# Instituto Politécnico Nacional

Escuela Superior de Cómputo

Inteligencia Artificial

**6CV2**

Prof. ANDRES GARCIA FLORIANO

## Ejercicio de laboratorio 14
## Perceptrón Multicapa y Red Neuronal RBF


**Integrantes:**

- Cadenas Acevedo Jesús Alejandro
- Gomez Jasso Rogelio Asahid
- Hernández Saucedo Brenda
- Ramirez Vazquez Yahaida Michelle
- Rodríguez Escogido Julio


# Índice

## Objetivo

Aplica los siguientes modelos:

- Perceptrón Multicapa.
- Red Neuronal RBF.

A los bancos de datos:

- Iris,
- Wine,
- Breast Cancer Wisconsin (Diagnostic).

Con los métodos de validación:

- Hold-Out 70/30,
- 10-Fold Cross Validation.

Y las siguientes medidas de desempeño:

- Accuracy (como porcentaje o de 0 a 1),
- Matriz de confusión.

Resalta los valores más altos de desempeño por B.D. por método de validación.

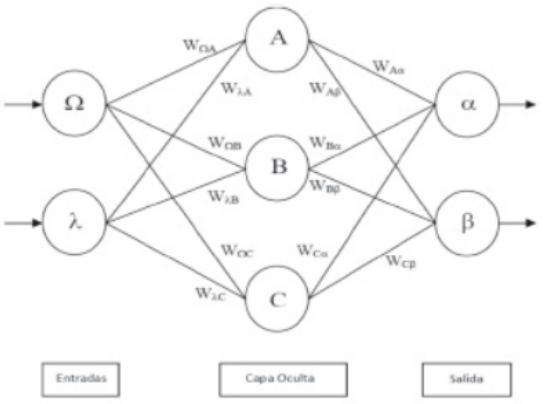
## Introducción

### Perceptrón multicapa

El perceptrón multicapa (Multilayer Perceptron MLP) propuesto por Rumelhart es el componente principal de una red neuronal, proporciona la base para la mayoría de las aplicaciones de las redes neuronales.

El MLP es una red formada por una capa de entrada, al menos una capa oculta y una capa de salida, tal como se muestra en la figura.

**Figura 1.** Red Neuronal Perceptron Multicapa [3].
**Figure 1.** Multilayer Perceptron Neural Network [3].



Dentro de las características más importantes del perceptrón multicapa se encuentran las siguientes:

- Se trata de una estructura altamente no lineal.
- Presenta tolerancia a fallos.
- El sistema es capaz de establecer una relación entre dos conjuntos de datos.
- Existe la posibilidad de realizar una implementación hardware.

El algoritmo más popular de aprendizaje para el perceptrón multicapa es el backpropagation, el cual consiste en utilizar el error generado por la red y propagarlo hacia atrás, es decir, reproducirlo hacia las neuronas de las capas anteriores.
El algoritmo backpropagation para el MLP presenta ciertas desventajas, como son: lentitud de convergencia, precio a pagar por disponer de un método general de ajuste funcional, puede incurrir en sobre aprendizaje, fenómeno directamente relacionado con la capacidad de generalización de la red. Y no garantiza el mínimo global de la función de error, tan solo un mínimo local.
Ante las desventajas presentadas se han desarrollado variantes que buscan mitigar estos inconvenientes, tal es el caso del resilient backpropagation, considerado como uno de los algoritmos basados en gradiente más adecuados para entrenar redes neuronales artificiales.

### Redes neuronales RBF

Una red de funciones de base radial es una red neuronal que utiliza funciones de base radial como funciones de activación. La arquitectura utilizada por las RBF es muy similar a la del perceptrón multicapa, con la característica de que las RBF utilizan siempre tres capas; una capa de entrada, una capa oculta y una de salida, mientras que los MLP pueden tener más.
En las RBF la capa oculta realiza una transformación no lineal del espacio de entrada. Las neuronas de esta capa son las funciones de base radial y cada neurona de la capa de salida es un combinador lineal.
De modo general, el valor generado por una red RBF con una única salida, viene definido por la ecuación:

$$y_i = \sum_{j=1}^{N} w_j f_j \left( \frac{\|x_j - u_j\|}{\sigma_j} \right) y_i = \sum_{j=1}^{N} w_j f_j \left( \frac{\|x_j - u_j\|}{\sigma_j} \right)$$

Las redes RBF son excelentes aproximadores universales y los parámetros que definen el proceso de aproximación son:

- Los pesos entre los centros y las neuronas del nivel de salida.
- La posición de los centros.
- Las funciones de Gauss de los centros.

