: range, inertias: list):
    """Visualiza el método del codo."""
    plt.figure(figsize=(8, 5))
    plt.plot(list(k_range), inertias, 'bo-')
    plt.xlabel('Número de clusters (k)')
    plt.ylabel('Inercia')
    plt.title('Método del Codo')
    plt.grid(True)
    plt.show()

# Demo
# inertias = elbow_method(X, range(1, 11))
# plot_elbow(range(1, 11), inertias)
```

2.3 Silhouette Score

```
import numpy as np

def silhouette_sample(X: np.ndarray, labels: np.ndarray, idx: int) -> float:
    """
    Calcula silhouette para un solo punto.

    s(i) = (b(i) - a(i)) / max(a(i), b(i))

    Donde:
    - a(i): distancia promedio a puntos del mismo cluster
    - b(i): distancia promedio mínima a puntos de otro cluster

    Rango: [-1, 1]
    - 1: punto bien asignado
    - 0: punto en frontera entre clusters
    - -1: punto mal asignado
    """
    point = X[idx]
    label = labels[idx]

    # a(i): distancia promedio intra-cluster
    same_cluster = X[labels == label]
    if len(same_cluster) > 1:
        a = np.mean([np.sqrt(np.sum((point - p) ** 2))
                     for p in same_cluster if not np.array_equal(p, point)])
    else:
        a = 0

    # b(i): distancia promedio al cluster más cercano
    unique_labels = np.unique(labels)
    b = float('inf')
    for other_label in unique_labels:
        if other_label != label:
            other_cluster = X[labels == other_label]
            if len(other_cluster) > 0:
                avg_dist = np.mean([np.sqrt(np.sum((point - p) ** 2))
                                    for p in other_cluster])
```

```
                b = min(b, avg_dist)

    if b == float('inf'):
        return 0

    return (b - a) / max(a, b)

def silhouette_score(X: np.ndarray, labels: np.ndarray) -> float:
    """
    Silhouette Score promedio para todos los puntos.

    Mayor es mejor (max = 1).
    """
    scores = [silhouette_sample(X, labels, i) for i in range(len(X))]
    return np.mean(scores)


# Demo
# score = silhouette_score(X, labels)
# print(f"Silhouette Score: {score:.4f}")
```

💻 Parte 3: PCA (Principal Component Analysis)

3.1 Concepto

```
"""
PCA - ANÁLISIS DE COMPONENTES PRINCIPALES

Objetivo: Reducir dimensionalidad preservando la máxima varianza.

Idea:
1. Centrar los datos (restar media)
2. Encontrar direcciones de máxima varianza (eigenvectors)
3. Proyectar datos en las top-k direcciones

Matemáticamente:
- Las componentes principales son los eigenvectors de la matriz de covarianza
- Los eigenvalues indican cuánta varianza captura cada componente

Aplicaciones:
- Visualización (reducir a 2D/3D)
- Preprocesamiento (eliminar ruido, reducir features)
- Compresión de datos/imágenes
"""
```

3.2 PCA via Eigendecomposition

```
import numpy as np

def pca_eigen(X: np.ndarray, n_components: int) -> tuple:
    """
    PCA usando eigendecomposition de la matriz de covarianza.

    Pasos:
    1. Centrar datos: X_centered = X - mean(X)
    2. Calcular matriz de covarianza: Σ = (1/(n-1)) X^T X
    3. Eigendecomposition: Σv = λv
    4. Ordenar eigenvectors por eigenvalue descendente
    5. Proyectar: X_pca = X_centered @ V[:, :k]

    Returns:
        X_pca: datos transformados
        components: eigenvectors (componentes principales)
        explained_variance_ratio: proporción de varianza por componente
    """
    # 1. Centrar
    mean = np.mean(X, axis=0)
    X_centered = X - mean

    # 2. Matriz de covarianza
    n_samples = X.shape[0]
    cov_matrix = (X_centered.T @ X_centered) / (n_samples - 1)

    # 3. Eigendecomposition
```

```
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Convertir a reales (puede haber componentes imaginarias pequeñas)
    eigenvalues = eigenvalues.real
    eigenvectors = eigenvectors.real

    # 4. Ordenar por eigenvalue descendente
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # 5. Seleccionar top k componentes
    components = eigenvectors[:, :n_components]

    # 6. Proyectar
    X_pca = X_centered @ components

    # 7. Varianza explicada
    total_variance = np.sum(eigenvalues)
    explained_variance_ratio = eigenvalues[:n_components] / total_variance

    return X_pca, components, explained_variance_ratio, mean
```

3.3 PCA via SVD (Más Estable)

```
import numpy as np

def pca_svd(X: np.ndarray, n_components: int) -> tuple:
    """
    PCA usando SVD (Singular Value Decomposition).

    Más estable numéricamente que eigendecomposition.

    Si X = UΣV^T, entonces:
    - V contiene las componentes principales
    - Σ²/(n-1) son los eigenvalues (varianzas)
    """
    # 1. Centrar
    mean = np.mean(X, axis=0)
    X_centered = X - mean

    # 2. SVD
    U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

    # 3. Componentes principales (filas de Vt, o columnas de V)
    components = Vt[:n_components].T

    # 4. Proyectar
    X_pca = X_centered @ components

    # 5. Varianza explicada
    n_samples = X.shape[0]
    variance = (S ** 2) / (n_samples - 1)
    explained_variance_ratio = variance[:n_components] / np.sum(variance)

    return X_pca, components, explained_variance_ratio, mean
```

3.4 Implementación Completa

```
import numpy as np

class PCA:
    """Principal Component Analysis implementado desde cero."""

    def __init__(self, n_components: int = 2):
        """
        Args:
            n_components: número de componentes a retener
        """
        self.n_components = n_components
        self.components_ = None  # (n_features, n_components)
        self.explained_variance_ratio_ = None
        self.mean_ = None
```

```
    def fit(self, X: np.ndarray) -> 'PCA':
        """Calcula componentes principales."""
        # Centrar
        self.mean_ = np.mean(X, axis=0)
        X_centered = X - self.mean_

        # SVD
        U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

        # Componentes principales
        self.components_ = Vt[:self.n_components].T

        # Varianza explicada
        n_samples = X.shape[0]
        variance = (S ** 2) / (n_samples - 1)
        self.explained_variance_ratio_ = variance[:self.n_components] / np.sum(variance)
        self.singular_values_ = S[:self.n_components]

        return self

    def transform(self, X: np.ndarray) -> np.ndarray:
        """Proyecta datos a espacio de componentes principales."""
        X_centered = X - self.mean_
        return X_centered @ self.components_

    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        """Fit y transform en un paso."""
        self.fit(X)
        return self.transform(X)

    def inverse_transform(self, X_pca: np.ndarray) -> np.ndarray:
        """
        Reconstruye datos desde el espacio PCA.

        X_reconstructed = X_pca @ components.T + mean

        Nota: hay pérdida de información si n_components < n_features
        """
        return X_pca @ self.components_.T + self.mean_

    def get_covariance(self) -> np.ndarray:
        """Retorna matriz de covarianza aproximada."""
        return self.components_ @ np.diag(self.singular_values_ ** 2) @ self.components_.T


# Demo
np.random.seed(42)

# Datos correlacionados en 3D
n_samples = 200
X = np.random.randn(n_samples, 3)
X[:, 1] = X[:, 0] * 2 + np.random.randn(n_samples) * 0.1  # y correlacionado con x
X[:, 2] = X[:, 0] + X[:, 1] + np.random.randn(n_samples) * 0.1

# PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print(f"Shape original: {X.shape}")
print(f"Shape reducido: {X_pca.shape}")
print(f"Varianza explicada: {pca.explained_variance_ratio_}")
print(f"Varianza total: {np.sum(pca.explained_variance_ratio_):.2%}")
```

3.5 Reconstrucción y Error

```
import numpy as np

def reconstruction_error(X: np.ndarray, pca: PCA) -> float:
    """
    Calcula el error de reconstrucción.

    Error = ||X - X_reconstructed||² / ||X||²
    """
```

```
    X_pca = pca.transform(X)
    X_reconstructed = pca.inverse_transform(X_pca)

    error = np.sum((X - X_reconstructed) ** 2)
    total = np.sum((X - np.mean(X, axis=0)) ** 2)

    return error / total

def choose_n_components(X: np.ndarray, variance_threshold: float = 0.95) -> int:
    """
    Elige número de componentes para retener cierta varianza.

    Args:
        variance_threshold: proporción de varianza a retener (ej: 0.95 = 95%)
    """
    # PCA con todos los componentes
    pca = PCA(n_components=min(X.shape))
    pca.fit(X)

    # Varianza acumulada
    cumulative_variance = np.cumsum(pca.explained_variance_ratio_)

    # Encontrar n_components
    n_components = np.argmax(cumulative_variance >= variance_threshold) + 1

    return n_components, cumulative_variance
```

💻 Parte 4: Aplicaciones de PCA

4.1 Compresión de Imágenes

```
import numpy as np

def compress_image_pca(image: np.ndarray, n_components: int) -> tuple:
    """
    Comprime una imagen usando PCA.

    Args:
        image: imagen grayscale (height, width)
        n_components: número de componentes a retener

    Returns:
        imagen comprimida, pca model
    """
    # Tratar filas como muestras
    pca = PCA(n_components=n_components)
    image_pca = pca.fit_transform(image)

    # Reconstruir
    image_reconstructed = pca.inverse_transform(image_pca)

    return image_reconstructed, pca

def compression_ratio_pca(original_shape: tuple, n_components: int) -> float:
    """Calcula ratio de compresión."""
    height, width = original_shape
    original_size = height * width
    # Almacenamos: componentes + proyecciones + media
    compressed_size = n_components * width + height * n_components + width
    return compressed_size / original_size
```

4.2 Visualización en 2D

```
import numpy as np
import matplotlib.pyplot as plt

def visualize_pca_2d(X: np.ndarray, labels: np.ndarray = None, title: str = "PCA"):
    """Reduce a 2D y visualiza."""
    pca = PCA(n_components=2)
    X_2d = pca.fit_transform(X)

    plt.figure(figsize=(10, 6))

    if labels is not None:
```

```
        for label in np.unique(labels):
            mask = labels == label
            plt.scatter(X_2d[mask, 0], X_2d[mask, 1],
                        label=f'Clase {label}', alpha=0.7)
        plt.legend()
    else:
        plt.scatter(X_2d[:, 0], X_2d[:, 1], alpha=0.7)

    plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} var)')
    plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} var)')
    plt.title(title)
    plt.grid(True, alpha=0.3)
    plt.show()
```

📦 Entregable del Módulo

`unsupervised_learning.py`

```
"""
Unsupervised Learning Module

Implementación desde cero de:
- K-Means Clustering (con K-Means++ initialization)
- PCA (Principal Component Analysis)
- Métricas de evaluación de clusters

Autor: [Tu nombre]
Módulo: 05 - Unsupervised Learning
"""

import numpy as np
from typing import Tuple, List


# ============================================================
# K-MEANS CLUSTERING
# ============================================================

def kmeans_plus_plus(X: np.ndarray, k: int, seed: int = None) -> np.ndarray:
    """Inicialización K-Means++."""
    if seed: np.random.seed(seed)
    n = len(X)
    centroids = [X[np.random.randint(n)]]

    for _ in range(1, k):
        distances = np.array([min(np.sum((x - c)**2) for c in centroids) for x in X])
        probs = distances / distances.sum()
        centroids.append(X[np.random.choice(n, p=probs)])

    return np.array(centroids)


class KMeans:
    def __init__(self, n_clusters=3, max_iter=300, tol=1e-4, seed=None):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
        self.seed = seed
        self.centroids = None
        self.labels_ = None
        self.inertia_ = None
        self.n_iter_ = 0

    def fit(self, X: np.ndarray) -> 'KMeans':
        self.centroids = kmeans_plus_plus(X, self.n_clusters, self.seed)

        for i in range(self.max_iter):
            old_centroids = self.centroids.copy()

            # Asignar
            distances = np.array([[np.sum((x - c)**2) for c in self.centroids] for x in X])
            self.labels_ = np.argmin(distances, axis=1)

            # Actualizar
```

```
            for j in range(self.n_clusters):
                points = X[self.labels_ == j]
                if len(points) > 0:
                    self.centroids[j] = points.mean(axis=0)

            if np.sum((self.centroids - old_centroids)**2) < self.tol:
                break

        self.n_iter_ = i + 1
        self.inertia_ = sum(np.sum((X[self.labels_ == j] - self.centroids[j])**2)
                            for j in range(self.n_clusters))
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        distances = np.array([[np.sum((x - c)**2) for c in self.centroids] for x in X])
        return np.argmin(distances, axis=1)

    def fit_predict(self, X: np.ndarray) -> np.ndarray:
        self.fit(X)
        return self.labels_


# ============================================================
# PCA
# ============================================================

class PCA:
    def __init__(self, n_components: int = 2):
        self.n_components = n_components
        self.components_ = None
        self.explained_variance_ratio_ = None
        self.mean_ = None

    def fit(self, X: np.ndarray) -> 'PCA':
        self.mean_ = X.mean(axis=0)
        X_centered = X - self.mean_

        U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

        self.components_ = Vt[:self.n_components].T
        variance = (S**2) / (len(X) - 1)
        self.explained_variance_ratio_ = variance[:self.n_components] / variance.sum()

        return self

    def transform(self, X: np.ndarray) -> np.ndarray:
        return (X - self.mean_) @ self.components_

    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        self.fit(X)
        return self.transform(X)

    def inverse_transform(self, X_pca: np.ndarray) -> np.ndarray:
        return X_pca @ self.components_.T + self.mean_


# ============================================================
# MÉTRICAS
# ============================================================

def inertia(X: np.ndarray, labels: np.ndarray, centroids: np.ndarray) -> float:
    """Within-cluster sum of squares."""
    return sum(np.sum((X[labels == i] - centroids[i])**2)
               for i in range(len(centroids)))

def silhouette_score(X: np.ndarray, labels: np.ndarray) -> float:
    """Silhouette score promedio."""
    n = len(X)
    scores = []

    for i in range(n):
        # a: distancia promedio intra-cluster
        same = X[labels == labels[i]]
        a = np.mean([np.sqrt(np.sum((X[i] - x)**2)) for x in same if not np.array_equal(x, X[i])])
```

```
        # b: distancia promedio al cluster más cercano
        b = float('inf')
        for label in np.unique(labels):
            if label != labels[i]:
                other = X[labels == label]
                if len(other) > 0:
                    b = min(b, np.mean([np.sqrt(np.sum((X[i] - x)**2)) for x in other]))

        if b == float('inf'):
            scores.append(0)
        else:
            scores.append((b - a) / max(a, b))

    return np.mean(scores)


# ============================================================
# TESTS
# ============================================================

if __name__ == "__main__":
    np.random.seed(42)

    # Test K-Means
    c1 = np.random.randn(50, 2) + [0, 0]
    c2 = np.random.randn(50, 2) + [5, 5]
    c3 = np.random.randn(50, 2) + [10, 0]
    X = np.vstack([c1, c2, c3])

    kmeans = KMeans(n_clusters=3, seed=42)
    labels = kmeans.fit_predict(X)

    print(f"K-Means Inertia: {kmeans.inertia_:.2f}")
    print(f"Silhouette Score: {silhouette_score(X, labels):.4f}")

    # Test PCA
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)
    X_reconstructed = pca.inverse_transform(X_pca)

    print(f"\nPCA Varianza explicada: {pca.explained_variance_ratio_}")
    print(f"Error reconstrucción: {np.mean((X - X_reconstructed)**2):.6f}")

    print("\n✓ Todos los tests pasaron!")
```

✅ Checklist de Finalización

- [ ] Implementé K-Means con inicialización K-Means++
- [ ] Entiendo el algoritmo de Lloyd (asignar-actualizar)
- [ ] Puedo calcular inercia y usarla para el método del codo
- [ ] Implementé silhouette score
- [ ] Implementé PCA usando SVD
- [ ] Entiendo varianza explicada y puedo elegir n_components
- [ ] Puedo reconstruir datos desde PCA
- [ ] Apliqué PCA para visualización 2D
- [ ] Todos los tests del módulo pasan

🔗 Navegación

Anterior	Índice	Siguiente
04_SUPERVISED_LEARNING	00_INDICE	06_DEEP_LEARNING

MÓDULO 07 - DEEP LEARNING + CNNS

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 07 - Deep Learning

> 🎯 **Objetivo:** Implementar MLP con backprop + entender fundamentos de CNNs
> **Fase:** 2 - Núcleo de ML | **Semanas** 17-20
> **Curso del Pathway:** Introduction to Deep Learning

🧠 ¿Por Qué Deep Learning?

