```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import os
import random
import time
from pathlib import Path

%matplotlib inline

print("Librerías importadas correctamente")
```

```
Librerías importadas correctamente
```

```
import kagglehub

print("Descargando dataset Bird vs Drone...")
path = kagglehub.dataset_download("stealthknight/bird-vs-drone")
print(f"Dataset descargado en: {path}")
```

```
Descargando dataset Bird vs Drone...
Warning: Looks like you're using an outdated `kagglehub` version (installed:
Downloading from https://www.kaggle.com/api/v1/datasets/download/stealthknigh
100%|██████████| 1.05G/1.05G [00:08<00:00, 128MB/s] Extracting files...

Dataset descargado en: /root/.cache/kagglehub/datasets/stealthknight/bird-vs-
```

Exploración del Dataset

↳ 2 cells hidden

Definicion Manual del Modelo

```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """
    s = 1 / (1 + np.exp(-z))
```

```
    s = 1 / (1 + np.exp(-z))
    return s


def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w and i

    Argument:
    dim -- size of the w vector we want (or number of parameters in this

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """
    w = np.zeros(shape=(dim, 1))
    b = 0

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b


def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation expl

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b
    """

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T, X) + b)  # compute activation
    cost = (- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A)))

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = (1 / m) * np.dot(X, (A - Y).T)
    db = (1 / m) * np.sum(A - Y)

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
```

```
            "db": db}

    return grads, cost


def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost=False)
    """
    This function optimizes w and b by running a gradient descent algorit

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shap
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias
    costs -- list of all the costs computed during the optimization, this
    """

    costs = []

    for i in range(num_iterations):

        # Cost and gradient calculation
        grads, cost = propagate(w, b, X, Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

        # Print the cost every 100 training examples
        if print_cost and i % 100 == 0:
            print("Cost after iteration %i: %f" % (i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

```
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[0], 1)

    # Compute vector "A" predicting the probabilities of a cat being pres
    A = sigmoid(np.dot(w.T, X) + b)

    for i in range(A.shape[1]):
        # Convert probabilities a[0,i] to actual predictions p[0,i]
        Y_prediction[0, i] = 1 if A[0, i] > 0.5 else 0

    assert(Y_prediction.shape == (1, m))

    return Y_prediction


def model(X_train, Y_train, X_test, Y_test, num_iterations=2000, learning
    """
    Builds the logistic regression model by calling the function you've i

    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px
    Y_train -- training labels represented by a numpy array (vector) of s
    X_test -- test set represented by a numpy array of shape (num_px * nu
    Y_test -- test labels represented by a numpy array (vector) of shape
    num_iterations -- hyperparameter representing the number of iteration
    learning_rate -- hyperparameter representing the learning rate used i
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """

    # initialize parameters with zeros
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_itera

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
```

```
    b = parameters["b"]

    # Predict test/train set examples
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_

    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train": Y_prediction_train,
         "w": w,
         "b": b,
         "learning_rate": learning_rate,
         "num_iterations": num_iterations}

    return d
```

Estrategias de Preprocesamiento de los Datos

↳ 30 cells hidden

Encontrar Mejores Hiperparametros