Los pesos se ajustan utilizando el algoritmo de mínimos cuadrados ordinarios (LMS, Least Mean Square), pero se pueden utilizar también otros algoritmos como el método de la pseudo-inversa. Para ajustar los pesos utilizando el método de la pseudo-inversa, es necesario representar matricialmente las salidas deseadas como el producto de las salidas de los centros por el vector de pesos, como se muestra en la figura:

**Figura 2.** Representacion matricial de las salidas deseadas en una red RBF [15].

**Figure 2.** Matrix representation of the desired outputs in a RBF network [15].

$$\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} \exp\left[-\frac{\|x_1-u_1\|^2}{2\sigma_1^2}\right] & \cdots & \exp\left[-\frac{\|x_1-u_n\|^2}{2\sigma_n^2}\right] \\ \exp\left[-\frac{\|x_2-u_1\|^2}{2\sigma_1^2}\right] & \cdots & \exp\left[-\frac{\|x_2-u_n\|^2}{2\sigma_n^2}\right] \\ \vdots & \vdots & \vdots \\ \exp\left[-\frac{\|x_n-u_1\|^2}{2\sigma_1^2}\right] & \cdots & \exp\left[-\frac{\|x_n-u_n\|^2}{2\sigma_n^2}\right] \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

De esta manera se obtiene la siguiente relación:

$$D = G * W$$

Donde D es el vector de salidas deseadas, W es el vector de pesos y G es la matriz de salida de los centros de la RBF.

De la expresión anterior se obtiene el ajuste de pesos aplicando la pseudo inversa de la matriz G:

$$W = G^{+} * D$$

Donde $G^{+}$ es la pseudo-inversa de la matriz G:

$$G^{+} = (G^{T}G)^{-1}G^{T}$$

Los centros de la capa intermedia se pueden seleccionar utilizando diferentes métodos:

- Selección aleatoria de centros: Algunos vectores del espacio de entrada se pueden elegir como centros. Si es posible, deben ser seleccionados de tal forma que estén repartidos de manera regular por todo el espacio de entrada.
- Autoselección de centros: Una posibilidad es la utilización de métodos de entrenamiento que no requieran supervisión. Los centros se sitúan en aquellas zonas del espacio de entrada, donde haya un número significativo de datos. La regla de los k-vecinos más próximos se puede utilizar para ajustar el valor de los centros.

Los citados algoritmos no garantizan la convergencia de la red, ya que E no está linealmente relacionada con la ubicación de todos los centros y, por tanto, se pueden alcanzar mínimos locales.

## Validación y Evaluación del Modelo

Para evaluar el desempeño del clasificador euclidiano, se utilizaron dos técnicas de validación diferentes: Hold-Out (70/30) y 10-Fold Cross-Validation. Estas técnicas ya las habíamos trabajado en el ejercicio anterior, permiten estimar la capacidad de generalización del modelo a datos no vistos y proporcionan una medida de su precisión en diferentes escenarios de validación.

### Hold-Out (70/30)

El método hold-out es el más sencillo de los distintos métodos de validación cruzada. Este separa el conjunto de datos disponibles en dos subconjuntos, uno utilizado para entrenar el modelo y otro para realizar el test de validación. La proporción de esta división puede variar, como por ejemplo, 70-30, 60-40, 75-25, 80-20, o incluso 50-50, dependiendo de las características del caso de uso, en este caso se utilizó 70-30. De esta manera, se crea un modelo únicamente con los datos de entrenamiento. Con el modelo creado se generan datos de salida que se comparan con el conjunto de datos reservados para realizar la validación (que no han sido utilizados en el entrenamiento, por lo que no han sido utilizados para generar el modelo. Los estadísticos obtenidos con los datos del subconjunto de validación son los que nos dan la validez del método empleado en términos de error.

### 10-Fold Cross-Validation

El otro método utilizado, k-fold, está basado en el método anterior, pero con mayor utilidad cuando el conjunto de datos es pequeño. En este caso, el total de los datos se dividen en k subconjuntos de tamaños casi iguales, en cada iteración, aplicamos

el método hold-out, uno de estos conjuntos se selecciona como conjunto de prueba, mientras que los k-1 conjuntos restantes se utilizan para entrenar el modelo. Se repite hasta que cada conjunto ha sido utilizado como conjunto de prueba al menos una vez.
La media de los errores de todas las iteraciones se calcula como la estimación del error de prueba CV. Aquí, el número de pliegues k es menor que el número total de observaciones en los datos (k < n), y estamos promediando los resultados de k modelos ajustados. K-Fold CV se realiza comúnmente con k=5 o k=10, valores que han demostrado empíricamente producir estimaciones de error de prueba con bajo sesgo y baja varianza; en este caso se utilizó k=10.