```
DEEP LEARNING = Redes Neuronales Multicapa + Arquitecturas Especializadas

Ventajas sobre ML clásico:
├── Aprende features automáticamente (no feature engineering manual)
├── Puede modelar relaciones NO LINEALES complejas
├── Escala con más datos y más compute
└── Estado del arte en visión (CNNs), NLP (Transformers), etc.

Desventajas:
├── Requiere más datos
├── "Caja negra" - menos interpretable
└── Costoso computacionalmente
```

📚 Contenido del Módulo

Semana	Tema	Entregable
17	Perceptrón y MLP	`activations.py` + forward pass
18	Backpropagation	`backward()` con Chain Rule
19	**CNNs: Teoría**	Entender convolución, pooling, stride
20	Optimizadores y Entrenamiento	`neural_network.py` completo

💻 Parte 1: Perceptrón y Activaciones

1.1 La Neurona Artificial

```python
import numpy as np

"""
NEURONA ARTIFICIAL (Perceptrón)

Inspiración biológica:
- Recibe señales de entrada (dendrites)
- Procesa y decide si "dispara" (soma)
- Envía señal de salida (axon)

Modelo matemático:
    z = Σ wᵢxᵢ + b = w·x + b  (combinación lineal)
    a = σ(z)                   (activación)

Donde:
- x: vector de entradas
- w: vector de pesos (learnable)
- b: bias (learnable)
- σ: función de activación (introduce no-linealidad)
"""

def perceptron(x: np.ndarray, w: np.ndarray, b: float) -> float:
    """
    Un perceptrón simple.

    Args:
        x: entrada (n_features,)
        w: pesos (n_features,)
        b: bias

    Returns:
        salida activada
    """
```

```
    z = np.dot(w, x) + b
    return 1 if z > 0 else 0  # Función escalón
```

1.2 Funciones de Activación

```
import numpy as np

class Activations:
    """Funciones de activación y sus derivadas."""

    @staticmethod
    def sigmoid(z: np.ndarray) -> np.ndarray:
        """
        Sigmoid: σ(z) = 1 / (1 + e^(-z))

        Rango: (0, 1)
        Uso: Capa de salida para clasificación binaria
        Problema: Vanishing gradient para |z| grande
        """
        z = np.clip(z, -500, 500)
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def sigmoid_derivative(a: np.ndarray) -> np.ndarray:
        """σ'(z) = σ(z) · (1 - σ(z)) = a · (1 - a)"""
        return a * (1 - a)

    @staticmethod
    def relu(z: np.ndarray) -> np.ndarray:
        """
        ReLU: f(z) = max(0, z)

        Rango: [0, ∞)
        Uso: Capas ocultas (default moderno)
        Ventaja: No vanishing gradient para z > 0
        Problema: "Dying ReLU" si z < 0 siempre
        """
        return np.maximum(0, z)

    @staticmethod
    def relu_derivative(z: np.ndarray) -> np.ndarray:
        """ReLU'(z) = 1 si z > 0, 0 si z ≤ 0"""
        return (z > 0).astype(float)

    @staticmethod
    def tanh(z: np.ndarray) -> np.ndarray:
        """
        Tanh: f(z) = (e^z - e^(-z)) / (e^z + e^(-z))

        Rango: (-1, 1)
        Uso: Alternativa a sigmoid (centrado en 0)
        """
        return np.tanh(z)

    @staticmethod
    def tanh_derivative(a: np.ndarray) -> np.ndarray:
        """tanh'(z) = 1 - tanh²(z) = 1 - a²"""
        return 1 - a ** 2

    @staticmethod
    def softmax(z: np.ndarray) -> np.ndarray:
        """
        Softmax: softmax(z)ᵢ = e^(zᵢ) / Σⱼ e^(zⱼ)

        Rango: (0, 1), suma = 1
        Uso: Capa de salida para clasificación multiclase
        Output: probabilidades de cada clase
        """
        # Restar máximo para estabilidad numérica
        z_shifted = z - np.max(z, axis=-1, keepdims=True)
        exp_z = np.exp(z_shifted)
        return exp_z / np.sum(exp_z, axis=-1, keepdims=True)
```

```
# Demo
z = np.array([-2, -1, 0, 1, 2])
act = Activations()

print("z:", z)
print("sigmoid:", act.sigmoid(z))
print("relu:", act.relu(z))
print("tanh:", act.tanh(z))
print("softmax:", act.softmax(z))
```

1.3 El Problema XOR

```
"""
XOR: La limitación del Perceptrón Simple

XOR truth table:
    x1  x2  |  y
    0   0   |  0
    0   1   |  1
    1   0   |  1
    1   1   |  0

Un perceptrón simple NO puede resolver XOR porque:
- XOR no es linealmente separable
- No existe una línea que separe las clases

Solución: Red multicapa (MLP)
- Una capa oculta puede aprender features intermedias
- Combinación de features no lineales resuelve XOR
"""

# Datos XOR
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

# Un perceptrón simple no puede aprender esto
# Necesitamos una red con al menos una capa oculta
```

💻 Parte 2: Forward Propagation

2.1 Arquitectura MLP

```
"""
MLP - Multilayer Perceptron

Arquitectura típica:
    Input Layer → Hidden Layer(s) → Output Layer

Ejemplo para clasificación binaria:
    x (n_features) → h (n_hidden) → y (1)

Forward Pass:
    z₁ = W₁x + b₁        (capa 1: lineal)
    a₁ = σ(z₁)           (capa 1: activación)
    z₂ = W₂a₁ + b₂       (capa 2: lineal)
    a₂ = σ(z₂)           (capa 2: activación = output)

Dimensiones:
    x: (n_features,)
    W₁: (n_hidden, n_features)
    b₁: (n_hidden,)
    z₁, a₁: (n_hidden,)
    W₂: (n_output, n_hidden)
    b₂: (n_output,)
    z₂, a₂: (n_output,)
"""
```

2.2 Implementación Forward Pass

```
import numpy as np
from typing import List, Dict

class Layer:
```

```
    """Una capa de la red neuronal."""

    def __init__(self, input_size: int, output_size: int, activation: str = 'relu'):
        """
        Args:
            input_size: número de entradas
            output_size: número de neuronas
            activation: 'relu', 'sigmoid', 'tanh', 'softmax', 'linear'
        """
        self.input_size = input_size
        self.output_size = output_size
        self.activation = activation

        # Inicialización Xavier/He
        if activation == 'relu':
            # He initialization para ReLU
            std = np.sqrt(2.0 / input_size)
        else:
            # Xavier initialization
            std = np.sqrt(1.0 / input_size)

        self.W = np.random.randn(output_size, input_size) * std
        self.b = np.zeros(output_size)

        # Cache para backprop
        self.cache = {}

    def forward(self, x: np.ndarray) -> np.ndarray:
        """
        Forward pass de una capa.

        z = Wx + b
        a = activation(z)
        """
        self.cache['x'] = x

        # Transformación lineal
        z = self.W @ x + self.b
        self.cache['z'] = z

        # Activación
        if self.activation == 'relu':
            a = np.maximum(0, z)
        elif self.activation == 'sigmoid':
            a = 1 / (1 + np.exp(-np.clip(z, -500, 500)))
        elif self.activation == 'tanh':
            a = np.tanh(z)
        elif self.activation == 'softmax':
            z_shifted = z - np.max(z)
            exp_z = np.exp(z_shifted)
            a = exp_z / np.sum(exp_z)
        else:  # linear
            a = z

        self.cache['a'] = a
        return a


class NeuralNetwork:
    """Red Neuronal Multicapa."""

    def __init__(self, layer_sizes: List[int], activations: List[str]):
        """
        Args:
            layer_sizes: [input_size, hidden1, hidden2, ..., output_size]
            activations: ['relu', 'relu', ..., 'sigmoid'] para cada capa
        """
        assert len(activations) == len(layer_sizes) - 1

        self.layers = []
        for i in range(len(layer_sizes) - 1):
            layer = Layer(layer_sizes[i], layer_sizes[i+1], activations[i])
            self.layers.append(layer)

    def forward(self, x: np.ndarray) -> np.ndarray:
```

```
        """Forward pass a través de todas las capas."""
        a = x
        for layer in self.layers:
            a = layer.forward(a)
        return a

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predicción para múltiples muestras."""
        predictions = []
        for x in X:
            output = self.forward(x)
            if len(output) == 1:
                predictions.append(1 if output[0] > 0.5 else 0)
            else:
                predictions.append(np.argmax(output))
        return np.array(predictions)


# Demo
net = NeuralNetwork(
    layer_sizes=[2, 4, 1],  # 2 inputs → 4 hidden → 1 output
    activations=['relu', 'sigmoid']
)

# Forward pass
x = np.array([0.5, 0.3])
output = net.forward(x)
print(f"Input: {x}")
print(f"Output: {output}")
```

💻 Parte 3: Backpropagation

3.1 Funciones de Pérdida

```
import numpy as np

def binary_cross_entropy(y_true: float, y_pred: float, eps: float = 1e-15) -> float:
    """
    Binary Cross-Entropy Loss.

    L = -[y·log(ŷ) + (1-y)·log(1-ŷ)]

    Args:
        y_true: etiqueta real (0 o 1)
        y_pred: predicción (probabilidad)
    """
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

def bce_derivative(y_true: float, y_pred: float, eps: float = 1e-15) -> float:
    """
    Derivada de BCE respecto a y_pred.

    ∂L/∂ŷ = -y/ŷ + (1-y)/(1-ŷ)
    """
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -y_true / y_pred + (1 - y_true) / (1 - y_pred)

def categorical_cross_entropy(y_true: np.ndarray, y_pred: np.ndarray, eps: float = 1e-15) -> float:
    """
    Categorical Cross-Entropy para multiclase.

    L = -Σᵢ yᵢ·log(ŷᵢ)

    Args:
        y_true: one-hot encoded (k,)
        y_pred: probabilidades softmax (k,)
    """
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -np.sum(y_true * np.log(y_pred))

def mse_loss(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """Mean Squared Error."""
    return np.mean((y_true - y_pred) ** 2)
```

3.2 Backpropagation: La Chain Rule en Acción

```
"""
BACKPROPAGATION

Objetivo: Calcular ∂L/∂W y ∂L/∂b para cada capa.

Usando Chain Rule:
    ∂L/∂W₂ = ∂L/∂a₂ · ∂a₂/∂z₂ · ∂z₂/∂W₂
    ∂L/∂W₁ = ∂L/∂a₂ · ∂a₂/∂z₂ · ∂z₂/∂a₁ · ∂a₁/∂z₁ · ∂z₁/∂W₁

Patrón:
1. Calcular ∂L/∂a_output (derivada de la loss)
2. Para cada capa, de atrás hacia adelante:
   a. δ = ∂L/∂z = ∂L/∂a · ∂a/∂z (error de la capa)
   b. ∂L/∂W = δ · x^T
   c. ∂L/∂b = δ
   d. Propagar: ∂L/∂a_prev = W^T · δ
"""

def backward_layer(layer, dL_da: np.ndarray) -> tuple:
    """
    Backward pass de una capa.

    Args:
        layer: capa con cache del forward pass
        dL_da: gradiente de la loss respecto a la activación

    Returns:
        dL_dx: gradiente respecto a la entrada
        dL_dW: gradiente respecto a los pesos
        dL_db: gradiente respecto al bias
    """
    z = layer.cache['z']
    x = layer.cache['x']
    a = layer.cache['a']

    # Derivada de la activación: ∂a/∂z
    if layer.activation == 'sigmoid':
        da_dz = a * (1 - a)
    elif layer.activation == 'relu':
        da_dz = (z > 0).astype(float)
    elif layer.activation == 'tanh':
        da_dz = 1 - a ** 2
    elif layer.activation == 'softmax':
        # Para softmax + cross-entropy, usamos el gradiente simplificado
        da_dz = np.ones_like(z)  # se maneja especialmente
    else:  # linear
        da_dz = np.ones_like(z)

    # δ = ∂L/∂z = ∂L/∂a · ∂a/∂z
    delta = dL_da * da_dz

    # Gradientes
    dL_dW = np.outer(delta, x)
    dL_db = delta
    dL_dx = layer.W.T @ delta

    return dL_dx, dL_dW, dL_db
```

3.3 Red Neuronal Completa con Backprop

```
import numpy as np
from typing import List, Tuple

class NeuralNetworkFull:
    """Red Neuronal con Backpropagation completo."""

    def __init__(self, layer_sizes: List[int], activations: List[str]):
        self.layers = []
        for i in range(len(layer_sizes) - 1):
            layer = Layer(layer_sizes[i], layer_sizes[i+1], activations[i])
            self.layers.append(layer)
```

```
        self.loss_history = []

    def forward(self, x: np.ndarray) -> np.ndarray:
        a = x
        for layer in self.layers:
            a = layer.forward(a)
        return a

    def backward(self, y_true: np.ndarray) -> List[Tuple[np.ndarray, np.ndarray]]:
        """
        Backward pass: calcula gradientes para todas las capas.

        Returns:
            Lista de (dW, db) para cada capa
        """
        gradients = []

        # Obtener predicción (última activación)
        y_pred = self.layers[-1].cache['a']

        # Gradiente inicial: ∂L/∂a_output
        # Para sigmoid + BCE: simplificado a (y_pred - y_true)
        # Para softmax + CCE: también (y_pred - y_true)
        if self.layers[-1].activation in ['sigmoid', 'softmax']:
            dL_da = y_pred - y_true
        else:
            # MSE: 2(y_pred - y_true)
            dL_da = 2 * (y_pred - y_true)

        # Propagar hacia atrás
        for layer in reversed(self.layers):
            dL_dx, dL_dW, dL_db = backward_layer(layer, dL_da)
            gradients.insert(0, (dL_dW, dL_db))
            dL_da = dL_dx

        return gradients

    def update_weights(self, gradients: List[Tuple], learning_rate: float):
        """Actualiza pesos usando gradient descent."""
        for layer, (dW, db) in zip(self.layers, gradients):
            layer.W -= learning_rate * dW
            layer.b -= learning_rate * db

    def fit(
        self,
        X: np.ndarray,
        y: np.ndarray,
        epochs: int = 1000,
        learning_rate: float = 0.1,
        verbose: bool = True
    ):
        """Entrena la red."""
        for epoch in range(epochs):
            total_loss = 0

            for xi, yi in zip(X, y):
                # Forward
                output = self.forward(xi)

                # Loss
                if isinstance(yi, (int, float)):
                    yi_arr = np.array([yi])
                else:
                    yi_arr = yi
                loss = binary_cross_entropy(yi_arr[0], output[0])
                total_loss += loss

                # Backward
                gradients = self.backward(yi_arr)

                # Update
                self.update_weights(gradients, learning_rate)

            avg_loss = total_loss / len(X)
            self.loss_history.append(avg_loss)
```

```
            if verbose and epoch % 100 == 0:
                print(f"Epoch {epoch}: Loss = {avg_loss:.4f}")

    def predict(self, X: np.ndarray) -> np.ndarray:
        predictions = []
        for x in X:
            output = self.forward(x)
            predictions.append(1 if output[0] > 0.5 else 0)
        return np.array(predictions)


# Demo: Resolver XOR
print("=== Entrenando para XOR ===")
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

net = NeuralNetworkFull(
    layer_sizes=[2, 4, 1],
    activations=['tanh', 'sigmoid']
)

net.fit(X_xor, y_xor, epochs=5000, learning_rate=0.5, verbose=True)

print("\n=== Predicciones XOR ===")
for x, y in zip(X_xor, y_xor):
    pred = net.forward(x)[0]
    print(f"{x} -> {pred:.4f} (target: {y})")
```

💻 Parte 4: Optimizadores

4.1 SGD (Stochastic Gradient Descent)

```
class SGD:
    """Vanilla Stochastic Gradient Descent."""

    def __init__(self, learning_rate: float = 0.01):
        self.lr = learning_rate

    def update(self, layer, dW: np.ndarray, db: np.ndarray):
        layer.W -= self.lr * dW
        layer.b -= self.lr * db
```

4.2 SGD con Momentum

```
class SGDMomentum:
    """
    SGD con Momentum.

    v_t = β·v_{t-1} + (1-β)·∇L
    θ = θ - lr·v_t

    Momentum ayuda a:
    - Acelerar convergencia
    - Escapar de mínimos locales
    - Reducir oscilaciones
    """

    def __init__(self, learning_rate: float = 0.01, momentum: float = 0.9):
        self.lr = learning_rate
        self.momentum = momentum
        self.velocities = {}

    def update(self, layer, dW: np.ndarray, db: np.ndarray, layer_id: int):
        if layer_id not in self.velocities:
            self.velocities[layer_id] = {
                'W': np.zeros_like(dW),
                'b': np.zeros_like(db)
            }

        v = self.velocities[layer_id]

        # Actualizar velocidad
        v['W'] = self.momentum * v['W'] + (1 - self.momentum) * dW
```

```
        v['b'] = self.momentum * v['b'] + (1 - self.momentum) * db

        # Actualizar parámetros
        layer.W -= self.lr * v['W']
        layer.b -= self.lr * v['b']
```

4.3 Adam Optimizer

```
class Adam:
    """
    Adam: Adaptive Moment Estimation.