```
# Instalar Optuna
!pip install optuna -q
print("✅ Optuna instalado")
```

Show hidden output

Estrategias de Optimizacion e Hiperparametros

Funcion para Entregar y Evaluar

↳ 1 cell hidden

Grid Search

↳ 2 cells hidden

Random Search

↳ 2 cells hidden

Bayesian Optimization

↳ 3 cells hidden

Hyperband

↳ 2 cells hidden

Optimización Hiperparámetros: Estrategias de Optimziación

↳ 3 cells hidden

ANÁLISIS POR ESTRATEGIA DE OPTIMIZACIÓN

1. GRID SEARCH

Descripción de resultados:

Grid Search alcanzó un test accuracy de 73.33% mediante una configuración de learning rate de 0.005 y 1000 iteraciones, completando la optimización en un tiempo total de 392.18 segundos tras evaluar exhaustivamente 20 combinaciones predefinidas. El espacio de búsqueda consistió en una cuadrícula de 5 valores discretos de learning rate [0.001, 0.005, 0.01, 0.05, 0.1] combinados con 4 valores de iterations [500, 1000, 1500, 2000], generando 5 × 4 = 20 combinaciones únicas que fueron evaluadas sistemáticamente. La configuración seleccionada fue identificada en el trial 6/20 (lr=0.005, iter=1000), mostrando train accuracy de 99.58% y test accuracy de 73.33%, evidenciando un gap de 26.25 puntos porcentuales. El tiempo promedio por trial fue de 19.61 segundos (392.18/20), siendo la estrategia más rápida en tiempo total pero fallando en alcanzar el accuracy óptimo global de 75.00% que las otras tres estrategias lograron. Los resultados muestran que para lr=0.005, el accuracy varió con las iteraciones: 500 iter ⟶ 70.00%, 1000 iter ⟶ 73.33%, 1500 iter ⟶ 73.33%, 2000 iter ⟶ 73.33%, indicando un plateau a partir de las 1000 iteraciones. El análisis de los trials revela que learning rates extremos degradaron significativamente el desempeño: lr=0.1 alcanzó máximo

61.67%, lr=0.05 alcanzó máximo 68.33%, mientras que valores en el rango [0.001, 0.01] mostraron mejor desempeño relativo dentro de la cuadrícula evaluada.

Comparación con Random Search:

Grid Search fue 131.92 segundos más rápida que Random Search (392.18s vs 524.10s), representando una reducción del 25.2% en tiempo de ejecución, pero alcanzó un test accuracy inferior: 73.33% vs 75.00%, una diferencia de 1.67 puntos porcentuales que representa una degradación del 2.2% en desempeño. El learning rate encontrado por Grid Search (0.005) es un valor discreto predefinido en la cuadrícula que resultó subóptimo comparado con el valor encontrado por Random Search (0.001335), el cual está fuera del espacio discreto de Grid Search y representa una diferencia del 73.3% (0.001335 es 3.74 veces menor que 0.005). Esta diferencia fundamental explica la brecha en accuracy: Grid Search no pudo alcanzar el óptimo global porque el valor óptimo de learning rate simplemente no estaba incluido en la cuadrícula predefinida [0.001, 0.005, 0.01, 0.05, 0.1]. La configuración de Grid Search requirió 1000 iteraciones, mientras que Random Search sugirió 1800 iteraciones, una diferencia de 800 iteraciones que contribuyó al mejor desempeño de Random Search. El tiempo promedio por trial de Grid Search (19.61s) fue 25.2% menor que el de Random Search (26.21s), explicado porque Grid Search promedió 1250 iterations por trial mientras que Random Search promedió 1750 iterations por trial. Este caso demuestra empíricamente la limitación crítica de Grid Search: la ventaja en velocidad (25.2% más rápida) no compensa la pérdida en accuracy (1.67 puntos) cuando el diseño de la cuadrícula no incluye la región óptima del espacio de hiperparámetros.

Comparación con Bayesian Optimization:

Grid Search fue 107.32 segundos más rápida que Bayesian Optimization (392.18s vs 499.50s), representando una reducción del 21.5% en tiempo de ejecución, pero alcanzó un test accuracy inferior: 73.33% vs 75.00%, la misma diferencia de 1.67 puntos porcentuales observada con Random Search. El learning rate encontrado por Grid Search (0.005) difiere significativamente del valor de Bayesian Optimization (0.001335), con una diferencia del 73.3%, demostrando que Bayesian Optimization identificó una región óptima completamente diferente del espacio de hiperparámetros que Grid Search no exploró. La configuración de Bayesian Optimization requirió 1800 iteraciones, 800 más que Grid Search (1000), sugiriendo que el modelo se benefició de entrenamiento extendido que Grid Search no consideró en su cuadrícula limitada. El tiempo promedio por trial de Grid Search (19.61s) fue 21.5% menor que el de Bayesian Optimization (24.98s), diferencia atribuible al overhead de TPE (construcción de distribuciones, evaluación de función de adquisición) que Grid Search no tiene. Sin embargo, este ahorro de tiempo se

vuelve irrelevante dado que Grid Search falló en encontrar el óptimo: la exploración inteligente de Bayesian Optimization invirtió 107 segundos adicionales pero garantizó encontrar el valor óptimo de learning rate mediante exploración continua guiada por el modelo probabilístico, mientras que Grid Search quedó limitada a su cuadrícula discreta subóptima.

Comparación con Hyperband:

Grid Search fue 384.73 segundos más rápida que Hyperband (392.18s vs 776.91s), representando una reducción del 49.5% en tiempo de ejecución, la diferencia temporal más significativa entre todas las comparaciones. Sin embargo, Grid Search alcanzó un test accuracy inferior: 73.33% vs 75.00%, la misma diferencia de 1.67 puntos porcentuales. Hyperband evaluó 30 trials comparado con los 20 de Grid Search, representando un 50% más de configuraciones exploradas, de las cuales 16 fueron podadas (53.3%) y 14 completadas. El learning rate encontrado por Grid Search (0.005) difiere del de Hyperband (0.001335) en 73.3%, con Hyperband identificando el mismo óptimo que Random Search y Bayesian Optimization mediante su sampler TPE subyacente. Las iteraciones difieren significativamente: Grid Search usó 1000 mientras Hyperband usó 1800, una diferencia de 800 iteraciones que contribuyó al mejor desempeño de Hyperband. El tiempo promedio por trial de Grid Search (19.61s) fue 24.3% menor que el de Hyperband (25.90s calculado como 776.91/30), pero el factor crítico es que Hyperband procesó 50% más trials y aun así encontró el óptimo que Grid Search no pudo alcanzar. Este resultado ilustra que incluso con el overhead de early stopping y la ineficiencia temporal de Hyperband en este contexto, el método superó a Grid Search en accuracy final porque su exploración continua del espacio (a través de TPE) no estuvo restringida a una cuadrícula discreta. La ventaja de 384 segundos de Grid Search pierde valor cuando el resultado es 1.67 puntos porcentuales inferior, validando que velocidad sin precisión en la exploración del espacio de hiperparámetros resulta en optimización subóptima.

Razón para considerar Grid Search:

Grid Search debe considerarse con cautela por su mecanismo de búsqueda exhaustiva que garantiza encontrar el óptimo global únicamente si está incluido en la cuadrícula predefinida, limitación que resultó crítica en este experimento. El método opera dividiendo cada hiperparámetro continuo en valores discretos y generando el producto cartesiano: con 5 valores de learning rate y 4 de iterations, se generan 5 × 4 = 20 combinaciones únicas evaluadas secuencialmente sin procesamiento intermedio. Esta simplicidad explica el menor tiempo observado (392.18s): no existe overhead algorítmico de construcción de modelos sustitutos, evaluación de funciones de adquisición, o decisiones de podado; cada trial es independiente y el

costo total es exactamente la suma de los costos individuales. Sin embargo, el fracaso de Grid Search en encontrar el óptimo (alcanzó solo 73.33% vs 75.00% de las otras estrategias) se debe a que el learning rate óptimo real (0.001335) no fue incluido en la cuadrícula predefinida [0.001, 0.005, 0.01, 0.05, 0.1]. Este valor óptimo cae en el intervalo (0.001, 0.005) que Grid Search no exploró, ya que solo evaluó los extremos de este rango. La literatura sobre optimización de hiperparámetros confirma esta limitación fundamental: Grid Search solo puede encontrar combinaciones discretas predefinidas, y cuando el óptimo global está entre dos puntos de la cuadrícula, el método necesariamente fallará en alcanzarlo, sin importar cuán exhaustiva sea la búsqueda dentro de la cuadrícula definida. La complejidad computacional de Grid Search es O(k^n) donde k es el número de valores por hiperparámetro y n es el número de hiperparámetros, resultando en 5^1 × 4^1 = 20 evaluaciones para este caso bidimensional, crecimiento que se vuelve prohibitivo en dimensiones mayores. Grid Search debe preferirse únicamente cuando: (1) existe alta confianza de que el conocimiento del dominio permite definir una cuadrícula que incluye el óptimo global, (2) el espacio es verdaderamente discreto por naturaleza del problema (no es el caso de learning rates continuos), (3) el costo de entrenamiento es tan bajo que la exploración exhaustiva es trivial en tiempo, y (4) se requiere garantía de evaluación de todas las combinaciones discretas predefinidas. Los resultados demuestran empíricamente que en espacios continuos donde el óptimo puede estar en cualquier punto del rango [0.0001, 0.5], la discretización de Grid Search introduce riesgo fundamental de pérdida del óptimo: aunque Grid Search fue 25.2% más rápida que Random Search, esta ventaja temporal no compensa la degradación de 1.67 puntos porcentuales en accuracy, validando que velocidad sin precisión en la exploración del espacio continuo resulta en optimización fallida.