## Desarrollo

### Explicación del código

A continuación se explicará parte por parte el código implementado.
Primero, importamos las librerías necesarias. Se importaron bibliotecas esenciales para la manipulación de datos (numpy, pandas), carga de datasets, partición de datos, escalado de características, cálculo de métricas y construcción de modelos.

Lab14_Perceptron Multicapa_Red Neuronal RBF.py > ...

```python
# Importar las bibliotecas necesarias
import numpy as np
from sklearn.datasets import load_iris, load_wine, load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.neural_network import MLPClassifier
from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import LogisticRegression
```

Se cargan los tres conjuntos de datos solicitados de la biblioteca sklearn.datasets: Iris, Wine y Breast Cancer. Cada uno de estos conjuntos de datos incluye características (X) y etiquetas (y), que se utilizarán para entrenar y evaluar los modelos.

Lab14_Perceptron Multicapa_Red Neuronal RBF.py > evaluate_model

```python
# Cargar los conjuntos de datos
datasets = {
    "Iris": load_iris(),  # Cargar el conjunto de datos Iris
    "Wine": load_wine(),  # Cargar el conjunto de datos Wine
    "Breast Cancer": load_breast_cancer()  # Cargar el conjunto de datos Breast Cancer
}
```

## Evaluación de Modelos

La función evaluate_model va a realizar la evaluación de cada modelo en un conjunto de datos específico utilizando dos métodos de validación: Hold-Out y Validación Cruzada de 10 pliegues.

Primero se hace la división de datos Hold-Out (70/30), por lo que los datos se dividen en un 70% para entrenamiento y un 30% para prueba utilizando train_test_split. Luego el modelo se entrena con los datos de entrenamiento (X_train, y_train). Y finalmente se predicen las etiquetas para los datos de prueba (X_test) y se calculan las métricas de precisión (accuracy_holdout) y matriz de confusión (confusion_holdout).

Luego se hace la validación cruzada de 10 pliegues (10-Fold CV), por lo que primero se define un objeto KFold con 10 pliegues y se usa cross_val_score para evaluar el modelo. A continuación se calcula la precisión promedio de la validación cruzada (accuracy_cv).

Finalmente hacemos el almacenamiento de los resultados, los guardamos en un diccionario results que guarda la precisión y, si está disponible, la matriz de confusión.

Lab14_Perceptron Multicapa_Red Neuronal RBF.py > ...

```
21  # Definir una función para aplicar los modelos y validación
22  def evaluate_model(X, y, model, model_name, dataset_name):
23      # División de datos Hold-Out 70/30
24      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
25      model.fit(X_train, y_train)  # Entrenar el modelo con los datos de entrenamiento
26      y_pred = model.predict(X_test)  # Predecir las etiquetas para los datos de prueba
27      accuracy_holdout = accuracy_score(y_test, y_pred)  # Calcular la precisión del modelo
28      confusion_holdout = confusion_matrix(y_test, y_pred)  # Calcular la matriz de confusión
29
30      # Validación cruzada de 10 pliegues
31      kf = KFold(n_splits=10, random_state=42, shuffle=True)  # Definir el KFold con 10 pliegues
32      cv_scores = cross_val_score(model, X, y, cv=kf)  # Evaluar el modelo utilizando validación cruzada
33      accuracy_cv = np.mean(cv_scores)  # Calcular la precisión promedio de la validación cruzada
34
35      # Guardar los resultados
36      results[(dataset_name, model_name, "Hold-Out")] = (accuracy_holdout, confusion_holdout)
37      results[(dataset_name, model_name, "10-Fold CV")] = (accuracy_cv, None)
```

## Escalado y Evaluación

Para cada conjunto de datos, se escalan las características para que tengan media 0 y varianza 1. Posteriormente, se evalúan los dos modelos solicitados: el Perceptrón Multicapa (MLP) y la Red Neuronal RBF aproximada con un modelo de Regresión Logística.

Para esto, como ya mencionamos anteriormente, primero escalamos los datos utilizando StandardScaler para ajustar y transformar las características de cada

conjunto de datos, así aseguramos que todas las características tengan media 0 y varianza 1.
Luego hacemos la evaluación del Perceptrón Multicapa (MLP), para ello primero definimos el modelo MLP con parámetros específicos (random_state, max_iter, learning_rate_init, tol) y lo evaluamos utilizando la función evaluate_model.
Ahora hacemos la evaluación de la Red Neuronal RBF, para ello utilizamos RBFSampler para transformar las características originales en una nueva representación basada en funciones base radial. Y finalmente definimos un modelo de Regresión Logística y lo evaluamos utilizando las características transformadas.