    Combina:
    - Momentum (primer momento)
    - RMSprop (segundo momento)

    m_t = β₁·m_{t-1} + (1-β₁)·g_t       (momentum)
    v_t = β₂·v_{t-1} + (1-β₂)·g_t²      (velocidad adaptativa)
    m̂_t = m_t / (1 - β₁^t)              (corrección de bias)
    v̂_t = v_t / (1 - β₂^t)
    θ = θ - lr · m̂_t / (√v̂_t + ε)
    """

    def __init__(
        self,
        learning_rate: float = 0.001,
        beta1: float = 0.9,
        beta2: float = 0.999,
        epsilon: float = 1e-8
    ):
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}
        self.v = {}
        self.t = 0

    def update(self, layer, dW: np.ndarray, db: np.ndarray, layer_id: int):
        if layer_id not in self.m:
            self.m[layer_id] = {'W': np.zeros_like(dW), 'b': np.zeros_like(db)}
            self.v[layer_id] = {'W': np.zeros_like(dW), 'b': np.zeros_like(db)}

        self.t += 1
        m, v = self.m[layer_id], self.v[layer_id]

        # Actualizar momentos
        m['W'] = self.beta1 * m['W'] + (1 - self.beta1) * dW
        m['b'] = self.beta1 * m['b'] + (1 - self.beta1) * db
        v['W'] = self.beta2 * v['W'] + (1 - self.beta2) * dW**2
        v['b'] = self.beta2 * v['b'] + (1 - self.beta2) * db**2

        # Corrección de bias
        m_hat_W = m['W'] / (1 - self.beta1**self.t)
        m_hat_b = m['b'] / (1 - self.beta1**self.t)
        v_hat_W = v['W'] / (1 - self.beta2**self.t)
        v_hat_b = v['b'] / (1 - self.beta2**self.t)

        # Actualizar parámetros
        layer.W -= self.lr * m_hat_W / (np.sqrt(v_hat_W) + self.epsilon)
        layer.b -= self.lr * m_hat_b / (np.sqrt(v_hat_b) + self.epsilon)
```

📦 Entregable del Módulo

`neural_network.py`

```
"""
Neural Network Module

Implementación desde cero de:
- MLP (Multilayer Perceptron)
- Backpropagation
- Optimizadores (SGD, Momentum, Adam)
- Funciones de activación
```

```
Autor: [Tu nombre]
Módulo: 06 - Deep Learning
"""

import numpy as np
from typing import List, Tuple, Optional


# ============================================================
# ACTIVACIONES
# ============================================================

def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

def sigmoid_deriv(a):
    return a * (1 - a)

def relu(z):
    return np.maximum(0, z)

def relu_deriv(z):
    return (z > 0).astype(float)

def tanh_deriv(a):
    return 1 - a**2

def softmax(z):
    exp_z = np.exp(z - np.max(z))
    return exp_z / np.sum(exp_z)


# ============================================================
# CAPA
# ============================================================

class Layer:
    def __init__(self, input_size: int, output_size: int, activation: str = 'relu'):
        self.activation = activation
        scale = np.sqrt(2.0 / input_size) if activation == 'relu' else np.sqrt(1.0 / input_size)
        self.W = np.random.randn(output_size, input_size) * scale
        self.b = np.zeros(output_size)
        self.cache = {}

    def forward(self, x: np.ndarray) -> np.ndarray:
        self.cache['x'] = x
        z = self.W @ x + self.b
        self.cache['z'] = z

        if self.activation == 'relu':
            a = relu(z)
        elif self.activation == 'sigmoid':
            a = sigmoid(z)
        elif self.activation == 'tanh':
            a = np.tanh(z)
        elif self.activation == 'softmax':
            a = softmax(z)
        else:
            a = z

        self.cache['a'] = a
        return a

    def backward(self, dL_da: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        z, x, a = self.cache['z'], self.cache['x'], self.cache['a']

        if self.activation == 'sigmoid':
            da_dz = sigmoid_deriv(a)
        elif self.activation == 'relu':
            da_dz = relu_deriv(z)
        elif self.activation == 'tanh':
            da_dz = tanh_deriv(a)
        else:
            da_dz = np.ones_like(z)
```

```
        delta = dL_da * da_dz
        dL_dW = np.outer(delta, x)
        dL_db = delta
        dL_dx = self.W.T @ delta

        return dL_dx, dL_dW, dL_db


# ============================================================
# OPTIMIZADORES
# ============================================================

class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def step(self, layers, gradients):
        for layer, (dW, db) in zip(layers, gradients):
            layer.W -= self.lr * dW
            layer.b -= self.lr * db


class Adam:
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999, eps=1e-8):
        self.lr, self.beta1, self.beta2, self.eps = lr, beta1, beta2, eps
        self.m, self.v, self.t = {}, {}, 0

    def step(self, layers, gradients):
        self.t += 1
        for i, (layer, (dW, db)) in enumerate(zip(layers, gradients)):
            if i not in self.m:
                self.m[i] = {'W': np.zeros_like(dW), 'b': np.zeros_like(db)}
                self.v[i] = {'W': np.zeros_like(dW), 'b': np.zeros_like(db)}

            self.m[i]['W'] = self.beta1 * self.m[i]['W'] + (1 - self.beta1) * dW
            self.m[i]['b'] = self.beta1 * self.m[i]['b'] + (1 - self.beta1) * db
            self.v[i]['W'] = self.beta2 * self.v[i]['W'] + (1 - self.beta2) * dW**2
            self.v[i]['b'] = self.beta2 * self.v[i]['b'] + (1 - self.beta2) * db**2

            m_hat_W = self.m[i]['W'] / (1 - self.beta1**self.t)
            m_hat_b = self.m[i]['b'] / (1 - self.beta1**self.t)
            v_hat_W = self.v[i]['W'] / (1 - self.beta2**self.t)
            v_hat_b = self.v[i]['b'] / (1 - self.beta2**self.t)

            layer.W -= self.lr * m_hat_W / (np.sqrt(v_hat_W) + self.eps)
            layer.b -= self.lr * m_hat_b / (np.sqrt(v_hat_b) + self.eps)


# ============================================================
# RED NEURONAL
# ============================================================

class NeuralNetwork:
    def __init__(self, layer_sizes: List[int], activations: List[str]):
        self.layers = [Layer(layer_sizes[i], layer_sizes[i+1], activations[i])
                       for i in range(len(layer_sizes)-1)]
        self.loss_history = []

    def forward(self, x: np.ndarray) -> np.ndarray:
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, y_true: np.ndarray) -> List[Tuple]:
        y_pred = self.layers[-1].cache['a']
        dL_da = y_pred - y_true

        gradients = []
        for layer in reversed(self.layers):
            dL_da, dW, db = layer.backward(dL_da)
            gradients.insert(0, (dW, db))
        return gradients

    def fit(self, X, y, epochs=1000, lr=0.1, optimizer='sgd', verbose=True):
```

```
        opt = Adam(lr) if optimizer == 'adam' else SGD(lr)

        for epoch in range(epochs):
            total_loss = 0
            for xi, yi in zip(X, y):
                yi_arr = np.atleast_1d(yi)
                output = self.forward(xi)

                # BCE loss
                output_clip = np.clip(output, 1e-15, 1-1e-15)
                loss = -np.sum(yi_arr * np.log(output_clip) + (1-yi_arr) * np.log(1-output_clip))
                total_loss += loss

                gradients = self.backward(yi_arr)
                opt.step(self.layers, gradients)

            self.loss_history.append(total_loss / len(X))
            if verbose and epoch % (epochs//10) == 0:
                print(f"Epoch {epoch}: Loss = {self.loss_history[-1]:.4f}")

    def predict(self, X: np.ndarray) -> np.ndarray:
        return np.array([1 if self.forward(x)[0] > 0.5 else 0 for x in X])

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        return np.mean(self.predict(X) == y)


# ============================================================
# TESTS
# ============================================================

if __name__ == "__main__":
    print("=== Test: XOR Problem ===")
    X = np.array([[0,0], [0,1], [1,0], [1,1]])
    y = np.array([0, 1, 1, 0])

    net = NeuralNetwork([2, 4, 1], ['tanh', 'sigmoid'])
    net.fit(X, y, epochs=5000, lr=0.5, verbose=True)

    print("\nPredicciones:")
    for xi, yi in zip(X, y):
        pred = net.forward(xi)[0]
        print(f"{xi} -> {pred:.4f} (target: {yi})")

    print(f"\nAccuracy: {net.score(X, y):.2%}")
    print("\n✓ Test XOR completado!")
```

💻 Parte 5: CNNs - Redes Convolucionales (Semana 19)

> ⚠️ **Nota:** En este módulo NO implementamos CNNs desde cero (es complejo). El objetivo es **entender la teoría** para el curso de Deep Learning de CU Boulder.

5.1 ¿Por Qué CNNs para Imágenes?

```
PROBLEMA CON MLP PARA IMÁGENES:

Imagen MNIST: 28x28 = 784 píxeles
MLP fully connected a capa de 256 neuronas:
  → 784 × 256 = 200,704 parámetros (¡solo primera capa!)

Imagen HD: 1920x1080x3 = 6,220,800 píxeles
  → Imposible conectar todo con todo

SOLUCIÓN: CONVOLUCIÓN
- Procesar regiones locales (no toda la imagen)
- Compartir pesos (el mismo filtro en toda la imagen)
- Detectar patrones sin importar su posición
```

5.2 La Operación de Convolución

```
import numpy as np

def convolve2d_simple(image: np.ndarray, kernel: np.ndarray) -> np.ndarray:
```

```
    """
    Convolución 2D simplificada (para entender el concepto).

    La convolución desliza un kernel (filtro) sobre la imagen
    y calcula el producto punto en cada posición.

    Args:
        image: Imagen de entrada (H, W)
        kernel: Filtro (kH, kW), típicamente 3x3 o 5x5

    Returns:
        Feature map (H-kH+1, W-kW+1)
    """
    H, W = image.shape
    kH, kW = kernel.shape

    # Tamaño del output (sin padding)
    out_H = H - kH + 1
    out_W = W - kW + 1

    output = np.zeros((out_H, out_W))

    for i in range(out_H):
        for j in range(out_W):
            # Extraer región de la imagen
            region = image[i:i+kH, j:j+kW]
            # Producto punto con el kernel
            output[i, j] = np.sum(region * kernel)

    return output


# Ejemplo: Detección de bordes verticales
image = np.array([
    [0, 0, 0, 1, 1, 1],
    [0, 0, 0, 1, 1, 1],
    [0, 0, 0, 1, 1, 1],
    [0, 0, 0, 1, 1, 1],
])

# Kernel Sobel para bordes verticales
sobel_vertical = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
])

edges = convolve2d_simple(image, sobel_vertical)
print("Feature map (bordes verticales):")
print(edges)
```

5.3 Conceptos Clave de CNNs

```
┌─────────────────────────────────────────────────────────────┐
│  VOCABULARIO CNN                                            │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  KERNEL (FILTRO)                                            │
│  ├── Matriz pequeña (3x3, 5x5) que detecta patrones         │
│  ├── Los valores del kernel son APRENDIDOS (backprop)       │
│  └── Diferentes kernels detectan diferentes features        │
│                                                             │
│  STRIDE                                                     │
│  ├── Cuántos píxeles se mueve el kernel en cada paso        │
│  ├── stride=1: mueve 1 píxel (output grande)                │
│  └── stride=2: mueve 2 píxeles (output más pequeño)         │
│                                                             │
│  PADDING                                                    │
│  ├── Añadir ceros alrededor de la imagen                    │
│  ├── 'valid': sin padding (output más pequeño)              │
│  └── 'same': padding para mantener tamaño                   │
│                                                             │
│  POOLING                                                    │
│  ├── Reduce dimensiones (downsampling)                      │
```

```
|  ├── Max Pooling: toma el máximo de cada región                  |
|  └── Average Pooling: toma el promedio                           |
|                                                                  |
|  FEATURE MAP                                                     |
|  └── Output de aplicar un filtro (lo que "ve" el filtro)         |
|                                                                  |
└──────────────────────────────────────────────────────────────────┘
```

5.4 Cálculo de Dimensiones (Importante para Exámenes)

```python
def output_size(input_size: int, kernel_size: int,
                stride: int = 1, padding: int = 0) -> int:
    """
    Fórmula para calcular tamaño del output de convolución.

    output_size = floor((input + 2*padding - kernel) / stride) + 1
    """
    return (input_size + 2 * padding - kernel_size) // stride + 1


# Ejemplos típicos de examen:
print("=== Ejercicios de dimensiones ===")

# Ejemplo 1: MNIST sin padding
# Input: 28x28, Kernel: 5x5, Stride: 1, Padding: 0
out = output_size(28, 5, stride=1, padding=0)
print(f"MNIST 28x28, kernel 5x5, stride 1: output = {out}x{out}")  # 24x24

# Ejemplo 2: Con padding 'same'
# Para mantener tamaño con kernel 3x3, necesitas padding=1
out = output_size(28, 3, stride=1, padding=1)
print(f"MNIST 28x28, kernel 3x3, padding 1: output = {out}x{out}")  # 28x28

# Ejemplo 3: Max Pooling 2x2 stride 2
out = output_size(24, 2, stride=2, padding=0)
print(f"24x24, pooling 2x2 stride 2: output = {out}x{out}")  # 12x12
```

5.5 Arquitectura Típica de CNN

```
┌──────────────────────────────────────────────────────────────────┐
|  ARQUITECTURA LENET-5 (Clásica para MNIST)                       |
├──────────────────────────────────────────────────────────────────┤
|                                                                  |
|  Input: 28x28x1 (imagen grayscale)                               |
|         |                                                        |
|         ▼                                                        |
|  [CONV 5x5, 6 filtros] → 24x24x6                                 |
|         |                                                        |
|         ▼                                                        |
|  [ReLU]                                                          |
|         |                                                        |
|         ▼                                                        |
|  [MaxPool 2x2] → 12x12x6                                         |
|         |                                                        |
|         ▼                                                        |
|  [CONV 5x5, 16 filtros] → 8x8x16                                 |
|         |                                                        |
|         ▼                                                        |
|  [ReLU]                                                          |
|         |                                                        |
|         ▼                                                        |
|  [MaxPool 2x2] → 4x4x16 = 256 neuronas                           |
|         |                                                        |
|         ▼                                                        |
|  [Flatten] → 256                                                 |
|         |                                                        |
|         ▼                                                        |
|  [FC 120] → 120                                                  |
|         |                                                        |
|         ▼                                                        |
|  [FC 84] → 84                                                    |
|         |                                                        |
|         ▼                                                        |
```

```
│  [FC 10 + Softmax] → 10 clases (dígitos 0-9)                    │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

5.6 Max Pooling

```
def max_pool2d(x: np.ndarray, pool_size: int = 2) -> np.ndarray:
    """
    Max Pooling 2D.

    Reduce dimensiones tomando el máximo de cada región.
    Hace la red más robusta a pequeñas traslaciones.

    Args:
        x: Feature map (H, W)
        pool_size: Tamaño de la ventana (típicamente 2)

    Returns:
        Pooled output (H//pool_size, W//pool_size)
    """
    H, W = x.shape
    out_H, out_W = H // pool_size, W // pool_size

    output = np.zeros((out_H, out_W))

    for i in range(out_H):
        for j in range(out_W):
            region = x[i*pool_size:(i+1)*pool_size,
                       j*pool_size:(j+1)*pool_size]
            output[i, j] = np.max(region)

    return output


# Ejemplo
feature_map = np.array([
    [1, 3, 2, 4],
    [5, 6, 1, 2],
    [3, 2, 1, 0],
    [1, 2, 3, 4]
])

pooled = max_pool2d(feature_map, pool_size=2)
print("Original 4x4:")
print(feature_map)
print("\nMax Pooled 2x2:")
print(pooled)  # [[6, 4], [3, 4]]
```

5.7 Por Qué Funcionan las CNNs

```
INTUICIÓN:

1. CAPAS INICIALES: Detectan features simples
   - Bordes horizontales, verticales, diagonales
   - Cambios de color, texturas

2. CAPAS MEDIAS: Combinan features simples
   - Esquinas, curvas, patrones

3. CAPAS PROFUNDAS: Features de alto nivel
   - Partes de objetos (ojos, ruedas, letras)

4. CAPAS FINALES: Objetos completos
   - "Esto es un 7", "Esto es un gato"

VENTAJAS CLAVE:
├── Parameter sharing: mismo filtro en toda la imagen
├── Sparse connectivity: cada output depende de región local
├── Translation invariance: detecta patrones sin importar posición
└── Hierarchical features: de simple a complejo
```

5.8 Recursos para Profundizar en CNNs

Recurso	Descripción
3 B 1 B - But what is a convolution?	Intuición visual
CS 2 3 1 n Stanford	Curso completo de CNNs
Deep Learning Book, Cap. 9	Teoría formal

📝 Derivación Analítica: Backpropagation a Mano (v3.2)

> 🎓 **Simulación de Examen:** "Derive las ecuaciones de backpropagation para una red de 2 capas". Este es un clásico de exámenes de posgrado.