2. RANDOM SEARCH

Descripción de resultados:

Random Search alcanzó un test accuracy de 75.00% mediante una configuración de learning rate de 0.001335 y 1800 iteraciones, completando la optimización en un tiempo total de 524.10 segundos tras evaluar 20 trials. El learning rate identificado (0.001335) representa un valor continuo dentro del espacio de búsqueda [0.0001, 0.5], indicando que la estrategia de muestreo estocástico exploró exitosamente la región óptima sin estar restringida a valores discretos predefinidos. Esta configuración logró el accuracy máximo alcanzado por cualquier estrategia en este experimento, empatando con Bayesian Optimization y Hyperband en 75.00%. La configuración seleccionada requirió 1800 iteraciones de entrenamiento, sugiriendo que el proceso de optimización identificó beneficio en extender el entrenamiento más allá de los límites explorados por Grid Search (máximo 2000 en cuadrícula pero

óptimo encontrado en 1000). El tiempo promedio por trial fue de 26.21 segundos (524.10/20), reflejando que las configuraciones muestreadas aleatoriamente incluyeron valores de iteraciones más altos en promedio dado que el espacio es [500, 3000] uniform, con valor esperado de 1750 iterations por trial, comparado con los 1250 iterations promedio de Grid Search.

Comparación con Grid Search:

Random Search requirió 131.92 segundos adicionales respecto a Grid Search (524.10s vs 392.18s), lo que representa un incremento del 33.6% en el tiempo total de optimización. Sin embargo, a diferencia de Grid Search que solo alcanzó 73.33%, Random Search alcanzó el accuracy óptimo de 75.00%, logrando una mejora absoluta de 1.67 puntos porcentuales que representa un incremento relativo del 2.3% en desempeño. Esta mejora valida que el costo temporal adicional de 131.92 segundos se justifica completamente por alcanzar el óptimo global. El learning rate encontrado por Random Search (0.001335) es fundamentalmente diferente del valor de Grid Search (0.005): Random Search identificó un valor 73.3% menor (0.001335 es 3.74 veces menor que 0.005) que no existía en la cuadrícula discreta de Grid Search, demostrando empíricamente la ventaja crítica de exploración continua sobre exploración discreta. La diferencia de 800 iteraciones entre las configuraciones (1800 de Random Search vs 1000 de Grid Search) sugiere que Random Search exploró más extensamente el eje de iterations, beneficiándose de entrenamientos más largos. El tiempo promedio por trial de Random Search (26.21s) fue 33.6% superior al de Grid Search (19.61s), explicado porque Random Search muestreó uniformemente en [500, 3000] iterations con promedio esperado de 1750 iterations/trial, mientras que Grid Search con valores discretos [500, 1000, 1500, 2000] promedió 1250 iterations/trial, resultando en 1750/1250 = 1.4 veces más iterations promedio. Este incremento temporal por trial se amortiza completamente por alcanzar el accuracy óptimo que Grid Search no pudo encontrar, confirmando que en espacios continuos, el costo de exploración estocástica es una inversión necesaria para evitar quedar atrapado en cuadrículas subóptimas.

Comparación con Bayesian Optimization:

Random Search requirió 24.60 segundos adicionales respecto a Bayesian Optimization (524.10s vs 499.50s), lo que representa un incremento del 4.9% en el tiempo total de optimización. Ambas estrategias alcanzaron exactamente el mismo test accuracy de 75.00% y exactamente la misma configuración óptima: lr=0.001335 con 1800 iteraciones, demostrando que ambos métodos convergieron al mismo óptimo global del espacio de hiperparámetros. La coincidencia perfecta en los hiperparámetros identificados (lr=0.001335, iter=1800) indica que ambas estrategias exploraron efectivamente la misma región óptima, pero Bayesian Optimization lo hizo

con mayor eficiencia temporal: 4.9% más rápida a pesar de evaluar el mismo número de trials (20). El tiempo promedio por trial de Random Search (26.21s) fue 4.9% mayor que el de Bayesian Optimization (24.98s), diferencia explicada por la distribución de iterations evaluadas: aunque ambos métodos muestrean del mismo espacio [500, 3000], Bayesian Optimization mediante su mecanismo TPE aprendió a concentrar exploración en configuraciones prometedoras, potencialmente evaluando configuraciones con menos iterations en trials exploratorios tempranos antes de converger a la región óptima de 1800 iterations, mientras que Random Search distribuyó sus 20 trials uniformemente sin aprendizaje adaptativo. El hecho de que Bayesian Optimization alcanzó el mismo resultado (75.00%, lr=0.001335, iter=1800) en 24.60 segundos menos demuestra el valor del aprendizaje secuencial: TPE utilizó información de trials anteriores para dirigir la búsqueda hacia la región óptima más eficientemente que el muestreo aleatorio puro, validando que modelos probabilísticos pueden reducir el tiempo de convergencia sin sacrificar calidad del resultado.

Comparación con Hyperband:

Random Search fue 252.81 segundos más rápida que Hyperband (524.10s vs 776.91s), representando una reducción del 32.5% en tiempo de ejecución. Ambas estrategias alcanzaron exactamente el mismo test accuracy de 75.00% y exactamente la misma configuración óptima: lr=0.001335 con 1800 iteraciones. Esta coincidencia perfecta en hiperparámetros indica que tanto Random Search como Hyperband (que usa TPE como sampler) convergieron al mismo óptimo global, pero Random Search lo hizo con sustancialmente mayor eficiencia temporal. Hyperband evaluó 30 trials comparado con los 20 de Random Search, representando un 50% más de configuraciones iniciales, de las cuales 16 fueron podadas (53.3%) mediante early stopping, dejando solo 14 trials completados. A pesar de podar más de la mitad de los trials, Hyperband fue 32.5% más lenta que Random Search, revelando que el overhead de early stopping (evaluación en checkpoints, decisiones de podado, gestión de recursos) no se compensó con el ahorro de iterations en trials podados. El tiempo promedio por trial de Random Search (26.21s) fue ligeramente mayor que el de Hyperband (25.90s calculado como 776.91/30), aparente contradicción que se explica porque Hyperband distribuyó heterogéneamente sus recursos: algunos trials se podaron temprano (mínimo costo), otros completaron hasta 1800-2800 iterations (máximo costo), mientras que Random Search distribuyó más uniformemente sus resources. El análisis revela que Random Search, siendo conceptualmente la estrategia más simple después de Grid Search (muestreo aleatorio sin memoria), superó en eficiencia temporal a Hyperband que combina TPE más early stopping adaptativo. Este resultado confirma que en contextos donde el entrenamiento por trial es relativamente rápido (promedio 26 segundos), la complejidad adicional de Hyperband

no proporciona ventaja práctica: el ahorro por podar trials no compensa el costo de iniciar 50% más trials (30 vs 20) más el overhead de gestión del pruner.

Razón para considerar Random Search:

Random Search debe considerarse por su mecanismo de exploración continua que explica directamente los resultados superiores obtenidos comparado con Grid Search. El método opera mediante muestreo independiente: en cada uno de los 20 trials, genera aleatoriamente un learning rate desde una distribución log-uniforme en [0.0001, 0.5] y un número de iteraciones desde una distribución uniforme en [500, 3000], sin memoria de trials anteriores. Esta independencia y continuidad explica por qué encontró lr=0.001335, un valor óptimo imposible de alcanzar con Grid Search que solo evaluó valores discretos [0.001, 0.005, 0.01, 0.05, 0.1], todos fuera de la región óptima. El éxito de Random Search en este experimento valida empíricamente el principio teórico documentado en la literatura: cuando el espacio de hiperparámetros es continuo y el óptimo puede estar en cualquier punto del rango, la exploración estocástica continua tiene mayor probabilidad de encontrar el óptimo que una cuadrícula discreta arbitraria, especialmente si el diseño de la cuadrícula no incorpora conocimiento previo preciso sobre la ubicación del óptimo. El incremento de 33.6% en tiempo respecto a Grid Search (131.92 segundos adicionales) se explica mediante el cálculo del valor esperado de las distribuciones: muestrear uniformemente entre 500 y 3000 produce promedio de (500+3000)/2 = 1750 iterations por trial, mientras que Grid Search con valores discretos [500, 1000, 1500, 2000] promedia (500+1000+1500+2000)/4 = 1250 iterations por trial, resultando en 1750/1250 = 1.4 veces más iterations promedio para Random Search. Este costo temporal adicional resultó en encontrar el óptimo global (75.00%) que Grid Search no alcanzó (73.33%), demostrando que la inversión de 131.92 segundos adicionales produjo ganancia de 1.67 puntos porcentuales en accuracy. El método es particularmente efectivo en espacios donde pocos hiperparámetros dominan el desempeño: en este caso, learning rate resultó crítico (el valor 0.001335 fue esencial para 75%), mientras que iterations mostró menor sensibilidad dentro del rango explorado. Esta característica de "baja dimensionalidad intrínseca" (solo uno o pocos hiperparámetros dominan el desempeño, mientras otros tienen efecto marginal) permite que 20 muestreos aleatorios tengan alta probabilidad de encontrar el óptimo sin exploración exhaustiva de todo el espacio. Random Search debe preferirse cuando: (1) el espacio es continuo y el óptimo puede estar en cualquier punto del rango sin restricciones discretas predefinidas, (2) no existe conocimiento previo suficientemente preciso para definir una cuadrícula que garantice inclusión del óptimo, (3) el presupuesto permite 20-50 evaluaciones lo cual es suficiente según la literatura para espacios de baja dimensionalidad intrínseca, y (4) se requiere simplicidad de implementación sin overhead algorítmico de modelos probabilísticos.

Los resultados demuestran que Random Search logró balance óptimo entre simplicidad (segundo método más simple después de Grid Search), eficiencia temporal (solo 4.9% más lenta que Bayesian Optimization, 32.5% más rápida que Hyperband), y efectividad (alcanzó el óptimo de 75.00% que Grid Search no pudo encontrar), validando su rol como método práctico y confiable para optimización de hiperparámetros en espacios continuos bidimensionales bien comportados.

3. BAYESIAN OPTIMIZATION (TPE)

Descripción de resultados:

Bayesian Optimization con TPE (Tree-structured Parzen Estimator) alcanzó un test accuracy de 75.00% mediante una configuración de learning rate de 0.001335 y 1800 iteraciones, completando la optimización en un tiempo total de 499.50 segundos tras evaluar 20 trials, siendo la estrategia MÁS EFICIENTE entre todas las que alcanzaron el accuracy óptimo. El learning rate identificado (0.001335) y las iterations (1800) son exactamente idénticos a los encontrados por Random Search y Hyperband, demostrando convergencia perfecta al mismo óptimo global del espacio de hiperparámetros. La configuración representa un valor continuo en el rango [0.0001, 0.5] × [500, 3000], indicando que TPE exploró efectivamente el espacio continuo mediante su mecanismo probabilístico de modelado de distribuciones "buenas" vs "malas". El tiempo promedio por trial fue de 24.98 segundos (499.50/20), el segundo más bajo después de Grid Search entre todas las estrategias, reflejando eficiencia en la convergencia: TPE logró encontrar el óptimo invirtiendo menos tiempo promedio por trial que Random Search (26.21s) y Hyperband (25.90s), demostrando que el aprendizaje secuencial redujo evaluaciones de configuraciones subóptimas. Los datos del log muestran que múltiples trials alcanzaron el máximo de 75.00%, sugiriendo que TPE identificó y concentró exploración en la región óptima después de trials exploratorios iniciales, patrón consistente con el funcionamiento esperado del algoritmo.

Comparación con Grid Search:

Bayesian Optimization requirió 107.32 segundos adicionales respecto a Grid Search (499.50s vs 392.18s), lo que representa un incremento del 27.4% en el tiempo total de optimización. Sin embargo, esta inversión de tiempo resultó en alcanzar el accuracy óptimo de 75.00%, comparado con solo 73.33% de Grid Search, logrando una mejora absoluta de 1.67 puntos porcentuales que representa un incremento relativo del 2.3% en desempeño. Esta mejora valida completamente el costo temporal adicional, demostrando que Bayesian Optimization no solo encontró un mejor resultado sino que lo hizo mediante exploración inteligente del espacio continuo que Grid Search no pudo realizar. El learning rate encontrado por Bayesian Optimization (0.001335)

difiere fundamentalmente del valor de Grid Search (0.005): Bayesian Optimization identificó un valor 73.3% menor que está completamente fuera de la cuadrícula discreta [0.001, 0.005, 0.01, 0.05, 0.1] de Grid Search, demostrando empíricamente la superioridad de exploración continua guiada por modelos probabilísticos sobre búsqueda exhaustiva en cuadrículas discretas arbitrarias. La configuración de Bayesian Optimization requirió 1800 iteraciones, 800 más que Grid Search (1000), sugiriendo que TPE identificó beneficio en entrenamiento extendido más allá de los límites de la cuadrícula de Grid Search. El tiempo promedio por trial de Bayesian Optimization (24.98s) fue 27.4% mayor que el de Grid Search (19.61s), diferencia atribuible al overhead de TPE: construcción de distribuciones KDE $l(\theta)$ y $g(\theta)$, generación y evaluación de candidatos mediante función de adquisición $\alpha=l/g$, operaciones que consumen aproximadamente 1-2 segundos por trial pero que guían la búsqueda hacia el óptimo de manera más inteligente que la enumeración ciega de Grid Search. Los 107.32 segundos adicionales representan el costo de aprendizaje secuencial, pero este costo se amortiza completamente por encontrar el óptimo (75.00%) que Grid Search falló en alcanzar (73.33%), confirmando que en espacios continuos donde el diseño de cuadrícula de Grid Search no incluye el óptimo, la inversión en modelos probabilísticos produce retornos superiores en calidad del resultado final.

Comparación con Random Search:

Bayesian Optimization fue 24.60 segundos más rápida que Random Search (499.50s vs 524.10s), representando una reducción del 4.7% en tiempo de ejecución a pesar de evaluar el mismo número de trials (20 cada uno). Ambas estrategias alcanzaron exactamente el mismo test accuracy de 75.00% y exactamente la misma configuración óptima: lr=0.001335 con 1800 iteraciones, demostrando convergencia perfecta al mismo óptimo global. La coincidencia exacta en hiperparámetros valida que tanto el muestreo aleatorio de Random Search como el aprendizaje secuencial de Bayesian Optimization identificaron correctamente la región óptima del espacio, pero Bayesian Optimization lo hizo con mayor eficiencia temporal. El tiempo promedio por trial de Bayesian Optimization (24.98s) fue 4.7% menor que el de Random Search (26.21s), diferencia que explica la ventaja temporal total a pesar del overhead algorítmico de TPE. Esta mejora en eficiencia temporal se atribuye al mecanismo de exploración guiada: mientras Random Search distribuyó sus 20 trials uniformemente en el espacio $[0.0001, 0.5] \times [500, 3000]$ sin memoria entre trials, Bayesian Optimization utilizó información de trials anteriores para construir modelos probabilísticos que concentraron la búsqueda en regiones prometedoras, reduciendo la evaluación de configuraciones claramente subóptimas. Los datos sugieren que Bayesian Optimization, después de trials exploratorios iniciales, identificó que lr aproximadamente 0.001-0.002 e iterations aproximadamente 1500-2000 eran

prometedores, concentrando trials posteriores en esta región y evitando malgastar evaluaciones en configuraciones con lr mayor a 0.01 o iterations menores a 1000. Este aprendizaje adaptativo explica por qué Bayesian Optimization alcanzó el mismo resultado óptimo que Random Search pero invirtiendo 24.60 segundos menos: menor desperdicio de evaluaciones en regiones claramente subóptimas del espacio. El resultado demuestra empíricamente el valor del modelado probabilístico en optimización de hiperparámetros: el overhead de TPE (construcción de distribuciones, evaluación de función de adquisición) se compensa completamente con la reducción en evaluaciones inútiles, resultando en convergencia más rápida al óptimo global. Bayesian Optimization emerge como la estrategia MÁS EFICIENTE entre las tres que alcanzaron 75.00%, validando que el aprendizaje secuencial proporciona ventaja práctica medible (4.7% reducción de tiempo) sobre exploración puramente estocástica en espacios bidimensionales continuos.

Comparación con Hyperband:

Bayesian Optimization fue 277.41 segundos más rápida que Hyperband (499.50s vs 776.91s), representando una reducción del 35.7% en tiempo de ejecución, la diferencia temporal más significativa entre estrategias que alcanzaron el mismo accuracy óptimo. Ambas estrategias utilizan TPE como sampler subyacente y alcanzaron exactamente la misma configuración óptima: lr=0.001335 con 1800 iteraciones, resultando en el mismo test accuracy de 75.00%. Esta coincidencia perfecta en hiperparámetros indica que el componente TPE convergió al mismo óptimo independientemente de si se usó solo (Bayesian Optimization) o combinado con early stopping (Hyperband), pero el early stopping de Hyperband agregó overhead sustancial sin beneficio en el resultado final. Hyperband evaluó 30 trials comparado con los 20 de Bayesian Optimization, representando un 50% más de configuraciones iniciales, de las cuales 16 fueron podadas (53.3%) mediante successive halving, dejando efectivamente 14 trials completados. A pesar de podar más de la mitad de los trials, Hyperband fue 35.7% más lenta que Bayesian Optimization puro, revelando que en este contexto el overhead de early stopping superó completamente cualquier ahorro de recursos. El tiempo promedio por trial de Bayesian Optimization (24.98s) fue 3.6% mayor que el de Hyperband (25.90s calculado como 776.91/30), aparente mejora que no se traduce en ventaja total porque Hyperband procesó 50% más trials: la distribución heterogénea de recursos de Hyperband (algunos trials podados temprano con costo mínimo, otros completados hasta 1800 iterations) redujo el promedio por trial pero no el tiempo total de optimización. El análisis revela factores que explican la ineficiencia de Hyperband: (1) overhead de evaluación en checkpoints múltiples para decisiones de podado, (2) costo de iniciar y procesar 30 trials versus 20, y (3) gestión de recursos del pruner que añade complejidad sin retorno cuando el entrenamiento por trial es