Red de 2 Capas: Derivación Completa

Arquitectura:

- Input: x (vector de features)
- Capa 1 : $z_1 = W_1 x + b_1$, $a_1 = \sigma(z_1)$
- Capa 2 : $z_2 = W_2 a_1 + b_2$, $\hat{y} = \sigma(z_2)$
- Loss: $L = -[y \log(\hat{y}) + (1 -y) \log(1 -\hat{y})]$

Paso 1: Gradiente de la Capa de Salida

$$\frac{\partial L}{\partial z_2} = \hat{y} - y = \delta_2 $$

(Resultado elegante gracias a la combinación sigmoid + cross-entropy)

$$\frac{\partial L}{\partial W_2} = \delta_2 \cdot a_1 ^T$$

$$\frac{\partial L}{\partial b_2} = \delta_2 $$

Paso 2: Propagar el Error Hacia Atrás (Capa Oculta)

$$\frac{\partial L}{\partial a_1} = W_2 ^T \delta_2 $$

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1} \odot \sigma'(z_1) = W_2 ^T \delta_2 \odot a_1 \odot (1 - a_1) = \delta_1 $$

$$\frac{\partial L}{\partial W_1} = \delta_1 \cdot x^T$$

$$\frac{\partial L}{\partial b_1} = \delta_1 $$

Resumen: Las 4 Ecuaciones de Backprop

```
┌─────────────────────────────────────────────────────┐
│ ECUACIONES DE BACKPROPAGATION                       │
│                                                     │
│ 1. δ_L = ∇_a L ⊙ σ'(z_L)     Error en capa final    │
│                                                     │
│ 2. δ_l = (W_{l+1}^T δ_{l+1}) ⊙ σ'(z_l)   Propagar atrás │
│                                                     │
│ 3. ∂L/∂W_l = δ_l · a_{l-1}^T   Gradiente de pesos   │
│                                                     │
│ 4. ∂L/∂b_l = δ_l              Gradiente de bias     │
└─────────────────────────────────────────────────────┘
```

Tu Entregable

Escribe en un documento (Markdown o LaTeX):

1. Derivación completa de backprop para red de 2 capas
2. Por qué $\delta_L = \hat{y} - y$ cuando usamos sigmoid + cross-entropy
3. Diagrama de grafo computacional mostrando el flujo de gradientes

🧪 Overfit on Small Batch: Debugging de Redes Neuronales (v3.3)

> ⚠️ **CRÍTICO:** Esta es la técnica # 1 de debugging en Deep Learning. Si tu red no puede hacer overfitting en 1 0 ejemplos, tiene un bug.

El Principio

```
REGLA DE ORO DEL DEBUGGING EN DL:

Una red neuronal DEBE poder memorizar un dataset pequeño.
```

```
Si entrenas con:
- 10 ejemplos
- Muchas épocas (1000+)
- Sin regularización

El loss DEBE llegar a ~0.00 (o muy cercano).

Si NO llega a 0 → TU IMPLEMENTACIÓN TIENE UN BUG
```

Por Qué Funciona

```
┌─────────────────────────────────────────────────────┐
│ OVERFIT TEST                                        │
│                                                     │
│ Dataset pequeño (10 ejemplos):                      │
│ - Capacidad de la red >> complejidad del dataset    │
│ - La red puede "memorizar" cada ejemplo perfectamente │
│ - Loss debe → 0 si backprop funciona                │
│                                                     │
│ Si loss NO baja:                                    │
│ - Gradiente mal calculado                           │
│ - Learning rate incorrecto                          │
│ - Arquitectura rota (dimensiones)                   │
│ - Bug en forward o backward pass                    │
└─────────────────────────────────────────────────────┘
```

Script: `overfit_test.py` (Entregable Obligatorio v3.3)

```python
"""
Overfit Test - Validación de Redes Neuronales
Si tu red no puede hacer overfit en 10 ejemplos, está rota.

Autor: [Tu nombre]
Módulo: 07 - Deep Learning
"""
import numpy as np
from typing import List, Tuple


def overfit_test(
    model,
    X_small: np.ndarray,
    y_small: np.ndarray,
    epochs: int = 2000,
    target_loss: float = 0.01,
    verbose: bool = True
) -> Tuple[bool, List[float]]:
    """
    Test de overfitting: la red debe memorizar un dataset pequeño.

    Args:
        model: Tu red neuronal (debe tener .fit() y .forward())
        X_small: Dataset pequeño (10-20 ejemplos)
        y_small: Labels del dataset
        epochs: Épocas de entrenamiento
        target_loss: Loss objetivo (default: 0.01)
        verbose: Mostrar progreso

    Returns:
        (passed, loss_history)
    """
    if verbose:
        print("=" * 60)
        print("OVERFIT TEST: ¿Puede tu red memorizar 10 ejemplos?")
        print("=" * 60)
        print(f"Dataset size: {len(y_small)}")
        print(f"Epochs: {epochs}")
        print(f"Target loss: {target_loss}")
        print("-" * 60)

    # Entrenar
    loss_history = []
```

```
    for epoch in range(epochs):
        # Forward pass para todos los ejemplos
        total_loss = 0.0
        for i in range(len(y_small)):
            output = model.forward(X_small[i])
            loss = np.mean((output - y_small[i]) ** 2)  # MSE
            total_loss += loss

            # Backward y update (asumiendo que model tiene estos métodos)
            model.backward(y_small[i])
            model.update(learning_rate=0.1)

        avg_loss = total_loss / len(y_small)
        loss_history.append(avg_loss)

        if verbose and epoch % 500 == 0:
            print(f"Epoch {epoch:4d}: Loss = {avg_loss:.6f}")

    final_loss = loss_history[-1]
    passed = final_loss < target_loss

    if verbose:
        print("-" * 60)
        print(f"Final Loss: {final_loss:.6f}")
        if passed:
            print("✓ PASSED: Tu red puede hacer overfitting")
            print("  → El forward y backward pass funcionan correctamente")
        else:
            print("✗ FAILED: Tu red NO puede hacer overfitting")
            print("  → Revisa tu implementación de backprop")
            print("  Posibles causas:")
            print("  - Gradiente mal calculado")
            print("  - Learning rate muy bajo")
            print("  - Bug en forward pass")
            print("  - Dimensiones incorrectas")

    return passed, loss_history


# ============================================================
# EJEMPLO: Test con XOR (debe pasar)
# ============================================================

def test_xor_overfit():
    """Test: Una red pequeña debe resolver XOR perfectamente."""
    print("\n" + "=" * 60)
    print("TEST: Overfit on XOR Problem")
    print("=" * 60)

    # XOR dataset (4 ejemplos)
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ], dtype=np.float64)

    y = np.array([
        [0],
        [1],
        [1],
        [0]
    ], dtype=np.float64)

    # Crear red simple (2 -> 8 -> 1)
    # NOTA: Reemplaza esto con tu clase NeuralNetwork
    class SimpleNet:
        def __init__(self):
            np.random.seed(42)
            self.W1 = np.random.randn(8, 2) * 0.5
            self.b1 = np.zeros((8, 1))
            self.W2 = np.random.randn(1, 8) * 0.5
            self.b2 = np.zeros((1, 1))

            # Cache para backprop
```

```
            self.cache = {}

        def sigmoid(self, z):
            return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

        def forward(self, x):
            x = x.reshape(-1, 1)
            z1 = self.W1 @ x + self.b1
            a1 = self.sigmoid(z1)
            z2 = self.W2 @ a1 + self.b2
            a2 = self.sigmoid(z2)

            self.cache = {'x': x, 'z1': z1, 'a1': a1, 'z2': z2, 'a2': a2}
            return a2.flatten()

        def backward(self, y_true):
            y_true = np.array(y_true).reshape(-1, 1)
            a2 = self.cache['a2']
            a1 = self.cache['a1']
            x = self.cache['x']

            # Gradientes
            dz2 = a2 - y_true
            self.dW2 = dz2 @ a1.T
            self.db2 = dz2

            da1 = self.W2.T @ dz2
            dz1 = da1 * a1 * (1 - a1)
            self.dW1 = dz1 @ x.T
            self.db1 = dz1

        def update(self, learning_rate):
            self.W1 -= learning_rate * self.dW1
            self.b1 -= learning_rate * self.db1
            self.W2 -= learning_rate * self.dW2
            self.b2 -= learning_rate * self.db2

    # Ejecutar test
    model = SimpleNet()
    passed, history = overfit_test(model, X, y, epochs=2000, target_loss=0.01)

    # Verificar predicciones finales
    print("\nPredicciones finales:")
    for i in range(len(X)):
        pred = model.forward(X[i])
        print(f"  Input: {X[i]} → Pred: {pred[0]:.3f} (Target: {y[i][0]})")

    return passed


if __name__ == "__main__":
    test_xor_overfit()
```

Checklist de Debugging con Overfit Test

Síntoma	Diagnóstico	Solución
Loss no baja	Gradiente = `0` o NaN	Verificar derivadas con grad_check
Loss baja muy lento	Learning rate muy bajo	Aumentar LR (probar `0.1`, `0.5`, `1.0`)
Loss oscila mucho	Learning rate muy alto	Reducir LR
Loss sube	Signos invertidos en gradiente	Revisar forward/backward
Loss = NaN	Overflow en exp/softmax	Usar versiones numéricamente estables

🎯 El Reto del Tablero Blanco (Metodología Feynman)

Explica en **máximo** `5` **líneas** sin jerga técnica:

1. **¿Qué es backpropagation?**

Pista: Piensa en "culpar" a cada peso por el error.

2. **¿Por qué ReLU es mejor que sigmoid en capas ocultas?**

   Pista: Piensa en qué pasa con el gradiente de sigmoid cuando z es muy grande o muy pequeño.

3. **¿Qué hace una convolución en una imagen?**

   Pista: Piensa en "deslizar una lupa" buscando un patrón específico.

4. **¿Por qué usamos pooling?**

   Pista: Piensa en "resumir" una región y hacerla más pequeña.

✅ Checklist de Finalización (v3.3)

Conocimiento

- [ ] Entiendo la analogía neurona biológica → neurona artificial
- [ ] Implementé sigmoid, ReLU, tanh, softmax y sus derivadas
- [ ] Entiendo por qué XOR no es linealmente separable
- [ ] Implementé forward pass para MLP
- [ ] Entiendo la Chain Rule aplicada a backpropagation
- [ ] Implementé backward pass calculando gradientes
- [ ] Implementé SGD, SGD+Momentum y Adam
- [ ] Mi red resuelve el problema XOR

CNNs (Teoría)

- [ ] Entiendo qué es convolución, stride, padding y pooling
- [ ] Puedo calcular dimensiones de output de una CNN
- [ ] Conozco la arquitectura LeNet-5

Entregables de Código

- [ ] `neural_network.py` con tests pasando
- [ ] `mypy src/` pasa sin errores
- [ ] `pytest tests/` pasa sin errores

Overfit Test (v3.3 - Obligatorio)

- [ ] **`overfit_test.py` implementado**
- [ ] **Mi red hace overfit en XOR (loss < 0.01)**
- [ ] Si el test falla, debuggeé con grad_check

Derivación Analítica (Obligatorio)

- [ ] Derivé las ecuaciones de backprop a mano
- [ ] Documento con derivación completa (Markdown o LaTeX)
- [ ] Diagrama de grafo computacional

Metodología Feynman

- [ ] Puedo explicar backpropagation en 5 líneas sin jerga
- [ ] Puedo explicar ReLU vs sigmoid en 5 líneas
- [ ] Puedo explicar convolución en 5 líneas
- [ ] Puedo explicar pooling en 5 líneas

🔗 Navegación

Anterior	Índice	Siguiente
06_UNSUPERVISED_LEARNING	00_INDICE	08_PROYECTO_MNIST

MÓDULO 08 - PROYECTO MNIST ANALYST

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

Módulo 08 - Proyecto Final: MNIST Analyst

🎯 **Objetivo:** Pipeline end-to-end que demuestra competencia en las 3 áreas del Pathway
Fase: 3 - Proyecto Integrador | **Semanas** 21-24 (4 semanas)
Dataset: MNIST (dígitos escritos a mano, 28×28 píxeles)

🧠 ¿Qué Estamos Construyendo?

```
┌─────────────────────────────────────────────────────────────┐
│                                                             │
│   PROYECTO: END-TO-END HANDWRITTEN DIGIT ANALYSIS PIPELINE  │
│   ──────────────────────────────────────────────────────    │
│                                                             │
│   LÍNEA 1: MACHINE LEARNING (3 créditos) - DEMOSTRADO EN 4 SEMANAS
│   ├── Semana 21: EDA + PCA + K-Means                        │
│   ├── Semana 22: Logistic Regression One-vs-All             │
│   ├── Semana 23: MLP con Backprop                           │
│   └── Semana 24: Comparación de Modelos + Informe           │
│                                                             │
│   RESULTADO:                                                │
│   Un pipeline que analiza, agrupa y clasifica dígitos MNIST │
│   usando algoritmos implementados 100% desde cero.          │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

💡 **Nota v3.1:** MNIST es un dataset simple. 4 semanas son suficientes para un proyecto bien estructurado.

📚 Estructura del Proyecto

Cronograma (4 Semanas)

Semana	Fase	Materia Demostrada	Entregable
21	EDA + No Supervisado	Unsupervised Algorithms	PCA + K-Means funcionando
22	Clasificación Clásica	Supervised Learning	Logistic Regression OvA
23	Deep Learning	Introduction to Deep Learning	MLP con backprop
24	Benchmark + Informe	Integración	MODEL_COMPARISON.md

Estructura de Archivos

```
mnist-analyst/
├── src/
│   ├── __init__.py
│   ├── data_loader.py         # Cargar y preprocesar MNIST (Módulo 01)
│   ├── linear_algebra.py      # Operaciones vectoriales (Módulo 02)
│   ├── pca.py                 # PCA desde cero (Módulo 06)
│   ├── kmeans.py              # K-Means desde cero (Módulo 06)
│   ├── logistic_regression.py # Logistic multiclase (Módulo 05)
│   ├── neural_network.py      # MLP con backprop (Módulo 07)
│   ├── metrics.py             # Métricas de evaluación (Módulo 05)
│   └── pipeline.py            # Pipeline integrado
│
├── notebooks/
│   ├── 01_data_exploration.ipynb
│   ├── 02_pca_visualization.ipynb
│   ├── 03_kmeans_clustering.ipynb
│   ├── 04_logistic_classification.ipynb
│   └── 05_neural_network_benchmark.ipynb
│
├── tests/
│   └── test_*.py
│
├── docs/
│   └── MODEL_COMPARISON.md
│
```

```
├── README.md
└── requirements.txt
```

💻 Parte 1: Cargar MNIST

1.1 Data Loader

```
"""
MNIST Dataset Loader

MNIST contiene:
- 60,000 imágenes de entrenamiento
- 10,000 imágenes de test
- Cada imagen: 28x28 píxeles grayscale (0-255)
- 10 clases: dígitos 0-9

Formato aplanado: cada imagen es un vector de 784 dimensiones
"""

import numpy as np
import struct
import gzip
from pathlib import Path
from typing import Tuple


def load_mnist_images(filepath: str) -> np.ndarray:
    """
    Carga imágenes MNIST desde archivo IDX.