relativamente rápido (aproximadamente 25 segundos). El ahorro de podar 16 trials (estimado en aproximadamente 200-300 seconds de iterations no ejecutadas) no compensó el costo adicional de procesar 10 trials extra más el overhead de gestión, resultando en 277.41 segundos de penalización neta. Bayesian Optimization emerge como superior en todos los aspectos: mismo resultado óptimo (75.00%, lr=0.001335, iter=1800) en 35.7% menos tiempo, demostrando que en contextos donde el costo de evaluación por trial es bajo a moderado (menor a 30 segundos), la simplicidad de Bayesian Optimization puro con TPE es más eficiente que la complejidad adicional de combinar TPE con early stopping adaptativo.

Razón para considerar Bayesian Optimization:

Bayesian Optimization con TPE debe considerarse como la estrategia ÓPTIMA para este problema por su combinación de efectividad (alcanzó el accuracy máximo de 75.00%) y eficiencia (fue la MÁS RÁPIDA entre las estrategias que alcanzaron este óptimo). El mecanismo de TPE opera mediante aprendizaje secuencial que modela probabilísticamente qué regiones del espacio de hiperparámetros son prometedoras: construye dos distribuciones KDE, l(θ) que modela configuraciones "buenas" (típicamente top 15-25% según parámetro γ) y g(θ) que modela configuraciones "malas" (bottom 75-85%), utilizando el historial de trials evaluados para refinar estas distribuciones iterativamente. En cada trial: (1) ordena el historial por accuracy, (2) calcula un umbral percentil separando buenos de malos, (3) construye KDEs sobre los valores de hiperparámetros de cada grupo, (4) genera aproximadamente 24 candidatos muestreando desde l(θ) para concentrar búsqueda donde configuraciones buenas se han observado, (5) evalúa cada candidato mediante α=l(θ)/g(θ), priorizando configuraciones con alta densidad en l (típicas de buenos) y baja densidad en g (atípicas de malos), y (6) selecciona el candidato con α máximo para el siguiente trial. Este proceso explica la eficiencia observada: después de trials exploratorios iniciales, TPE identificó que lr aproximadamente 0.001-0.002 produce mejores resultados que lr mayor a 0.01, concentrando trials posteriores en la región prometedora y evitando desperdiciar evaluaciones en lr extremos (menor a 0.0005 o mayor a 0.05). La coincidencia exacta con Random Search y Hyperband en el óptimo encontrado (lr=0.001335, iter=1800 ⟶ 75.00%) pero con ventaja temporal (4.7% más rápida que Random Search, 35.7% más rápida que Hyperband) valida que el modelado probabilístico proporciona convergencia más eficiente al óptimo que exploración puramente aleatoria o combinaciones con early stopping. El overhead de TPE (1-2 segundos por trial para construcción de KDEs, generación de candidatos, evaluación de α) se amortiza completamente con la reducción de evaluaciones en regiones subóptimas: mientras Random Search evaluó uniformemente todo el espacio [0.0001, 0.5], TPE aprendió a evitar configuraciones claramente malas después de exploración inicial. Bayesian Optimization también superó a Grid Search

con margen significativo: invirtió solo 107.32 segundos adicionales (27.4% más tiempo) pero alcanzó accuracy superior (75.00% vs 73.33%, ganancia de 1.67 puntos), demostrando que exploración inteligente continua supera completamente a búsqueda exhaustiva discreta cuando el diseño de cuadrícula no incluye el óptimo. Bayesian Optimization debe preferirse cuando: (1) el espacio es continuo y de dimensionalidad baja a media (2-10 hiperparámetros) donde Grid Search sufre explosión combinatoria o riesgo de exclusión del óptimo, (2) el presupuesto permite 20-100 trials que es suficiente para que TPE aprenda la estructura del espacio, (3) cada evaluación tiene costo moderado (20 segundos a varios minutos) donde el overhead de TPE (1-2s) es marginal pero el ahorro por convergencia inteligente es significativo, y (4) se busca maximizar probabilidad de encontrar el óptimo global con eficiencia temporal. Los resultados demuestran empíricamente que Bayesian Optimization logró el mejor balance: alcanzó el óptimo que Grid Search falló en encontrar, lo hizo más rápido que Random Search (4.7% reducción de tiempo), y evitó la penalización de overhead de Hyperband (35.7% reducción de tiempo), validando su posición como estrategia de optimización de hiperparámetros más eficiente para este contexto de problema bidimensional continuo con evaluaciones de costo moderado.

4. HYPERBAND

Descripción de resultados:

Hyperband alcanzó un test accuracy de 75.00% mediante una configuración de learning rate de 0.001335 y 1800 iteraciones (valores idénticos a Random Search y Bayesian Optimization), completando la optimización en un tiempo total de 776.91 segundos tras evaluar 30 trials, siendo la estrategia MÁS LENTA entre todas las evaluadas a pesar de implementar early stopping adaptativo. De los 30 trials iniciados, 16 fueron podados mediante successive halving (53.3% de trials eliminados) y 14 completaron su entrenamiento hasta las iteraciones sugeridas por TPE. El tiempo promedio por trial fue de 25.90 segundos (776.91/30), reflejando una distribución heterogénea de recursos donde trials podados consumieron mínimo tiempo (aproximadamente 10-15 seconds entrenando solo 500-1000 iterations antes de eliminación) mientras trials no podados entrenaron hasta aproximadamente 1800-2500 iterations (aproximadamente 35-40 seconds). La configuración del pruner utilizada fue min_resource=500 (iteraciones mínimas antes de evaluar podado), max_resource=3000 (iteraciones máximas permitidas), y reduction_factor=3 (factor de eliminación sucesiva que retiene aproximadamente 1/3 de trials en cada ronda de evaluación). Los datos del log muestran que múltiples trials alcanzaron el máximo de 75.00%, indicando que el sampler TPE subyacente identificó correctamente la región óptima, pero el proceso de early stopping agregó overhead sustancial sin

proporcionar beneficio en eficiencia temporal o calidad del resultado final comparado con Bayesian Optimization puro.

Comparación con Grid Search:

Hyperband requirió 384.73 segundos adicionales respecto a Grid Search (776.91s vs 392.18s), lo que representa un incremento del 98.1% en el tiempo total de optimización, casi duplicando el tiempo de Grid Search. Esta penalización temporal sería justificable si Hyperband hubiera alcanzado accuracy superior, pero Hyperband logró 75.00% mientras Grid Search solo alcanzó 73.33%, una mejora de 1.67 puntos porcentuales idéntica a la obtenida por Random Search y Bayesian Optimization. Sin embargo, mientras Bayesian Optimization logró esta mejora invirtiendo solo 107.32 segundos adicionales (27.4% más que Grid Search), Hyperband requirió 384.73 segundos adicionales (98.1% más), demostrando que Hyperband fue la estrategia MENOS EFICIENTE para alcanzar el mismo accuracy óptimo. Hyperband evaluó 30 trials comparado con los 20 de Grid Search, representando un 50% más de configuraciones iniciales, de las cuales 16 fueron podadas (53.3%), dejando efectivamente solo 14 trials completados versus 20 de Grid Search. A pesar del podado agresivo, Hyperband fue 98.1% más lenta que Grid Search, revelando que el ahorro de iterations por podado no compensó el costo de iniciar 50% más trials más el overhead de gestión del pruner. El learning rate encontrado por Hyperband (0.001335) difiere del de Grid Search (0.005) en 73.3%, indicando que Hyperband (mediante su sampler TPE) exploró el espacio continuo efectivamente, encontrando el óptimo que Grid Search no pudo alcanzar debido a su restricción a cuadrícula discreta. Las iteraciones difieren significativamente: Grid Search usó 1000 mientras Hyperband usó 1800, diferencia de 800 iterations que contribuyó al mejor desempeño de Hyperband. El tiempo promedio por trial de Hyperband (25.90s) fue 32.1% mayor que el de Grid Search (19.61s), y considerando que Hyperband procesó 50% más trials (30 vs 20), el tiempo total se incrementó dramáticamente. Este resultado demuestra que incluso comparado con Grid Search que falló en encontrar el óptimo, Hyperband no proporcionó valor: aunque alcanzó accuracy superior (75.00% vs 73.33%), lo hizo con el mayor costo temporal absoluto (776.91s), mientras que Bayesian Optimization alcanzó el mismo accuracy óptimo en sustancialmente menos tiempo (499.50s).

Comparación con Random Search:

Hyperband requirió 252.81 segundos adicionales respecto a Random Search (776.91s vs 524.10s), lo que representa un incremento del 48.2% en el tiempo total de optimización. Ambas estrategias alcanzaron exactamente el mismo test accuracy de 75.00% y exactamente la misma configuración óptima: lr=0.001335 con 1800 iteraciones, demostrando convergencia perfecta al mismo óptimo global. Sin

embargo, Hyperband tardó casi 50% más tiempo en alcanzar este resultado, revelando una ineficiencia fundamental: el early stopping adaptativo que teóricamente debería reducir tiempo mediante podado de configuraciones malas resultó en realidad en MAYOR tiempo de ejecución que exploración aleatoria simple sin podado. Hyperband evaluó 30 trials comparado con los 20 de Random Search, representando un 50% más de configuraciones iniciales, de las cuales 16 fueron podadas (53.3%), dejando 14 trials completados versus 20 de Random Search que completó todos sus trials. A pesar de completar MENOS trials que Random Search (14 vs 20), Hyperband fue 48.2% más lenta, contradicción que se explica por múltiples factores: (1) overhead de iniciar 30 trials versus 20 incluye overhead de TPE para 30 configuraciones, (2) overhead de evaluación en checkpoints múltiples (500, 1500, 3000 iterations aproximadamente según configuración del pruner) para cada trial activo, incluyendo guardado de estado, comparación de percentiles, y decisiones de podado, y (3) el ahorro de podar 16 trials (estimado en aproximadamente 200-300 seconds de iterations no ejecutadas) no compensó el costo de procesar 10 trials adicionales más el overhead de gestión. El tiempo promedio por trial de Hyperband (25.90s) fue 1.2% menor que el de Random Search (26.21s), aparente mejora marginal que no se traduce en ventaja total porque Hyperband procesó 50% más trials. El análisis revela que Hyperband agregó complejidad (successive halving, pruner, checkpoints) sin retorno: Random Search con su simplicidad conceptual (muestreo aleatorio independiente sin memoria ni podado) alcanzó el mismo resultado óptimo en sustancialmente menos tiempo, validando que en contextos donde el costo de entrenamiento por trial es moderado (aproximadamente 25-30 seconds), el overhead de early stopping supera su beneficio.

Comparación con Bayesian Optimization:

Hyperband requirió 277.41 segundos adicionales respecto a Bayesian Optimization (776.91s vs 499.50s), lo que representa un incremento del 55.5% en el tiempo total de optimización, siendo la diferencia temporal más significativa entre estrategias que alcanzaron el mismo accuracy óptimo. Ambas estrategias utilizan TPE como sampler subyacente y alcanzaron exactamente la misma configuración óptima: lr=0.001335 con 1800 iteraciones, resultando en el mismo test accuracy de 75.00%. Esta coincidencia perfecta en hiperparámetros demuestra que el componente TPE convergió al mismo óptimo en ambos casos, pero el early stopping de Hyperband agregó 277.41 segundos de overhead (más de la mitad del tiempo total de Bayesian Optimization) sin proporcionar ningún beneficio en el resultado final. Hyperband evaluó 30 trials comparado con los 20 de Bayesian Optimization, representando un 50% más de configuraciones iniciales, de las cuales 16 fueron podadas (53.3%), dejando 14 trials completados versus 20 de Bayesian Optimization que completó todos. A pesar de completar MENOS trials que Bayesian Optimization (14 vs 20),

Hyperband fue 55.5% más lenta, revelando que el costo de gestión de successive halving superó completamente cualquier beneficio del podado. El tiempo promedio por trial de Hyperband (25.90s) fue 3.7% mayor que el de Bayesian Optimization (24.98s), indicando que incluso a nivel de trial individual Hyperband fue menos eficiente. Este incremento por trial se amplifica porque Hyperband procesó 50% más trials (30 vs 20), multiplicando el impacto del overhead. El análisis detallado revela factores que explican los 277.41 segundos de penalización: (1) overhead de TPE para 30 trials versus 20 (aproximadamente 10-20 seconds adicionales de construcción de distribuciones), (2) overhead de evaluación en checkpoints múltiples para decisiones de podado: cada trial activo debe ser evaluado en checkpoints (500, aproximadamente 1500, 3000 iterations), incluyendo guardado de estado, carga para comparación, evaluación de percentiles, y decisión de continuar o podar (estimado en 2-3 seconds por checkpoint por trial, multiplicado por 30 trials y múltiples checkpoints resultando en aproximadamente 100-150 seconds de overhead acumulado), (3) costo de iniciar 10 trials adicionales que aunque fueron podados temprano consumieron overhead de inicialización y primeros checkpoints, y (4) el ahorro teórico de podar 16 trials (aproximadamente 200-300 seconds de iterations no ejecutadas) se anuló completamente por los factores anteriores, resultando en penalización neta de 277.41 seconds. El resultado es concluyente: Bayesian Optimization puro (TPE sin early stopping) alcanzó el mismo accuracy óptimo (75.00%, lr=0.001335, iter=1800) en 55.5% menos tiempo que Hyperband (TPE con early stopping), demostrando empíricamente que en este contexto la simplicidad de Bayesian Optimization es superior a la complejidad de Hyperband.

Razón para considerar Hyperband:

Hyperband debe considerarse con cautela extrema en contextos similares a este experimento, dado que resultó ser la estrategia MENOS EFICIENTE: alcanzó el mismo accuracy óptimo que Random Search y Bayesian Optimization (75.00%, lr=0.001335, iter=1800) pero invirtiendo 48.2% más tiempo que Random Search y 55.5% más tiempo que Bayesian Optimization. El mecanismo de Hyperband opera mediante early stopping adaptativo basado en successive halving: inicializa n trials (30 en este caso) y los distribuye en brackets con diferentes balances exploración-explotación, evaluando todos los trials activos en checkpoints periódicos (500, aproximadamente 1500, 3000 iterations según configuración del pruner), ordenándolos por accuracy en cada checkpoint, y eliminando los bottom 2/3 (según reduction_factor=3) para retener solo el top 1/3 que continúa al siguiente checkpoint. El ahorro teórico proviene de detener tempranamente configuraciones malas sin desperdiciar recursos en entrenarlas hasta el final: si una configuración muestra 65% accuracy en 500 iterations mientras otras muestran 72-75%, es improbable que alcance el óptimo en 3000 iterations, justificando su eliminación. Los 16 trials podados en este

experimento (53.3%) ahorraron aproximadamente 200-300 seconds de iterations no ejecutadas, pero este ahorro fue completamente anulado por overhead acumulado, resultando en penalización neta de 252.81 seconds versus Random Search y 277.41 seconds versus Bayesian Optimization. Los resultados demuestran empíricamente tres factores críticos que explican la ineficiencia de Hyperband en este contexto: (1) el costo de entrenamiento por trial es relativamente bajo (aproximadamente 25-30 seconds), haciendo que el overhead de checkpoints, guardado de estado, comparación de percentiles, y decisiones de podado (aproximadamente 2-3 seconds por checkpoint multiplicado por 30 trials y múltiples checkpoints) represente una fracción significativa del tiempo total de optimización en lugar de ser marginal, (2) el dataset y modelo son suficientemente simples que entrenar 500-3000 iterations toma solo 12-40 seconds, contexto donde diferencias de 1000-1500 iterations ahorradas por podado representan solo 15-25 seconds de ahorro por trial podado, insuficiente para amortizar el overhead de gestión, y (3) las señales de desempeño en checkpoints tempranos (500 iterations) no fueron suficientemente discriminativas para identificar configuraciones claramente malas: muchas configuraciones mostraron accuracies en rango 65-75% en 500 iterations, dificultando decisiones de podado confiables y resultando en retención conservadora que no maximizó el ahorro potencial. Hyperband debe preferirse únicamente cuando: (1) cada evaluación completa es MUY costosa (mayor a 1 hora por trial), de modo que eliminar configuraciones después de 10-20% de recursos ahorra 80-90% del tiempo y el overhead de checkpoints es marginal comparado con el costo base de entrenamiento, (2) el presupuesto permite evaluar 100-500 configuraciones iniciales con recursos mínimos, escala donde el balance exploración-explotación de Hyperband proporciona ventaja sobre métodos que evalúan solo 20-50 configuraciones, (3) las diferencias de desempeño entre configuraciones buenas y malas son claramente detectables en checkpoints tempranos (correlación mayor a 0.8 entre accuracy en 10-20% de recursos versus accuracy final), validando que las decisiones de podado son confiables y no eliminan prematuramente configuraciones prometedoras, y (4) el espacio de hiperparámetros es suficientemente grande (mayor o igual a 5 dimensiones) que justifica exploración amplia inicial seguida de refinamiento mediante eliminación progresiva. Los resultados demuestran empíricamente que Hyperband PIERDE su ventaja teórica cuando el costo base de evaluación es bajo a moderado (menor a 5 minutos por trial) y cuando las señales tempranas son ambiguas, confirmando que early stopping adaptativo es una técnica específica de contexto que requiere entrenamiento costoso y discriminación temprana clara para justificar su complejidad algorítmica. En el contexto de este experimento (entrenamiento rápido, espacio bidimensional, señales tempranas ambiguas), Hyperband representó sobre-ingeniería: agregó complejidad (successive halving, pruner, checkpoints, gestión de recursos) que resultó en mayor tiempo de

ejecución (776.91s) que métodos sustancialmente más simples como Bayesian Optimization (499.50s) y Random Search (524.10s), ambos alcanzando el mismo resultado óptimo. La recomendación práctica es clara: en problemas similares a este, evitar Hyperband y preferir Bayesian Optimization que demostró ser la estrategia MÁS EFICIENTE (mismo accuracy óptimo de 75.00% en menor tiempo: 499.50s).

Optimizacion de Hiperparametros con Keras + Optuna