    Formato IDX:
    - 4 bytes: magic number
    - 4 bytes: número de imágenes
    - 4 bytes: número de filas
    - 4 bytes: número de columnas
    - resto: píxeles (unsigned bytes)
    """
    with gzip.open(filepath, 'rb') as f:
        magic, num_images, rows, cols = struct.unpack('>IIII', f.read(16))
        images = np.frombuffer(f.read(), dtype=np.uint8)
        images = images.reshape(num_images, rows * cols)
    return images


def load_mnist_labels(filepath: str) -> np.ndarray:
    """Carga etiquetas MNIST."""
    with gzip.open(filepath, 'rb') as f:
        magic, num_labels = struct.unpack('>II', f.read(8))
        labels = np.frombuffer(f.read(), dtype=np.uint8)
    return labels


def load_mnist(data_dir: str = 'data/mnist') -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """
    Carga dataset MNIST completo.

    Returns:
        X_train: (60000, 784)
        y_train: (60000,)
        X_test: (10000, 784)
        y_test: (10000,)
    """
    data_dir = Path(data_dir)

    X_train = load_mnist_images(data_dir / 'train-images-idx3-ubyte.gz')
    y_train = load_mnist_labels(data_dir / 'train-labels-idx1-ubyte.gz')
    X_test = load_mnist_images(data_dir / 't10k-images-idx3-ubyte.gz')
    y_test = load_mnist_labels(data_dir / 't10k-labels-idx1-ubyte.gz')

    return X_train, y_train, X_test, y_test


def normalize_data(X: np.ndarray) -> np.ndarray:
    """Normaliza píxeles a rango [0, 1]."""
```

```
    return X.astype(np.float64) / 255.0


def one_hot_encode(y: np.ndarray, num_classes: int = 10) -> np.ndarray:
    """Convierte etiquetas a one-hot encoding."""
    one_hot = np.zeros((len(y), num_classes))
    one_hot[np.arange(len(y)), y] = 1
    return one_hot


# Alternativa: generar datos sintéticos si no tienes MNIST
def generate_synthetic_mnist(n_samples: int = 1000, seed: int = 42) -> Tuple:
    """
    Genera datos sintéticos similares a MNIST para pruebas.
    """
    np.random.seed(seed)

    X = np.random.rand(n_samples, 784)  # Imágenes aleatorias
    y = np.random.randint(0, 10, n_samples)  # Etiquetas aleatorias

    # Split 80/20
    split = int(0.8 * n_samples)
    return X[:split], y[:split], X[split:], y[split:]
```

1.2 Visualización

```
import numpy as np
import matplotlib.pyplot as plt

def visualize_digits(X: np.ndarray, y: np.ndarray, n_samples: int = 25):
    """Visualiza una cuadrícula de dígitos."""
    n_cols = 5
    n_rows = (n_samples + n_cols - 1) // n_cols

    fig, axes = plt.subplots(n_rows, n_cols, figsize=(10, 2*n_rows))
    axes = axes.flatten()

    for i, ax in enumerate(axes):
        if i < n_samples:
            img = X[i].reshape(28, 28)
            ax.imshow(img, cmap='gray')
            ax.set_title(f'Label: {y[i]}')
        ax.axis('off')

    plt.tight_layout()
    plt.show()


def visualize_digit_single(x: np.ndarray, title: str = ''):
    """Visualiza un solo dígito."""
    plt.figure(figsize=(4, 4))
    plt.imshow(x.reshape(28, 28), cmap='gray')
    plt.title(title)
    plt.axis('off')
    plt.show()
```

💻 Parte 2: Exploración No Supervisada (Semanas 21-22)

2.1 PCA para Visualización

```
"""
SEMANA 21: PCA en MNIST

Objetivo: Reducir de 784 dimensiones a 2-3 para visualización.

Preguntas a responder:
1. ¿Cuánta varianza se retiene con pocos componentes?
2. ¿Se separan visualmente las clases en 2D?
3. ¿Qué "aprenden" las componentes principales?
"""

import numpy as np
from typing import Tuple
```

```
class PCA:
    """PCA implementado desde cero (del Módulo 05)."""

    def __init__(self, n_components: int):
        self.n_components = n_components
        self.components_ = None
        self.mean_ = None
        self.explained_variance_ratio_ = None

    def fit(self, X: np.ndarray) -> 'PCA':
        self.mean_ = np.mean(X, axis=0)
        X_centered = X - self.mean_

        # SVD (más estable que eigendecomposition)
        U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

        self.components_ = Vt[:self.n_components].T
        variance = (S ** 2) / (len(X) - 1)
        self.explained_variance_ratio_ = variance[:self.n_components] / np.sum(variance)

        return self

    def transform(self, X: np.ndarray) -> np.ndarray:
        return (X - self.mean_) @ self.components_

    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        self.fit(X)
        return self.transform(X)

    def inverse_transform(self, X_pca: np.ndarray) -> np.ndarray:
        return X_pca @ self.components_.T + self.mean_


def analyze_pca_mnist(X: np.ndarray, y: np.ndarray):
    """Análisis PCA completo de MNIST."""

    # 1. PCA con diferentes números de componentes
    print("=== Análisis de Varianza Explicada ===")
    pca_full = PCA(n_components=min(50, X.shape[1]))
    pca_full.fit(X)

    cumulative_var = np.cumsum(pca_full.explained_variance_ratio_)

    for n in [2, 10, 50]:
        if n <= len(cumulative_var):
            print(f"  {n} componentes: {cumulative_var[n-1]:.2%} varianza")

    # 2. Visualización 2D
    print("\n=== Proyección 2D ===")
    pca_2d = PCA(n_components=2)
    X_2d = pca_2d.fit_transform(X)

    plt.figure(figsize=(10, 8))
    for digit in range(10):
        mask = y == digit
        plt.scatter(X_2d[mask, 0], X_2d[mask, 1],
                    alpha=0.5, label=str(digit), s=10)
    plt.legend()
    plt.xlabel(f'PC1 ({pca_2d.explained_variance_ratio_[0]:.1%})')
    plt.ylabel(f'PC2 ({pca_2d.explained_variance_ratio_[1]:.1%})')
    plt.title('MNIST en 2D (PCA)')
    plt.show()

    # 3. Visualizar componentes principales
    print("\n=== Componentes Principales como Imágenes ===")
    fig, axes = plt.subplots(2, 5, figsize=(12, 5))
    pca_10 = PCA(n_components=10)
    pca_10.fit(X)

    for i, ax in enumerate(axes.flatten()):
        component = pca_10.components_[:, i].reshape(28, 28)
        ax.imshow(component, cmap='RdBu')
        ax.set_title(f'PC{i+1}')
        ax.axis('off')
    plt.suptitle('Top 10 Componentes Principales')
```

```
    plt.tight_layout()
    plt.show()

    return pca_2d, X_2d
```

2.2 K-Means Clustering

```
"""
SEMANA 22: K-Means en MNIST

Objetivo: Agrupar dígitos SIN usar etiquetas.

Preguntas a responder:
1. ¿K-Means encuentra los 10 dígitos?
2. ¿Qué tan puros son los clusters?
3. ¿Cómo se ven los centroides?
"""

import numpy as np

class KMeans:
    """K-Means implementado desde cero (del Módulo 05)."""

    def __init__(self, n_clusters: int = 10, max_iter: int = 100, seed: int = None):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.seed = seed
        self.centroids = None
        self.labels_ = None
        self.inertia_ = None

    def _init_centroids_plusplus(self, X: np.ndarray) -> np.ndarray:
        """K-Means++ initialization."""
        if self.seed:
            np.random.seed(self.seed)

        n_samples = len(X)
        centroids = [X[np.random.randint(n_samples)]]

        for _ in range(1, self.n_clusters):
            distances = np.array([min(np.sum((x - c)**2) for c in centroids) for x in X])
            probs = distances / distances.sum()
            centroids.append(X[np.random.choice(n_samples, p=probs)])

        return np.array(centroids)

    def fit(self, X: np.ndarray) -> 'KMeans':
        self.centroids = self._init_centroids_plusplus(X)

        for _ in range(self.max_iter):
            # Asignar
            distances = np.array([[np.sum((x - c)**2) for c in self.centroids] for x in X])
            self.labels_ = np.argmin(distances, axis=1)

            # Actualizar
            new_centroids = np.array([X[self.labels_ == k].mean(axis=0)
                                      if np.sum(self.labels_ == k) > 0
                                      else self.centroids[k]
                                      for k in range(self.n_clusters)])

            if np.allclose(self.centroids, new_centroids):
                break
            self.centroids = new_centroids

        self.inertia_ = sum(np.sum((X[self.labels_ == k] - self.centroids[k])**2)
                            for k in range(self.n_clusters))
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        distances = np.array([[np.sum((x - c)**2) for c in self.centroids] for x in X])
        return np.argmin(distances, axis=1)


def analyze_kmeans_mnist(X: np.ndarray, y: np.ndarray):
```

```
    """Análisis K-Means de MNIST."""

    print("=== K-Means Clustering ===")
    kmeans = KMeans(n_clusters=10, seed=42)
    kmeans.fit(X)

    # 1. Visualizar centroides
    print("\n=== Centroides (promedio de cada cluster) ===")
    fig, axes = plt.subplots(2, 5, figsize=(12, 5))
    for i, ax in enumerate(axes.flatten()):
        centroid = kmeans.centroids[i].reshape(28, 28)
        ax.imshow(centroid, cmap='gray')
        ax.set_title(f'Cluster {i}')
        ax.axis('off')
    plt.suptitle('Centroides K-Means')
    plt.tight_layout()
    plt.show()

    # 2. Analizar pureza de clusters
    print("\n=== Pureza de Clusters ===")
    print("Cluster | Dígito Dominante | Pureza")
    print("-" * 40)

    total_correct = 0
    for cluster in range(10):
        cluster_mask = kmeans.labels_ == cluster
        cluster_labels = y[cluster_mask]

        if len(cluster_labels) > 0:
            dominant_digit = np.bincount(cluster_labels).argmax()
            purity = np.sum(cluster_labels == dominant_digit) / len(cluster_labels)
            total_correct += np.sum(cluster_labels == dominant_digit)
            print(f"   {cluster}    |        {dominant_digit}         | {purity:.2%}")

    overall_purity = total_correct / len(y)
    print(f"\nPureza Global: {overall_purity:.2%}")

    return kmeans
```

💻 Parte 3: Clasificación Supervisada (Semanas 23-24)

3.1 Logistic Regression One-vs-All

```
"""
SEMANAS 23-24: Logistic Regression Multiclase

Estrategia One-vs-All (OvA):
- Entrenar 10 clasificadores binarios
- Cada uno: "¿Es este dígito X o no?"
- Predicción: elegir la clase con mayor probabilidad
"""

import numpy as np
from typing import List

def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))


class LogisticRegressionBinary:
    """Logistic Regression binario."""

    def __init__(self, lr: float = 0.1, n_iter: int = 100, reg: float = 0.01):
        self.lr = lr
        self.n_iter = n_iter
        self.reg = reg  # L2 regularization
        self.theta = None

    def fit(self, X: np.ndarray, y: np.ndarray) -> 'LogisticRegressionBinary':
        n_samples, n_features = X.shape
        self.theta = np.zeros(n_features)

        for _ in range(self.n_iter):
            h = sigmoid(X @ self.theta)
```

```
            grad = (1/n_samples) * X.T @ (h - y) + (self.reg/n_samples) * self.theta
            self.theta -= self.lr * grad

        return self

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        return sigmoid(X @ self.theta)


class LogisticRegressionOvA:
    """Logistic Regression One-vs-All para clasificación multiclase."""

    def __init__(self, n_classes: int = 10, lr: float = 0.1, n_iter: int = 100):
        self.n_classes = n_classes
        self.lr = lr
        self.n_iter = n_iter
        self.classifiers: List[LogisticRegressionBinary] = []

    def fit(self, X: np.ndarray, y: np.ndarray) -> 'LogisticRegressionOvA':
        """Entrena un clasificador por clase."""
        # Añadir bias
        X_b = np.column_stack([np.ones(len(X)), X])

        self.classifiers = []
        for c in range(self.n_classes):
            print(f"  Entrenando clasificador para clase {c}...", end='\r')
            y_binary = (y == c).astype(int)
            clf = LogisticRegressionBinary(self.lr, self.n_iter)
            clf.fit(X_b, y_binary)
            self.classifiers.append(clf)

        print("  Entrenamiento completado.                ")
        return self

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Retorna probabilidades para cada clase."""
        X_b = np.column_stack([np.ones(len(X)), X])
        probs = np.column_stack([clf.predict_proba(X_b) for clf in self.classifiers])
        return probs

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predice la clase con mayor probabilidad."""
        probs = self.predict_proba(X)
        return np.argmax(probs, axis=1)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        """Accuracy."""
        return np.mean(self.predict(X) == y)


def train_logistic_mnist(X_train, y_train, X_test, y_test):
    """Entrena y evalúa Logistic Regression en MNIST."""

    print("=== Logistic Regression One-vs-All ===")

    # Entrenar
    lr_model = LogisticRegressionOvA(n_classes=10, lr=0.1, n_iter=200)
    lr_model.fit(X_train, y_train)

    # Evaluar
    train_acc = lr_model.score(X_train, y_train)
    test_acc = lr_model.score(X_test, y_test)

    print(f"\nTrain Accuracy: {train_acc:.2%}")
    print(f"Test Accuracy:  {test_acc:.2%}")

    # Métricas detalladas
    y_pred = lr_model.predict(X_test)

    print("\n=== Métricas por Clase ===")
    print("Dígito | Precision | Recall | F1-Score")
    print("-" * 45)

    for digit in range(10):
        tp = np.sum((y_test == digit) & (y_pred == digit))
```

```
        fp = np.sum((y_test != digit) & (y_pred == digit))
        fn = np.sum((y_test == digit) & (y_pred != digit))

        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0 else 0

        print(f"   {digit}   |   {precision:.3f}   |  {recall:.3f}  |   {f1:.3f}")

    # Matriz de confusión
    print("\n=== Matriz de Confusión ===")
    cm = np.zeros((10, 10), dtype=int)
    for true, pred in zip(y_test, y_pred):
        cm[true, pred] += 1

    print("    " + "  ".join(str(i) for i in range(10)))
    for i in range(10):
        print(f"{i}: " + " ".join(f"{cm[i,j]:3d}" for j in range(10)))

    return lr_model
```

💻 Parte 4: Deep Learning (Semanas 25-26)

4.1 MLP para MNIST

```
"""
SEMANAS 25-26: Neural Network para MNIST

Arquitectura:
- Input: 784 (28x28 píxeles aplanados)
- Hidden 1: 128 neuronas, ReLU
- Hidden 2: 64 neuronas, ReLU
- Output: 10 neuronas, Softmax

Objetivo: Superar a Logistic Regression
"""

import numpy as np
from typing import List, Tuple

# Funciones de activación
def relu(z):
    return np.maximum(0, z)

def relu_deriv(z):
    return (z > 0).astype(float)

def softmax(z):
    exp_z = np.exp(z - np.max(z))
    return exp_z / np.sum(exp_z)


class NeuralNetworkMNIST:
    """Red Neuronal optimizada para MNIST."""

    def __init__(self, layer_sizes: List[int] = [784, 128, 64, 10], seed: int = 42):
        """
        Args:
            layer_sizes: [input, hidden1, hidden2, ..., output]
        """
        np.random.seed(seed)

        self.layer_sizes = layer_sizes
        self.n_layers = len(layer_sizes)

        # Inicializar pesos (He initialization para ReLU)
        self.weights = []
        self.biases = []

        for i in range(self.n_layers - 1):
            w = np.random.randn(layer_sizes[i+1], layer_sizes[i]) * np.sqrt(2.0 / layer_sizes[i])
            b = np.zeros(layer_sizes[i+1])
            self.weights.append(w)
            self.biases.append(b)
```

```
        self.cache = {}
        self.loss_history = []

    def forward(self, x: np.ndarray) -> np.ndarray:
        """Forward pass."""
        self.cache['a0'] = x
        a = x

        for i in range(self.n_layers - 2):
            z = self.weights[i] @ a + self.biases[i]
            a = relu(z)
            self.cache[f'z{i+1}'] = z
            self.cache[f'a{i+1}'] = a

        # Última capa: softmax
        z = self.weights[-1] @ a + self.biases[-1]
        a = softmax(z)
        self.cache[f'z{self.n_layers-1}'] = z
        self.cache[f'a{self.n_layers-1}'] = a

        return a

    def backward(self, y_true: np.ndarray) -> Tuple[List, List]:
        """Backward pass."""
        y_pred = self.cache[f'a{self.n_layers-1}']

        # Gradiente de softmax + cross-entropy
        dz = y_pred - y_true

        dW_list = []
        db_list = []

        for i in range(self.n_layers - 2, -1, -1):
            a_prev = self.cache[f'a{i}']

            dW = np.outer(dz, a_prev)
            db = dz

            dW_list.insert(0, dW)
            db_list.insert(0, db)

            if i > 0:
                da_prev = self.weights[i].T @ dz
                z_prev = self.cache[f'z{i}']
                dz = da_prev * relu_deriv(z_prev)

        return dW_list, db_list

    def fit(
        self,
        X: np.ndarray,
        y: np.ndarray,
        epochs: int = 10,
        batch_size: int = 32,
        learning_rate: float = 0.01,
        verbose: bool = True
    ):
        """Entrena la red con mini-batch SGD."""
        n_samples = len(X)

        for epoch in range(epochs):
            # Shuffle
            indices = np.random.permutation(n_samples)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            total_loss = 0

            for i in range(0, n_samples, batch_size):
                X_batch = X_shuffled[i:i+batch_size]
                y_batch = y_shuffled[i:i+batch_size]

                # Acumular gradientes del batch
                dW_accum = [np.zeros_like(w) for w in self.weights]
```

```
                db_accum = [np.zeros_like(b) for b in self.biases]

                for x, y_true_label in zip(X_batch, y_batch):
                    # One-hot encode
                    y_one_hot = np.zeros(10)
                    y_one_hot[y_true_label] = 1

                    # Forward
                    y_pred = self.forward(x)

                    # Loss
                    loss = -np.sum(y_one_hot * np.log(np.clip(y_pred, 1e-15, 1)))
                    total_loss += loss

                    # Backward
                    dW_list, db_list = self.backward(y_one_hot)

                    for j in range(len(self.weights)):
                        dW_accum[j] += dW_list[j]
                        db_accum[j] += db_list[j]

                # Update
                batch_len = len(X_batch)
                for j in range(len(self.weights)):
                    self.weights[j] -= learning_rate * dW_accum[j] / batch_len
                    self.biases[j] -= learning_rate * db_accum[j] / batch_len

            avg_loss = total_loss / n_samples
            self.loss_history.append(avg_loss)

            if verbose:
                train_acc = self.score(X[:1000], y[:1000])
                print(f"Epoch {epoch+1}/{epochs} - Loss: {avg_loss:.4f} - Acc: {train_acc:.2%}")

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predice clases."""
        return np.array([np.argmax(self.forward(x)) for x in X])

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        """Accuracy."""
        return np.mean(self.predict(X) == y)


def train_neural_network_mnist(X_train, y_train, X_test, y_test):
    """Entrena y evalúa Neural Network en MNIST."""

    print("=== Neural Network (MLP) ===")
    print("Arquitectura: 784 → 128 → 64 → 10")

    nn = NeuralNetworkMNIST([784, 128, 64, 10])
    nn.fit(X_train, y_train, epochs=10, batch_size=32, learning_rate=0.01)

    train_acc = nn.score(X_train, y_train)
    test_acc = nn.score(X_test, y_test)

    print(f"\nTrain Accuracy: {train_acc:.2%}")
    print(f"Test Accuracy:  {test_acc:.2%}")

    return nn
```

💻 Parte 5: Benchmark y Comparación

5.1 Pipeline Completo

```
"""
Pipeline completo que ejecuta todos los análisis
y compara los modelos.
"""

import numpy as np
import time

def run_mnist_pipeline(X_train, y_train, X_test, y_test, use_subset: bool = True):
    """
```

```
    Ejecuta el pipeline completo de MNIST.

    Args:
        use_subset: Si True, usa solo 10k samples para rapidez
    """
    if use_subset:
        X_train = X_train[:10000]
        y_train = y_train[:10000]
        X_test = X_test[:2000]
        y_test = y_test[:2000]

    # Normalizar
    X_train = X_train / 255.0
    X_test = X_test / 255.0

    print("=" * 60)
    print("MNIST ANALYST PIPELINE")
    print("=" * 60)
    print(f"Train samples: {len(X_train)}")
    print(f"Test samples: {len(X_test)}")
    print(f"Features: {X_train.shape[1]}")
    print("=" * 60)

    results = {}

    # === FASE 1: Unsupervised ===
    print("\n" + "=" * 60)
    print("FASE 1: EXPLORACIÓN NO SUPERVISADA")
    print("=" * 60)

    # PCA
    print("\n[PCA]")
    pca = PCA(n_components=50)
    pca.fit(X_train)
    print(f"Varianza explicada (50 PCs): {sum(pca.explained_variance_ratio_):.2%}")

    # K-Means
    print("\n[K-Means]")
    start = time.time()
    kmeans = KMeans(n_clusters=10, seed=42)
    kmeans.fit(X_train)
    kmeans_time = time.time() - start
    print(f"Inercia: {kmeans.inertia_:.2f}")
    print(f"Tiempo: {kmeans_time:.2f}s")

    # === FASE 2: Supervised ===
    print("\n" + "=" * 60)
    print("FASE 2: CLASIFICACIÓN SUPERVISADA")
    print("=" * 60)

    # Logistic Regression
    print("\n[Logistic Regression One-vs-All]")
    start = time.time()
    lr_model = LogisticRegressionOvA(n_classes=10, lr=0.1, n_iter=100)
    lr_model.fit(X_train, y_train)
    lr_time = time.time() - start
    lr_acc = lr_model.score(X_test, y_test)
    print(f"Test Accuracy: {lr_acc:.2%}")
    print(f"Tiempo: {lr_time:.2f}s")
    results['Logistic Regression'] = lr_acc

    # === FASE 3: Deep Learning ===
    print("\n" + "=" * 60)
    print("FASE 3: DEEP LEARNING")
    print("=" * 60)

    # Neural Network
    print("\n[Neural Network MLP]")
    start = time.time()
    nn = NeuralNetworkMNIST([784, 128, 64, 10])
    nn.fit(X_train, y_train, epochs=5, batch_size=32, learning_rate=0.01, verbose=False)
    nn_time = time.time() - start
    nn_acc = nn.score(X_test, y_test)
    print(f"Test Accuracy: {nn_acc:.2%}")
    print(f"Tiempo: {nn_time:.2f}s")
```

```
    results['Neural Network'] = nn_acc

    # === COMPARACIÓN ===
    print("\n" + "=" * 60)
    print("COMPARACIÓN DE MODELOS")
    print("=" * 60)

    print("\nModelo               | Accuracy | Mejora vs LR")
    print("-" * 50)
    baseline = results['Logistic Regression']
    for name, acc in results.items():
        improvement = ((acc - baseline) / baseline) * 100 if name != 'Logistic Regression' else 0
        print(f"{name:<20} | {acc:.2%}    | {improvement:+.1f}%")

    # === ANÁLISIS ===
    print("\n" + "=" * 60)
    print("ANÁLISIS: ¿Por qué NN es mejor?")
    print("=" * 60)
    print("""
1. NO-LINEALIDAD: ReLU permite aprender fronteras no lineales.
   Logistic Regression solo puede aprender fronteras lineales.

2. REPRESENTACIÓN JERÁRQUICA: Las capas ocultas aprenden features
   de complejidad creciente (bordes → formas → dígitos).

3. CAPACIDAD: Más parámetros = puede memorizar patrones más complejos.
   Pero cuidado con overfitting si hay pocos datos.

4. COMPOSICIÓN: La red compone funciones simples (lineales + activaciones)
   para aproximar funciones complejas.
""")

    return results
```

📦 Entregable Final

MODEL_COMPARISON.md

```
# Model Comparison Report - MNIST Analyst

## Executive Summary

This project demonstrates competency in all three courses of the Machine Learning
Pathway (Line 1) through a complete analysis of the MNIST dataset.

## Results

| Model | Test Accuracy | Training Time |
|-------|---------------|---------------|
| K-Means (Unsupervised) | N/A (clustering) | ~5s |
| Logistic Regression | ~85-90% | ~30s |
| Neural Network (MLP) | ~95%+ | ~60s |

## Analysis

### Why does the Neural Network outperform Logistic Regression?

1. **Non-linearity**: ReLU activations allow learning non-linear decision boundaries
2. **Hierarchical features**: Hidden layers learn increasingly abstract representations
3. **Capacity**: More parameters enable capturing complex patterns

### Mathematical Explanation

Logistic Regression:
```

$\hat{y} = \sigma(Wx + b)$

```
- Single linear transformation + sigmoid
- Can only learn linear decision boundaries

Neural Network:
```

$\hat{y} = \text{softmax}(W_3 \cdot \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 x + b_1) + b_2) + b_3)$

```
- Multiple non-linear transformations
- Universal function approximator

### PCA Insights

- 50 components retain ~85% of variance
- First 2 components show partial class separation
- Principal components capture stroke patterns

### K-Means Insights

- Cluster centroids resemble average digit shapes
- Some digits (1, 7) cluster well; others (4, 9) overlap
- Unsupervised clustering achieves ~60% purity

## Conclusion

The Neural Network achieves the highest accuracy by learning hierarchical,
non-linear representations of the input images. This project demonstrates
practical implementation of Supervised Learning, Unsupervised Learning,
and Deep Learning algorithms from scratch.
```

📊 Análisis Bias-Variance (v3.2)

> 🎓 **Concepto Central de la Maestría:** Entender el tradeoff Bias-Variance es fundamental para diseñar modelos de ML.

El Tradeoff Bias-Variance

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│   ERROR TOTAL = BIAS² + VARIANCE + RUIDO IRREDUCIBLE    │
│                                                         │
│   BIAS (Sesgo): Error por suposiciones simplificadoras  │
│   - Modelo muy simple → NO captura patrones → UNDERFITTING │
│                                                         │
│   VARIANCE (Varianza): Error por sensibilidad a datos   │
│   - Modelo muy complejo → Memoriza ruido → OVERFITTING  │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

Análisis de los Modelos del Proyecto

Modelo	Bias	Variance	Comportamiento Esperado
Logistic Regression	Alto	Baja	Underfitting (solo límites lineales)
MLP pequeño (`128-64`)	Medio	Media	Balance óptimo
MLP grande (`512-256-128`)	Bajo	Alta	Riesgo de overfitting

Tu Entregable: Sección en MODEL_COMPARISON.md

Añade una sección que responda:

1. **¿Por qué Logistic Regression tiene alto bias?**
2. Solo puede aprender fronteras de decisión lineales
3. MNIST tiene patrones no lineales (curvas, esquinas)
4. **¿Por qué MLP puede tener alta variance?**
5. Muchos parámetros pueden memorizar ejemplos de entrenamiento
6. Solución: Regularización `L2`, Dropout, Early Stopping
7. **Experimento práctico:**
8. Entrenar MLP con diferentes tamaños
9. Graficar train_accuracy vs test_accuracy
10. Identificar punto de overfitting

```
# Código para análisis Bias-Variance
def train_and_evaluate(hidden_sizes: list, X_train, y_train, X_test, y_test):
    """Entrenar modelos de diferentes tamaños y comparar."""
    results = []

    for sizes in hidden_sizes:
        model = NeuralNetwork(layers=[784] + list(sizes) + [10])
        model.fit(X_train, y_train, epochs=100)

        train_acc = model.score(X_train, y_train)
        test_acc = model.score(X_test, y_test)
        gap = train_acc - test_acc  # Gap grande = overfitting

        results.append({
            'hidden_sizes': sizes,
            'train_acc': train_acc,
            'test_acc': test_acc,
            'gap': gap
        })

    return results

# Experimento
sizes_to_test = [
    (32,),           # Muy pequeño (alto bias)
    (128, 64),       # Medio (balanceado)
    (512, 256, 128)  # Grande (alta variance)
]
```

📝 Formato de Informe: Paper Científico (v3.2)

💎 **Profesionalismo:** El notebook final debe tener el formato de un paper académico.

Estructura del Jupyter Notebook Final

```
# MNIST Digit Classification: A From-Scratch Implementation

## Abstract
[3-4 oraciones resumiendo objetivo, métodos y resultados principales]

## 1. Introduction
- Problema: clasificación de dígitos escritos a mano
- Motivación: demostrar competencia en ML
- Contribución: implementación 100% desde cero

## 2. Dataset
- Descripción de MNIST (60K train, 10K test, 28x28 pixels)
- Preprocesamiento aplicado

## 3. Methodology
### 3.1 Unsupervised Learning
- PCA: reducción dimensional
- K-Means: clustering

### 3.2 Supervised Learning
- Logistic Regression One-vs-All

### 3.3 Deep Learning
- MLP architecture: 784→128→64→10
- Training: SGD with momentum

## 4. Results
### 4.1 PCA Analysis
[Gráficos de varianza explicada, visualización 2D]

### 4.2 K-Means Clustering
[Centroides, pureza de clusters]

### 4.3 Model Comparison
[Tabla comparativa, matriz de confusión]

### 4.4 Bias-Variance Analysis
[Gap train-test para diferentes modelos]
```

```
## 5. Discussion
- ¿Por qué MLP supera a Logistic Regression?
- Limitaciones del estudio
- Trabajo futuro

## 6. Conclusion
[2-3 oraciones de cierre]

## References
- LeCun, Y., et al. "Gradient-based learning applied to document recognition."
- Deep Learning Book (Goodfellow et al.)
```

🔎 Análisis de Errores: Nivel Senior (v3.3)

> 💎 **Profesionalismo:** No solo muestres el accuracy. Muestra las imágenes que la red falló y explica por qué.

Por Qué Es Importante

```
ANÁLISIS DE ERRORES = LO QUE SEPARA JUNIOR DE SENIOR

Junior: "Mi modelo tiene 92% accuracy"

Senior: "Mi modelo tiene 92% accuracy. Los errores se concentran en:
        - Confusión 4↔9 (formas similares)
        - Confusión 3↔8 (curvas similares)
        - Dígitos mal escritos o cortados
        Esto sugiere que el modelo necesita más ejemplos
        de estos casos difíciles o data augmentation."
```

Script: Análisis de Errores

```
"""
Error Analysis - Visualización de Fallos del Modelo
Nivel Senior: No solo accuracy, también entender los errores.
"""
import numpy as np
import matplotlib.pyplot as plt
from typing import Tuple, List


def analyze_errors(
    model,
    X_test: np.ndarray,
    y_test: np.ndarray,
    n_errors: int = 20
) -> dict:
    """
    Analiza y visualiza los errores del modelo.

    Args:
        model: Modelo entrenado (con .predict())
        X_test: Datos de test
        y_test: Labels de test
        n_errors: Número de errores a visualizar

    Returns:
        Diccionario con análisis completo
    """
    # Predicciones
    y_pred = model.predict(X_test)

    # Identificar errores
    errors_mask = y_pred != y_test
    error_indices = np.where(errors_mask)[0]

    print("=" * 60)
    print("ANÁLISIS DE ERRORES")
    print("=" * 60)
    print(f"Total errores: {len(error_indices)} / {len(y_test)}")
    print(f"Error rate: {100 * len(error_indices) / len(y_test):.2f}%")
```

```
    # Matriz de confusión de errores
    confusion_pairs = {}
    for idx in error_indices:
        pair = (y_test[idx], y_pred[idx])
        confusion_pairs[pair] = confusion_pairs.get(pair, 0) + 1

    # Top confusiones
    sorted_pairs = sorted(confusion_pairs.items(), key=lambda x: -x[1])

    print("\n📊 TOP CONFUSIONES:")
    for (true, pred), count in sorted_pairs[:10]:
        print(f"  {true} → {pred}: {count} errores")

    # Visualizar errores
    fig, axes = plt.subplots(4, 5, figsize=(12, 10))
    fig.suptitle("Ejemplos de Errores del Modelo", fontsize=14)

    for i, ax in enumerate(axes.flat):
        if i < min(n_errors, len(error_indices)):
            idx = error_indices[i]
            img = X_test[idx].reshape(28, 28)
            ax.imshow(img, cmap='gray')
            ax.set_title(f"True: {y_test[idx]}, Pred: {y_pred[idx]}",
                         color='red', fontsize=10)
            ax.axis('off')
        else:
            ax.axis('off')

    plt.tight_layout()
    plt.savefig('error_analysis.png', dpi=150)
    plt.show()

    return {
        'n_errors': len(error_indices),
        'error_rate': len(error_indices) / len(y_test),
        'confusion_pairs': sorted_pairs,
        'error_indices': error_indices
    }


def plot_learning_curves(
    train_losses: List[float],
    val_losses: List[float],
    train_accs: List[float],
    val_accs: List[float]
) -> None:
    """
    Visualiza curvas de aprendizaje para diagnóstico Bias-Variance.