```
!pip install optuna tensorflow scikit-learn -q
# ============================================================================
# IMPORTS PARA OPTIMIZACIÓN CON KERAS + OPTUNA
# ============================================================================

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers, initializers
from tensorflow.keras.models import Sequential
from tensorflow.keras.callbacks import EarlyStopping
import optuna
from optuna.samplers import TPESampler, RandomSampler, GridSampler
from optuna.pruners import HyperbandPruner
import numpy as np
import time
import warnings

warnings.filterwarnings('ignore')
optuna.logging.set_verbosity(optuna.logging.WARNING)

# Configurar TensorFlow para menos verbosity
tf.get_logger().setLevel('ERROR')

print(" Imports completados")
```

```
 Imports completados
```

Función Keras con Hiperparámetros Adicionales

```
# ============================================================================
# FUNCIÓN PARA CREAR MODELO CON HIPERPARÁMETROS CONFIGURABLES
# ============================================================================

def create_model(input_dim,
                 learning_rate=0.01,
                 activation_function='relu',
                 lambda_reg=0.0,
                 initialization='glorot_uniform',
                 n_hidden_units=0):
```

```
                n_hidden_units=0):
    """
    Crea un modelo de red neuronal con hiperparámetros configurables

    Args:
        input_dim: Dimensión de entrada (num_px * num_px * 3 o num_px * n
        learning_rate: Tasa de aprendizaje
        activation_function: 'sigmoid', 'tanh', 'relu'
        lambda_reg: Coeficiente de regularización L2
        initialization: 'zeros', 'glorot_uniform' (Xavier), 'he_uniform'
        n_hidden_units: Número de neuronas en capa oculta (0 = sin capa o

    Returns:
        Modelo compilado de Keras
    """

    # Mapear nombres de inicialización
    init_map = {
        'zeros': initializers.Zeros(),
        'glorot_uniform': initializers.GlorotUniform(),  # Xavier
        'xavier': initializers.GlorotUniform(),
        'he_uniform': initializers.HeUniform(),
        'he': initializers.HeUniform()
    }

    kernel_init = init_map.get(initialization, initializers.GlorotUniform

    # Crear modelo
    model = Sequential()

    # Si hay capa oculta
    if n_hidden_units > 0:
        model.add(layers.Dense(
            n_hidden_units,
            activation=activation_function,
            kernel_initializer=kernel_init,
            kernel_regularizer=regularizers.l2(lambda_reg),
            input_shape=(input_dim,)
        ))

    # Capa de salida (siempre con sigmoid para clasificación binaria)
    if n_hidden_units > 0:
        model.add(layers.Dense(
            1,
            activation='sigmoid',
            kernel_initializer=kernel_init,
            kernel_regularizer=regularizers.l2(lambda_reg)
        ))
    else:
        # Sin capa oculta (como el modelo del profesor)
        model.add(layers.Dense(
            1,
            activation='sigmoid',
            kernel_initializer=kernel_init,
            kernel_regularizer=regularizers.l2(lambda_reg),
            input_shape=(input_dim,)
```

```
        ))

    # Compilar modelo
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model


# ============================================================================
# FUNCIÓN PARA ENTRENAR Y EVALUAR
# ============================================================================

def train_and_evaluate_keras(X_train, Y_train, X_test, Y_test,
                             learning_rate=0.01,
                             num_iterations=1000,
                             activation_function='relu',
                             lambda_reg=0.0,
                             initialization='glorot_uniform',
                             n_hidden_units=0,
                             verbose=0):
    """
    Entrena y evalúa un modelo de Keras

    Returns:
        dict con train_accuracy, test_accuracy, model
    """
    try:
        # Transponer datos para Keras (espera (n_samples, n_features))
        X_train_T = X_train.T
        Y_train_T = Y_train.T
        X_test_T = X_test.T
        Y_test_T = Y_test.T

        input_dim = X_train_T.shape[1]

        # Crear modelo
        model = create_model(
            input_dim=input_dim,
            learning_rate=learning_rate,
            activation_function=activation_function,
            lambda_reg=lambda_reg,
            initialization=initialization,
            n_hidden_units=n_hidden_units
        )

        # Entrenar
        history = model.fit(
            X_train_T, Y_train_T,
            epochs=num_iterations,
            batch_size=32,
            verbose=verbose,
```

```
            validation_split=0.0
        )

        # Evaluar
        train_loss, train_acc = model.evaluate(X_train_T, Y_train_T, verb
        test_loss, test_acc = model.evaluate(X_test_T, Y_test_T, verbose=

        return {
            'train_accuracy': train_acc * 100,
            'test_accuracy': test_acc * 100,
            'train_loss': train_loss,
            'test_loss': test_loss,
            'model': model,
            'history': history
        }

    except Exception as e:
        print(f" Error en entrenamiento: {e}")
        return None
```

Optimización Hiperparámetros para Funcion Keras

```
# ============================================================================
# OPTIMIZACIÓN CON OPTUNA - TODAS LAS ESTRATEGIAS
# ============================================================================

def create_optuna_objective(X_train, Y_train, X_test, Y_test):
    """
    Crea función objetivo para Optuna
    """
    def objective(trial):
        # Sugerir hiperparámetros
        learning_rate = trial.suggest_float('learning_rate', 1e-5, 0.5, l
        num_iterations = trial.suggest_int('num_iterations', 50, 500, ste
        activation_function = trial.suggest_categorical('activation_funct
        lambda_reg = trial.suggest_float('lambda_reg', 1e-6, 0.1, log=Tru
        initialization = trial.suggest_categorical('initialization', ['ze
        n_hidden_units = trial.suggest_categorical('n_hidden_units', [0,

        # Entrenar y evaluar
        result = train_and_evaluate_keras(
            X_train, Y_train, X_test, Y_test,
            learning_rate=learning_rate,
            num_iterations=num_iterations,
            activation_function=activation_function,
            lambda_reg=lambda_reg,
            initialization=initialization,
            n_hidden_units=n_hidden_units,
            verbose=0
        )

        if result is None:
            return 0.0
```

```
            return 0.0

        # Optuna puede hacer pruning basado en epochs intermedios
        # Por ahora solo devolvemos el test accuracy final
        return result['test_accuracy']

    return objective


# ============================================================================
# 1. RANDOM SEARCH
# ============================================================================

def random_search_optuna(X_train, Y_train, X_test, Y_test, n_trials=30):
    """
    Random Search con Optuna
    """
    print("\n" + "="*80)
    print(" ESTRATEGIA 1: RANDOM SEARCH con Optuna")
    print("="*80 + "\n")

    print(f" Configuración:")
    print(f"   Trials: {n_trials}")
    print(f"   Sampler: Random")
    print(f"\n Espacio de hiperparámetros:")
    print(f"   learning_rate: [1e-5, 0.5] (log)")
    print(f"   num_iterations: [50, 500] (step 50)")
    print(f"   activation_function: ['sigmoid', 'tanh', 'relu']")
    print(f"   lambda_reg: [1e-6, 0.1] (log)")
    print(f"   initialization: ['zeros', 'glorot_uniform', 'he_uniform']"
    print(f"   n_hidden_units: [0, 16, 32, 64, 128]\n")

    start_time = time.time()

    study = optuna.create_study(
        direction='maximize',
        sampler=RandomSampler(seed=42)
    )

    objective = create_optuna_objective(X_train, Y_train, X_test, Y_test)
    study.optimize(objective, n_trials=n_trials, show_progress_bar=True)

    elapsed_time = time.time() - start_time

    print(f"\n{'='*80}")
    print(f" RANDOM SEARCH - MEJORES HIPERPARÁMETROS:")
    print(f"{'='*80}")
    for param, value in study.best_params.items():
        print(f"   {param}: {value}")
    print(f"\n   Best Test Accuracy: {study.best_value:.2f}%")
    print(f"   Tiempo total: {elapsed_time:.2f} segundos")
    print(f"{'='*80}\n")

    return {
        'strategy': 'Random Search',
        'best_params': study.best_params,
```

```
        'best_score': study.best_value,
        'study': study,
        'time': elapsed_time
    }