    - Train alto, Val alto → Underfitting (High Bias)
    - Train bajo, Val alto → Overfitting (High Variance)
    - Train bajo, Val bajo → Buen modelo
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    epochs = range(1, len(train_losses) + 1)

    # Loss curves
    ax1.plot(epochs, train_losses, 'b-', label='Train Loss')
    ax1.plot(epochs, val_losses, 'r-', label='Validation Loss')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.set_title('Learning Curves: Loss')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Accuracy curves
    ax2.plot(epochs, train_accs, 'b-', label='Train Accuracy')
    ax2.plot(epochs, val_accs, 'r-', label='Validation Accuracy')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.set_title('Learning Curves: Accuracy')
    ax2.legend()
    ax2.grid(True, alpha=0.3)
```

```
    # Diagnóstico
    final_gap = train_accs[-1] - val_accs[-1]

    if val_accs[-1] < 0.7:
        diagnosis = "⚠️ UNDERFITTING: Modelo muy simple o poco entrenamiento"
    elif final_gap > 0.1:
        diagnosis = "⚠️ OVERFITTING: Gap train-val > 10%"
    else:
        diagnosis = "✓ BUEN AJUSTE: Modelo generaliza bien"

    fig.suptitle(f"Diagnóstico: {diagnosis}", fontsize=12, y=1.02)

    plt.tight_layout()
    plt.savefig('learning_curves.png', dpi=150)
    plt.show()

    print("\n📈 DIAGNÓSTICO BIAS-VARIANCE:")
    print(f"  Train Accuracy Final: {train_accs[-1]:.4f}")
    print(f"  Val Accuracy Final:   {val_accs[-1]:.4f}")
    print(f"  Gap:                  {final_gap:.4f}")
    print(f"  → {diagnosis}")
```

Sección Obligatoria en MODEL_COMPARISON.md

```
## Error Analysis

### Top Confusiones del Modelo

| True → Pred | Count | Explicación |
|-------------|-------|-------------|
| 4 → 9 | 23 | Formas similares (bucle arriba) |
| 9 → 4 | 18 | Formas similares |
| 3 → 8 | 15 | Curvas similares |
| 7 → 1 | 12 | Trazo vertical dominante |

### Visualización de Errores

![Errores del modelo](error_analysis.png)

### Interpretación

Los errores se concentran principalmente en dígitos con formas similares.
Esto sugiere que:
1. El modelo captura bien las features principales
2. Features más finas (bucles, cruces) necesitan más ejemplos
3. Data augmentation podría ayudar
```

✅ Checklist de Finalización (v3.3)

Semana 21: EDA + No Supervisado

- [ ] PCA reduce MNIST a 2 D/ 5 0 D con visualización
- [ ] Analicé varianza explicada por componente
- [ ] K-Means agrupa dígitos sin etiquetas
- [ ] Visualicé centroides como imágenes 2 8 x 2 8

Semana 22: Clasificación Supervisada

- [ ] Logistic Regression One-vs-All funcional
- [ ] Accuracy > 8 5 % en test set
- [ ] Calculé Precision, Recall, F 1 por clase
- [ ] Analicé matriz de confusión

Semana 23: Deep Learning

- [ ] MLP con arquitectura 7 8 $4 \rightarrow 1$ 2 $8 \rightarrow 6$ $4 \rightarrow 1$ 0
- [ ] Forward y backward pass implementados
- [ ] Mini-batch SGD funcionando
- [ ] Accuracy > 9 0 % en test set

Semana 24: Benchmark + Informe

- [ ] MODEL_COMPARISON.md completo

- [ ] README.md profesional en inglés

Requisitos v3.3

- [ ] **Análisis Bias-Variance** con experimento práctico
- [ ] **Notebook en formato Paper** (Abstract, Methods, Results, Discussion)
- [ ] **Análisis de Errores** con visualización de fallos
- [ ] **Curvas de Aprendizaje** con diagnóstico Bias-Variance
- [ ] Sección "Error Analysis" en MODEL_COMPARISON.md
- [ ] `mypy src/` pasa sin errores
- [ ] `pytest tests/` pasa sin errores

Metodología Feynman

- [ ] Puedo explicar por qué MLP supera a Logistic en 5 líneas
- [ ] Puedo explicar Bias vs Variance en 5 líneas
- [ ] Puedo explicar por qué 4 ↔️ 9 se confunden frecuentemente

🔗 Navegación

Anterior	Índice
07_DEEP_LEARNING	00_INDICE

CHECKLIST FINAL

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

☑️ Checklist Final - ML Specialist v3.3

Verificación completa del programa de 24 **semanas** con validación matemática rigurosa.

📚 Fase 1: Fundamentos (Semanas 1-8)

Módulo 01: Python + Pandas + NumPy (Semanas 1-2)

Conocimiento

- [ ] Pandas: cargar CSV con `read_csv()`
- [ ] Pandas: limpiar datos con `dropna()`, `fillna()`
- [ ] Pandas: selección con `.loc[]`, `.iloc[]`
- [ ] Pandas → NumPy: `.to_numpy()`
- [ ] NumPy: creación de arrays (1D, 2D, 3D)
- [ ] NumPy: indexing, slicing, broadcasting
- [ ] NumPy: agregaciones por eje (axis=0, axis=1)
- [ ] **Conozco los 5 errores comunes de NumPy y sus soluciones**

Estándares Profesionales (v3.2)

- [ ] `mypy src/` pasa sin errores
- [ ] `ruff check src/` pasa sin errores
- [ ] Al menos 3 tests con `pytest` pasando

Metodología Feynman

- [ ] Puedo explicar broadcasting en 5 líneas sin jerga

Módulo 02: Álgebra Lineal para ML (Semanas 3-5)

- [ ] Producto punto y significado geométrico
- [ ] Normas L_1, L_2, L_∞ implementadas
- [ ] Distancia euclidiana y similitud coseno
- [ ] Multiplicación de matrices con `@`
- [ ] Eigenvalues/eigenvectors con `np.linalg.eig()`
- [ ] SVD con `np.linalg.svd()`
- [ ] `linear_algebra.py` con tests pasando

Módulo 03: Cálculo Multivariante (Semanas 6-7)

- [ ] Derivadas parciales calculadas
- [ ] Gradiente de funciones multivariables
- [ ] Gradient Descent implementado desde cero
- [ ] Efecto del learning rate entendido
- [ ] Chain Rule aplicada a funciones compuestas
- [ ] `calculus.py` con Gradient Descent funcional

Gradient Checking (v3.3 - Obligatorio)

- [ ] **`grad_check.py` implementado**
- [ ] **Validé derivadas de MSE, sigmoid y capa lineal**
- [ ] Error relativo $< 10^{-7}$ en todos los tests

Módulo 04: Probabilidad para ML (Semana 8)

- [ ] Teorema de Bayes explicado con ejemplo
- [ ] Gaussiana univariada: PDF implementada
- [ ] Gaussiana multivariada: concepto entendido
- [ ] MLE: conexión con Cross-Entropy explicada
- [ ] **Softmax con Log-Sum-Exp trick implementado (v3.3)**
- [ ] `probability.py` con tests pasando

🤖 Fase 2: Núcleo de ML (Semanas 9-20) ⭐ PATHWAY

Módulo 05: Supervised Learning (Semanas 9-12)

Conocimiento

- [ ] Regresión lineal (Normal Equation + GD)
- [ ] MSE y su gradiente derivado

- [ ] Regresión logística desde cero
- [ ] Sigmoid y binary cross-entropy
- [ ] Matriz de confusión (TP, TN, FP, FN)
- [ ] Accuracy, Precision, Recall, F 1 implementados
- [ ] Train/test split manual
- [ ] K-fold cross validation
- [ ] Regularización L 2 (Ridge)

Derivación Analítica (v3.2 - Obligatorio)

- [ ] **Derivé el gradiente de Cross-Entropy a mano**
- [ ] **Documento con derivación completa (Markdown o LaTeX)**

Metodología Feynman

- [ ] Puedo explicar sigmoid vs softmax en 5 líneas

Módulo 06: Unsupervised Learning (Semanas 13-16)

- [ ] K-Means con K-Means++ initialization
- [ ] Algoritmo de Lloyd (asignar-actualizar-repetir)
- [ ] Inercia y método del codo
- [ ] PCA usando SVD (`np.linalg.svd()`)
- [ ] Varianza explicada y elección de n_components
- [ ] Reconstrucción desde componentes principales
- [ ] `kmeans.py` y `pca.py` con tests pasando

Módulo 07: Deep Learning + CNNs (Semanas 17-20)

Conocimiento

- [ ] Neurona artificial y perceptrón
- [ ] Sigmoid, ReLU, tanh, softmax + derivadas
- [ ] Problema XOR y su no-linealidad
- [ ] Forward pass para MLP
- [ ] Backpropagation con Chain Rule
- [ ] SGD, Momentum, Adam implementados
- [ ] Red resuelve problema XOR
- [ ] **CNNs (teoría):** convolución, stride, padding, pooling

Derivación Analítica (v3.2 - Obligatorio)

- [ ] **Derivé las ecuaciones de backprop para red de 2 capas**
- [ ] **Diagrama de grafo computacional**

Metodología Feynman

- [ ] Puedo explicar backpropagation en 5 líneas sin jerga

🎯 Fase 3: Proyecto MNIST Analyst (Semanas 21-24)

Semana 21: EDA + No Supervisado

- [ ] MNIST cargado y normalizado
- [ ] PCA reduce a 2 D con visualización
- [ ] Varianza explicada analizada
- [ ] K-Means agrupa dígitos sin etiquetas
- [ ] Centroides visualizados como imágenes 2 8 x 2 8

Semana 22: Clasificación Supervisada

- [ ] Logistic Regression One-vs-All implementado
- [ ] Accuracy > 8 5 % en test set
- [ ] Precision, Recall, F 1 por clase
- [ ] Matriz de confusión analizada
- [ ] Errores visualizados (imágenes mal clasificadas)

Semana 23: Deep Learning

- [ ] MLP 7 8 4 → 1 2 8 → 6 4 → 1 0 implementado
- [ ] Forward y backward pass funcionales
- [ ] Mini-batch SGD funcionando

- [ ] Accuracy > 90% en test set

Semana 24: Benchmark + Informe

- [ ] Comparación MLP vs Logistic Regression
- [ ] `MODEL_COMPARISON.md` explicando diferencias
- [ ] `README.md` profesional en inglés
- [ ] Demo notebook completo

Requisitos v3.2 (Obligatorios)

- [ ] **Análisis Bias-Variance** con experimento práctico (3 tamaños de MLP)
- [ ] **Notebook en formato Paper** (Abstract, Methods, Results, Discussion)
- [ ] `mypy src/` pasa sin errores en todo el proyecto
- [ ] `pytest tests/` con cobertura significativa

Metodología Feynman

- [ ] Puedo explicar Bias vs Variance en 5 líneas
- [ ] Puedo explicar por qué MLP supera a Logistic en 5 líneas

💻 Código

Estructura del Proyecto MNIST

```
mnist-analyst/
├── src/
│   ├── __init__.py
│   ├── data_loader.py
│   ├── linear_algebra.py
│   ├── probability.py
│   ├── pca.py
│   ├── kmeans.py
│   ├── logistic_regression.py
│   ├── neural_network.py
│   ├── metrics.py
│   └── pipeline.py
├── notebooks/
│   ├── 01_eda_pca_kmeans.ipynb
│   ├── 02_logistic_classification.ipynb
│   └── 03_neural_network_benchmark.ipynb
├── tests/
│   └── test_*.py
├── docs/
│   └── MODEL_COMPARISON.md
├── README.md
└── requirements.txt
```

Calidad de Código

- [ ] Type hints en todas las funciones
- [ ] Docstrings con Args, Returns
- [ ] `mypy src/` pasa sin errores
- [ ] Código vectorizado (sin loops innecesarios)

Tests

- [ ] Tests unitarios para cada módulo
- [ ] Tests para edge cases
- [ ] Todos los tests pasan

📝 Documentación

README.md del Proyecto

- [ ] Descripción del proyecto
- [ ] Instrucciones de instalación
- [ ] Ejemplo de uso
- [ ] Resultados y métricas
- [ ] Escrito en inglés

MODEL_COMPARISON.md

- [ ] Tabla comparativa de modelos

- [ ] Explicación matemática de diferencias
- [ ] Análisis de PCA
- [ ] Análisis de K-Means
- [ ] Conclusiones

🚀 Verificación Final

```
# 1. Tests
python -m pytest tests/ -v

# 2. Pipeline completo
python -c "
from src.pipeline import run_mnist_pipeline
# Ejecutar pipeline demo
"

# 3. Verificar accuracy
# Logistic Regression: > 85%
# Neural Network: > 90%
```

✅ Declaración de Completitud

Por Fase

- [ ] **Fase 1:** Fundamentos matemáticos dominados
- [ ] **Fase 2:** Algoritmos ML implementados desde cero
- [ ] **Fase 3:** Proyecto MNIST completo

Por Curso del Pathway

- [ ] **Supervised Learning:** Regresión + Clasificación
- [ ] **Unsupervised Learning:** K-Means + PCA
- [ ] **Deep Learning:** MLP con Backpropagation

Métricas Finales

Métrica	Objetivo	Logrado
Logistic Regression Accuracy	> 85%	___%
Neural Network Accuracy	> 90%	___%
Módulos completados	8/8	___/8
Tests pasando	100%	___%

Fecha de completitud: ___

Listo para el MS in AI Pathway - Línea 1: ☐ Sí ☐ No

RECURSOS DE APRENDIZAJE

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

📚 Recursos de Aprendizaje - ML Specialist v3.0

Recursos organizados para dominar la **Línea 1 : Machine Learning** del MS in AI Pathway.

🎯 CURSOS DEL PATHWAY (LÍNEA 1 - FOCO)

Estos son los 3 **cursos** de la Línea de Machine Learning:

Curso	Enlace	Módulo de Preparación
Introduction to ML: Supervised Learning	Coursera	Módulo 04
Unsupervised Algorithms in ML	Coursera	Módulo 05
Introduction to Deep Learning	Coursera	Módulo 06

💡 Puedes auditar estos cursos GRATIS en Coursera para ver el contenido.

Línea 2: Probabilidad y Estadística (Lectura Opcional)

Curso	Enlace
Probability Theory: Foundation for Data Science	Coursera
Statistical Inference for Estimation	Coursera
Discrete-Time Markov Chains	Coursera

🔥 CURSOS DE PREPARACIÓN RECOMENDADOS

Machine Learning (Prioridad Alta)

Curso	Plataforma	Por Qué
Machine Learning Specialization	Coursera (Andrew Ng)	El mejor curso de ML, usa Python
Deep Learning Specialization	Coursera (Andrew Ng)	Profundiza en redes neuronales

Matemáticas para ML (Esencial)

Curso	Plataforma	Por Qué
Mathematics for ML: Linear Algebra	Coursera (Imperial)	Vectores, matrices, eigenvalues
Mathematics for ML: Multivariate Calculus	Coursera (Imperial)	Gradientes, Chain Rule, optimización

NumPy y Python Científico

Curso	Plataforma	Por Qué
NumPy Documentation	numpy.org	Referencia oficial
Real Python - NumPy	Real Python	Tutorial práctico

📖 Libros

Machine Learning (Prioridad Alta)

Libro	Autor	Por Qué
Hands-On Machine Learning	Aurélien Géron	Práctico, código completo
Pattern Recognition and ML	Christopher Bishop	Fundamentos teóricos sólidos

Libro	Autor	Por Qué
The Hundred-Page ML Book	Andriy Burkov	Resumen conciso

Deep Learning

Libro	Autor	Por Qué
Deep Learning	Goodfellow et al.	Gratis online - La biblia
Neural Networks and Deep Learning	Michael Nielsen	Gratis online - Muy didáctico

Matemáticas para ML

Libro	Autor	Por Qué
Mathematics for ML	Deisenroth et al.	Gratis online - Fundamental
Linear Algebra Done Right	Axler	Álgebra lineal rigurosa

🎥 Videos Esenciales

Canales de YouTube

Canal	Tema	Por Qué
3Blue1Brown	Matemáticas + DL	Visualización excepcional
StatQuest	ML + Estadística	Explicaciones claras
Sentdex	ML práctico	Implementaciones desde cero

Playlists Esenciales

Playlist	Tema	Link
Neural Networks	Deep Learning	3B1B Neural Networks
Linear Algebra	Matemáticas	3B1B Linear Algebra
Calculus	Matemáticas	3B1B Calculus
Machine Learning	ML	StatQuest ML

🛠️ Herramientas

Desarrollo

Herramienta	Propósito	Instalación
Python 3.11+	Lenguaje base	`brew install python` / `apt install python3`
VS Code	Editor de código	Descargar de web
Git	Control de versiones	`brew install git` / `apt install git`

Python ML Stack

Paquete	Propósito	Comando
numpy	Operaciones vectoriales	`pip install numpy`
matplotlib	Visualización	`pip install matplotlib`

Paquete	Propósito	Comando
mypy	Type checking	`pip install mypy`
pytest	Testing	`pip install pytest`

Datasets

Dataset	Descripción	Acceso
MNIST	Dígitos escritos a mano (2 8 x 2 8)	`keras.datasets.mnist` o descarga directa
Iris	Clasificación clásica (4 features)	CSV disponible en UCI
Boston Housing	Regresión (precios de casas)	CSV disponible en Kaggle

📅 Ruta de Aprendizaje (26 Semanas)

Semanas 1-8: Fundamentos Matemáticos

Semana	Tema	Recursos
1 - 2	Python Científico + NumPy	Módulo 0 1 + NumPy docs
3 - 5	Álgebra Lineal	Módulo 0 2 + 3 B 1 B Linear Algebra
6 - 8	Cálculo Multivariante	Módulo 0 3 + 3 B 1 B Calculus

Semanas 9-20: Núcleo ML

Semana	Tema	Recursos
9 - 1 2	Supervised Learning	Módulo 0 4 + StatQuest ML
1 3 - 1 6	Unsupervised Learning	Módulo 0 5 + Andrew Ng videos
1 7 - 2 0	Deep Learning	Módulo 0 6 + 3 B 1 B Neural Networks

Semanas 21-26: Proyecto MNIST

Semana	Fase	Entregable
2 1 - 2 2	PCA + K-Means	Exploración no supervisada
2 3 - 2 4	Logistic Regression	Clasificación supervisada
2 5 - 2 6	MLP + Benchmark	Comparación de modelos

🔗 Links Directos

Programa

- MS in AI - CU Boulder
- Pathway Admissions

Libros Gratis

- Mathematics for ML
- Deep Learning Book
- Neural Networks and DL

Datasets

- MNIST
- UCI ML Repository
- Kaggle Datasets

GLOSARIO TÉCNICO

MS IN AI PATHWAY - ML SPECIALIST V3.0

MATEMÁTICAS APLICADAS A CÓDIGO

📖 Glosario Técnico - ML Specialist v3.0

Definiciones A-Z de términos de Machine Learning usados en la guía.

A

Activation Function

Definición: Función no lineal aplicada a la salida de una neurona.
Ejemplos: ReLU, Sigmoid, Tanh, Softmax.
Por qué: Sin activaciones, una red sería solo transformaciones lineales.

Adam

Definición: Adaptive Moment Estimation - optimizador que combina Momentum y RMSprop.
Parámetros: lr= 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 1e\text{-}8$
Uso: Default moderno para entrenar redes neuronales.

Accuracy

Definición: Proporción de predicciones correctas.
Fórmula: (TP + TN) / (TP + TN + FP + FN)
Limitación: Engañoso con clases desbalanceadas.

B

Backpropagation

Definición: Algoritmo para calcular gradientes en redes neuronales usando la Chain Rule.
Proceso: Forward pass → calcular loss → backward pass → actualizar pesos.
Base matemática: $\partial L/\partial w = \partial L/\partial a \cdot \partial a/\partial z \cdot \partial z/\partial w$

Batch Size

Definición: Número de muestras procesadas antes de actualizar pesos.
Trade-off: Grande = estable pero lento; pequeño = ruidoso pero rápido.
Común: 32, 64, 128, 256.

Bias (parámetro)

Definición: Término constante en z = Wx + b que permite desplazar la función.
Analogía: El intercepto en una recta y = mx + b.

Binary Cross-Entropy

Definición: Función de pérdida para clasificación binaria.
Fórmula: $L = -[y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y})]$
Uso: Salida sigmoid, predicción de probabilidad.

Broadcasting

Definición: Expansión automática de arrays para operaciones elemento a elemento.
Ejemplo: array(3, 1) + array(1, 4) → array(3, 4)
Regla: Dimensiones deben ser iguales o una debe ser 1.

C

Centroid

Definición: Punto central de un cluster (promedio de sus puntos).
En K-Means: Se actualiza iterativamente hasta convergencia.

Chain Rule

Definición: Regla para derivar funciones compuestas.
Fórmula: d/dx f(g(x)) = f'(g(x)) · g'(x)
Importancia: Base matemática de Backpropagation.

Classification

Definición: Tarea de predecir una categoría discreta.
Binaria: 2 clases (spam/no spam).
Multiclase: > 2 clases (dígitos 0-9).

Clustering

Definición: Agrupar puntos similares sin etiquetas supervisadas.
Algoritmos: K-Means, DBSCAN, Hierarchical.

Confusion Matrix
Definición: Tabla que muestra predicciones vs valores reales.
Componentes: TP, TN, FP, FN.

Convergence
Definición: Cuando el algoritmo deja de mejorar significativamente.
Criterio: Cambio en loss < tolerancia, o gradiente ≈ 0 .

Cosine Similarity
Definición: Similitud basada en el ángulo entre vectores.
Fórmula: cos(θ) = (a·b) / (||a|| ||b||)
Rango: [- 1 , 1], donde 1 = idénticos.

Cross-Validation
Definición: Técnica para evaluar modelo dividiendo datos en K folds.
K-Fold: Entrenar K veces, cada vez con diferente fold como validación.
Uso: Estimar rendimiento real, evitar overfitting.

D

Deep Learning
Definición: ML con redes neuronales de múltiples capas ocultas.
Ventaja: Aprende features automáticamente.
Requisito: Muchos datos y compute.

Derivative
Definición: Tasa de cambio instantánea de una función.
Notación: f'(x), df/dx, ∂f/∂x (parcial).

Dimensionality Reduction
Definición: Reducir número de features preservando información.
Métodos: PCA, t-SNE, UMAP.
Uso: Visualización, eliminar ruido, acelerar entrenamiento.

Dot Product
Definición: Suma de productos elemento a elemento.
Fórmula: $a \cdot b = \Sigma a_i b_i$
Uso: Similitud, proyecciones, capas de red neuronal.

E

Eigenvalue / Eigenvector
Definición: Para matriz A, Av = λv donde v es eigenvector y λ es eigenvalue.
Interpretación: Direcciones principales de la transformación.
Uso en ML: PCA usa eigenvectors de la matriz de covarianza.

Epoch
Definición: Una pasada completa por todo el dataset de entrenamiento.
Típico: 1 0 - 1 0 0 epochs dependiendo del problema.

Euclidean Distance
Definición: Distancia en línea recta entre dos puntos.
Fórmula: $d(a,b) = \sqrt{\Sigma(a_i - b_i)^2}$
Uso: K-Means, KNN.

F

F1 Score
Definición: Media armónica de Precision y Recall.
Fórmula: F 1 = 2 · (P · R) / (P + R)
Uso: Balance entre precision y recall.

Feature
Definición: Variable de entrada (columna) en un dataset.
Ejemplo: En MNIST, cada píxel es un feature (7 8 4 total).

Forward Pass
Definición: Propagación de input a través de la red para obtener output.
Cálculo: z = Wx + b, a = activation(z), repetir por capa.

G

Gradient

Definición: Vector de derivadas parciales.
Notación: $\nabla f = [\partial f/\partial x_1, \partial f/\partial x_2, ...]$
Propiedad: Apunta en dirección de máximo ascenso.

Gradient Descent

Definición: Algoritmo de optimización que sigue el gradiente negativo.
Update: $\theta = \theta - \alpha \cdot \nabla L(\theta)$
Variantes: Batch, Mini-batch, Stochastic (SGD).

H

Hidden Layer

Definición: Capa entre input y output en una red neuronal.
Función: Aprende representaciones intermedias.

Hyperparameter

Definición: Parámetro configurado antes del entrenamiento (no aprendido).
Ejemplos: Learning rate, número de capas, batch size.

I

Inertia

Definición: Suma de distancias cuadradas de puntos a sus centroides.
En K-Means: Métrica a minimizar.
Uso: Método del codo para elegir K.

K

K-Means

Definición: Algoritmo de clustering que particiona en K grupos.
Pasos: 1) Inicializar centroides 2) Asignar puntos 3) Actualizar centroides 4) Repetir.
Complejidad: $O(n \cdot k \cdot i \cdot d)$ donde i=iteraciones, d=dimensiones.

K-Means++

Definición: Inicialización inteligente para K-Means.
Método: Elegir centroides iniciales lejos entre sí.
Ventaja: Mejor convergencia, evita mínimos locales.

L

L1 Norm (Manhattan)

Definición: Suma de valores absolutos.
Fórmula: $||x||_1 = \Sigma|x_i|$
Uso: Regularización Lasso, promueve sparsity.

L2 Norm (Euclidean)

Definición: Raíz de suma de cuadrados (longitud del vector).
Fórmula: $||x||_2 = \sqrt{\Sigma x_i^2}$
Uso: Regularización Ridge, normalización.

Learning Rate

Definición: Tamaño del paso en Gradient Descent.
Símbolo: α (alpha) o lr.
Trade-off: Grande = rápido pero inestable; pequeño = estable pero lento.

Linear Regression

Definición: Modelo que predice valor continuo con combinación lineal.
Fórmula: $\hat{y} = X\theta$
Loss: MSE (Mean Squared Error).

Logistic Regression

Definición: Modelo de clasificación binaria usando sigmoid.
Fórmula: $P(y=1) = \sigma(X\theta)$
Loss: Binary Cross-Entropy.

Loss Function
Definición: Función que mide error entre predicción y valor real.
Ejemplos: MSE (regresión), Cross-Entropy (clasificación).
Objetivo: Minimizar durante entrenamiento.

M

Matrix Multiplication
Definición: Operación (m×n) @ (n×p) → (m×p).
Elemento: C[i,j] = Σₖ A[i,k] · B[k,j]
Uso: Transformaciones lineales, capas de red.

Mini-batch
Definición: Subconjunto de datos usado en una iteración de SGD.
Ventaja: Balance entre eficiencia y estabilidad.

MLP (Multilayer Perceptron)
Definición: Red neuronal fully-connected con capas ocultas.
Arquitectura: Input → Hidden(s) → Output.

MNIST
Definición: Dataset de dígitos escritos a mano (28 × 28 píxeles).
Tamaño: 60k train, 10k test.
Uso: Benchmark clásico de clasificación de imágenes.

MSE (Mean Squared Error)
Definición: Promedio de errores al cuadrado.
Fórmula: MSE = (1/n) Σ(y - ŷ)²
Uso: Loss para regresión.

Momentum
Definición: Técnica que acelera SGD acumulando gradientes pasados.
Fórmula: v = β·v + (1-β)·∇L; θ = θ - α·v
Ventaja: Escapa mínimos locales, reduce oscilaciones.

N

Normalization
Definición: Escalar datos a un rango estándar.
Min-Max: x' = (x - min) / (max - min) → [0, 1]
Z-score: x' = (x - μ) / σ → media 0, std 1.

NumPy
Definición: Librería de Python para computación numérica eficiente.
Ventaja: Operaciones vectorizadas (evita loops).
Objeto principal: ndarray (n-dimensional array).

O

One-Hot Encoding
Definición: Representar categoría como vector binario.
Ejemplo: clase 3 de 5 → [0, 0, 0, 1, 0]
Uso: Labels para clasificación multiclase.

Overfitting
Definición: Modelo que memoriza training data pero no generaliza.
Síntoma: Train loss bajo, test loss alto.
Soluciones: Más datos, regularización, dropout, early stopping.

P

Partial Derivative
Definición: Derivada respecto a una variable, tratando otras como constantes.
Notación: ∂f/∂x
Uso: Calcular gradientes en funciones multivariable.

PCA (Principal Component Analysis)

Definición: Reducción dimensional que preserva máxima varianza.
Método: Proyectar datos en eigenvectors principales.
Output: Componentes principales ordenados por varianza explicada.

Precision

Definición: De los predichos positivos, ¿cuántos son correctos?
Fórmula: TP / (TP + FP)
Importancia: Cuando FP es costoso.

Projection

Definición: Mapear un punto a un subespacio (línea, plano).
En PCA: Proyectar datos al espacio de componentes principales.

R

Recall

Definición: De los positivos reales, ¿cuántos capturé?
Fórmula: TP / (TP + FN)
Importancia: Cuando FN es costoso.

Regression

Definición: Predecir un valor continuo.
Ejemplos: Precio de casa, temperatura.

Regularization

Definición: Técnica para prevenir overfitting penalizando complejidad.
L 1 (Lasso): Añade $\lambda \cdot ||\theta||_1$ al loss.
L 2 (Ridge): Añade $\lambda \cdot ||\theta||_2^2$ al loss.

ReLU (Rectified Linear Unit)

Definición: f(x) = max(0 , x)
Derivada: 1 si x > 0 , 0 si x ≤ 0 .
Ventaja: Simple, evita vanishing gradient.

S

SGD (Stochastic Gradient Descent)

Definición: Gradient descent con una muestra (o mini-batch) por update.
Ventaja: Más rápido, escapa mínimos locales.
Desventaja: Updates ruidosos.

Sigmoid

Definición: $\sigma(x) = 1 / (1 + e^{-x})$
Rango: (0 , 1)
Uso: Clasificación binaria, probabilidades.
Derivada: $\sigma(x) \cdot (1 - \sigma(x))$

Silhouette Score

Definición: Métrica de calidad de clustering.
Rango: [- 1 , 1], mayor es mejor.
Cálculo: Basado en cohesión intra-cluster y separación inter-cluster.

Softmax

Definición: Convierte vector a distribución de probabilidad.
Fórmula: $\text{softmax}(z)_i = e^{z_i} / \Sigma_j e^{z_j}$
Uso: Capa de salida para clasificación multiclase.

Supervised Learning

Definición: Aprender de datos con etiquetas (X, y).
Tareas: Clasificación, Regresión.

SVD (Singular Value Decomposition)

Definición: Factorización $A = U\Sigma V^T$.
Uso: PCA (más estable), compresión, sistemas de recomendación.

T

Tanh
Definición: Tangente hiperbólica, similar a sigmoid pero centrada en 0.
Rango: (-1, 1)
Derivada: 1 - tanh²(x)

Test Set
Definición: Datos reservados para evaluación final del modelo.
Regla: NUNCA usar para entrenar o seleccionar hiperparámetros.

Training Set
Definición: Datos usados para entrenar el modelo.
Típico: 70-80% del dataset total.

Transpose
Definición: Intercambiar filas y columnas de una matriz.
Notación: A^T
Propiedad: $(AB)^T = B^TA^T$

U

Underfitting
Definición: Modelo demasiado simple que no captura patrones.
Síntoma: Train loss alto, test loss alto.
Soluciones: Modelo más complejo, más features, más entrenamiento.

Unsupervised Learning
Definición: Aprender de datos sin etiquetas.
Tareas: Clustering, reducción dimensional, detección de anomalías.

V

Validation Set
Definición: Datos para ajustar hiperparámetros y detectar overfitting.
Típico: 10-20% del training data.

Variance (estadística)
Definición: Medida de dispersión de los datos.
Fórmula: $\text{Var}(X) = E[(X - \mu)^2]$

Variance (ML)
Definición: Error por sensibilidad a fluctuaciones en training data.
Alta varianza: Overfitting.

Vectorization
Definición: Reemplazar loops por operaciones de arrays.
Ventaja: 10-100x más rápido con NumPy.
Ejemplo: `np.dot(a, b)` en lugar de `sum(a[i]*b[i] for i in range(n))`

W

Weight
Definición: Parámetro aprendido que determina importancia de input.
En redes: Matriz W en $z = Wx + b$.

X

Xavier Initialization
Definición: Inicializar pesos con varianza 1/n_inputs.
Fórmula: $W \sim N(0, 1/n_{in})$ o $U(-\sqrt{1/n_{in}}, \sqrt{1/n_{in}})$
Uso: Capas con tanh/sigmoid.

XOR Problem
Definición: Problema no linealmente separable clásico.
Importancia: Demuestra necesidad de capas ocultas en redes neuronales.
Solución: MLP con al menos una capa oculta.