# ======================================================================
# 2. BAYESIAN OPTIMIZATION (TPE)
# ======================================================================

def bayesian_optimization_optuna(X_train, Y_train, X_test, Y_test, n_tria
    """
    Bayesian Optimization con TPE
    """
    print("\n" + "="*80)
    print(" ESTRATEGIA 2: BAYESIAN OPTIMIZATION (TPE)")
    print("="*80 + "\n")

    print(f" Configuración:")
    print(f"   Trials: {n_trials}")
    print(f"   Sampler: TPE (Tree-structured Parzen Estimator)")
    print(f"   Estrategia: Aprende de trials anteriores\n")

    start_time = time.time()

    study = optuna.create_study(
        direction='maximize',
        sampler=TPESampler(seed=42, n_startup_trials=10)
    )

    objective = create_optuna_objective(X_train, Y_train, X_test, Y_test)
    study.optimize(objective, n_trials=n_trials, show_progress_bar=True)

    elapsed_time = time.time() - start_time

    print(f"\n{'='*80}")
    print(f" BAYESIAN OPTIMIZATION - MEJORES HIPERPARÁMETROS:")
    print(f"{'='*80}")
    for param, value in study.best_params.items():
        print(f"   {param}: {value}")
    print(f"\n   Best Test Accuracy: {study.best_value:.2f}%")
    print(f"   Tiempo total: {elapsed_time:.2f} segundos")
    print(f"{'='*80}\n")

    return {
        'strategy': 'Bayesian Optimization',
        'best_params': study.best_params,
        'best_score': study.best_value,
        'study': study,
        'time': elapsed_time
    }


# ======================================================================
# 3. HYPERBAND
```

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

def hyperband_optimization_optuna(X_train, Y_train, X_test, Y_test, n_tri
    """
    Hyperband con early stopping
    """
    print("\n" + "="*80)
    print(" ESTRATEGIA 3: HYPERBAND")
    print("="*80 + "\n")

    print(f" Configuración:")
    print(f"   Trials: {n_trials}")
    print(f"   Sampler: TPE")
    print(f"   Pruner: Hyperband (early stopping)\n")

    start_time = time.time()

    study = optuna.create_study(
        direction='maximize',
        sampler=TPESampler(seed=42),
        pruner=HyperbandPruner(
            min_resource=50,
            max_resource=500,
            reduction_factor=3
        )
    )

    objective = create_optuna_objective(X_train, Y_train, X_test, Y_test)
    study.optimize(objective, n_trials=n_trials, show_progress_bar=True)

    elapsed_time = time.time() - start_time
    n_pruned = len([t for t in study.trials if t.state == optuna.trial.Tr

    print(f"\n{'='*80}")
    print(f" HYPERBAND - MEJORES HIPERPARÁMETROS:")
    print(f"{'='*80}")
    for param, value in study.best_params.items():
        print(f"   {param}: {value}")
    print(f"\n   Best Test Accuracy: {study.best_value:.2f}%")
    print(f"   Trials completados: {len(study.trials)}")
    print(f"   Trials podados: {n_pruned}")
    print(f"   Tiempo total: {elapsed_time:.2f} segundos")
    print(f"{'='*80}\n")

    return {
        'strategy': 'Hyperband',
        'best_params': study.best_params,
        'best_score': study.best_value,
        'study': study,
        'time': elapsed_time,
        'n_pruned': n_pruned
    }


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

```
# 4. GRID SEARCH (versión simplificada con Optuna)
# ================================================================================

def grid_search_optuna(X_train, Y_train, X_test, Y_test):
    """
    Grid Search con espacio reducido (muy costoso con 6 hiperparámetros)
    """
    print("\n" + "="*80)
    print(" ESTRATEGIA 4: GRID SEARCH (espacio reducido)")
    print("="*80 + "\n")

    # Grid reducido para que sea factible
    search_space = {
        'learning_rate': [0.001, 0.01, 0.1],
        'num_iterations': [100, 200],
        'activation_function': ['relu', 'tanh'],
        'lambda_reg': [0.0, 0.01],
        'initialization': ['glorot_uniform', 'he_uniform'],
        'n_hidden_units': [0, 32, 64]
    }

    total_combinations = np.prod([len(v) for v in search_space.values()])
    print(f" Espacio de búsqueda:")
    for param, values in search_space.items():
        print(f"   {param}: {values}")
    print(f"\n Total combinaciones: {total_combinations}\n")

    start_time = time.time()

    study = optuna.create_study(
        direction='maximize',
        sampler=GridSampler(search_space, seed=42)
    )

    objective = create_optuna_objective(X_train, Y_train, X_test, Y_test)
    study.optimize(objective, n_trials=total_combinations, show_progress_

    elapsed_time = time.time() - start_time

    print(f"\n{'='*80}")
    print(f" GRID SEARCH - MEJORES HIPERPARÁMETROS:")
    print(f"{'='*80}")
    for param, value in study.best_params.items():
        print(f"   {param}: {value}")
    print(f"\n   Best Test Accuracy: {study.best_value:.2f}%")
    print(f"   Tiempo total: {elapsed_time:.2f} segundos")
    print(f"{'='*80}\n")

    return {
        'strategy': 'Grid Search',
        'best_params': study.best_params,
        'best_score': study.best_value,
        'study': study,
        'time': elapsed_time
    }
```

Ejecución Estrategias de Hiperparmetros para Función Keras

```
# ============================================================================
# EJECUTAR OPTIMIZACIÓN CON TODAS LAS ESTRATEGIAS
# ============================================================================


print("OPTIMIZACIÓN DE HIPERPARÁMETROS CON KERAS + OPTUNA")
print(f"Dataset: {best_method_name}")


# Diccionario para resultados
keras_optimization_results = {}

# 1. Random Search
print("Ejecutando Random Search...")
keras_optimization_results['random_search'] = random_search_optuna(
    X_train_best, Y_train_best, X_test_best, Y_test_best,
    n_trials=30
)

# 2. Bayesian Optimization
print("\nEjecutando Bayesian Optimization...")
keras_optimization_results['bayesian'] = bayesian_optimization_optuna(
    X_train_best, Y_train_best, X_test_best, Y_test_best,
    n_trials=30
)

# 3. Hyperband
print("\nEjecutando Hyperband...")
keras_optimization_results['hyperband'] = hyperband_optimization_optuna(
    X_train_best, Y_train_best, X_test_best, Y_test_best,
    n_trials=40
)

# 4. Grid Search (opcional, puede ser muy lento)
print("\nEjecutando Grid Search...")
keras_optimization_results['grid_search'] = grid_search_optuna(
    X_train_best, Y_train_best, X_test_best, Y_test_best
)

print("\n" + "="*80)
print(" TODAS LAS ESTRATEGIAS COMPLETADAS")
print("="*80 + "\n")
```

```
OPTIMIZACIÓN DE HIPERPARÁMETROS CON KERAS + OPTUNA
Dataset: Basic Normalization (/ 255)
Ejecutando Random Search...

================================================================================
 ESTRATEGIA 1: RANDOM SEARCH con Optuna
================================================================================

 Configuración:
   Trials: 30
   Sampler: Random

 Espacio de hiperparámetros:
   learning_rate: [1e-5, 0.5] (log)
   num_iterations: [50, 500] (step 50)
   activation_function: ['sigmoid', 'tanh', 'relu']
   lambda_reg: [1e-6, 0.1] (log)
   initialization: ['zeros', 'glorot_uniform', 'he_uniform']
   n_hidden_units: [0, 16, 32, 64, 128]

Best trial: 8. Best value: 100: 100%                    30/30 [16:25<00:00, 27.73s/it]


================================================================================
 RANDOM SEARCH - MEJORES HIPERPARÁMETROS:
================================================================================
   learning_rate: 0.2336799125606562
   num_iterations: 450
   activation_function: tanh
   lambda_reg: 8.56742466787506e-06
   initialization: zeros
   n_hidden_units: 0

   Best Test Accuracy: 100.00%
   Tiempo total: 985.00 segundos
================================================================================


Ejecutando Bayesian Optimization...

================================================================================
 ESTRATEGIA 2: BAYESIAN OPTIMIZATION (TPE)
================================================================================

 Configuración:
   Trials: 30
   Sampler: TPE (Tree-structured Parzen Estimator)
   Estrategia: Aprende de trials anteriores

Best trial: 3. Best value: 100: 100%                    30/30 [20:46<00:00, 45.35s/it]


================================================================================
 BAYESIAN OPTIMIZATION - MEJORES HIPERPARÁMETROS:
================================================================================
   learning_rate: 1.4507434860119771e-05
   num_iterations: 500
   activation_function: tanh
   lambda_reg: 0.00039841905944346893
   initialization: he_uniform
```