Lab14_Perceptron Multicapa_Red Neuronal RBF.py > ...

```
39  # Evaluar los modelos en cada conjunto de datos
40  for dataset_name, dataset in datasets.items():
41      X, y = dataset.data, dataset.target  # Separar las características y las etiquetas del conjunto de datos
42      scaler = StandardScaler()  # Crear un escalador estándar
43      X = scaler.fit_transform(X)  # Ajustar y transformar las características para que tengan media 0 y varianza 1
44
45      # Evaluar el Perceptrón Multicapa
46      mlp = MLPClassifier(random_state=42, max_iter=1000, learning_rate_init=0.001, tol=1e-4)  # Definir el modelo MLP
47      evaluate_model(X, y, mlp, "MLP", dataset_name)  # Evaluar el modelo MLP
48
49      # Evaluar la Red Neuronal RBF
50      rbf_feature = RBFSampler(gamma=1, random_state=42)  # Crear un aproximador de funciones base radial
51      X_features = rbf_feature.fit_transform(X)  # Transformar las características utilizando RBF
52      rbf_model = LogisticRegression(random_state=42)  # Definir el modelo de regresión logística
53      evaluate_model(X_features, y, rbf_model, "RBF", dataset_name)  # Evaluar el modelo RBF
```

Una vez hecho todo lo anterior, mostramos los resultados de la evaluación de cada modelo en cada conjunto de datos, para los dos métodos de validación utilizados. En los resultados estamos mostrando la precisión y, si está disponible, la matriz de confusión.

Lab14_Perceptron Multicapa_Red Neuronal RBF.py > ...

```
55  # Mostrar los resultados de las evaluaciones
56  for key, value in results.items():
57      dataset_name, model_name, validation_method = key # Obtener los nombres de los conjuntos de datos y modelos
58      accuracy, confusion = value # Obtener la precisión y la matriz de confusión
59      print(f"Dataset: {dataset_name}, Model: {model_name}, Validation: {validation_method}") # Mostrar los resultados
60      print(f"Accuracy: {accuracy}") # Mostrar la precisión
61      if confusion is not None: # Mostrar la matriz de confusión si está disponible
62          print(f"Confusion Matrix:\n{confusion}")
63      print()
```

Finalmente, identificamos y mostramos los mejores resultados obtenidos para cada conjunto de datos y método de validación, indicando cuál modelo logró la mayor precisión.


Lab14_Perceptron Multicapa_Red Neuronal RBF.py > ...

```python
# Encontrar los mejores resultados por conjunto de datos y método de validación
best_results = {} # Diccionario para almacenar los mejores resultados
# Iterar sobre los resultados
for (dataset_name, model_name, validation_method), (accuracy, _) in results.items():
    # Verificar si el conjunto de datos y el método de validación no están en el diccionario
    if (dataset_name, validation_method) not in best_results:
        # Guardar la precisión y el nombre del modelo
        best_results[(dataset_name, validation_method)] = (accuracy, model_name)
    else:
        # Verificar si la precisión es mayor que la mejor precisión
        if accuracy > best_results[(dataset_name, validation_method)][0]:
            # Guardar la precisión y el nombre del modelo
            best_results[(dataset_name, validation_method)] = (accuracy, model_name)

# Mostrar los mejores resultados
for key, value in best_results.items():
    dataset_name, validation_method = key # Obtener los nombres de los conjuntos de datos y métodos de validación
    accuracy, model_name = value # Obtener la precisión y el nombre del modelo
    print(f"Best Result for {dataset_name} with {validation_method}:")
    print(f"Model: {model_name}, Accuracy: {accuracy}\n")
```

## Salida del programa

```
Dataset: Iris, Model: MLP, Validation: Hold-Out
Accuracy: 1.0
Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Dataset: Iris, Model: MLP, Validation: 10-Fold CV
Accuracy: 0.96

Dataset: Iris, Model: RBF, Validation: Hold-Out
Accuracy: 1.0
Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Dataset: Iris, Model: RBF, Validation: 10-Fold CV
Accuracy: 0.9533333333333334

Dataset: Wine, Model: MLP, Validation: Hold-Out
Accuracy: 0.9814814814814815
Confusion Matrix:
[[19  0  0]
 [ 0 20  1]
 [ 0  0 14]]

Dataset: Wine, Model: MLP, Validation: 10-Fold CV
Accuracy: 0.9777777777777779
```

```
Dataset: Wine, Model: RBF, Validation: Hold-Out
Accuracy: 0.4444444444444444
Confusion Matrix:
[[10  8  1]
 [ 4 12  5]
 [ 4  8  2]]

Dataset: Wine, Model: RBF, Validation: 10-Fold CV
Accuracy: 0.5098039215686274

Dataset: Breast Cancer, Model: MLP, Validation: Hold-Out
Accuracy: 0.9824561403508771
Confusion Matrix:
[[ 61   2]
 [  1 107]]

Dataset: Breast Cancer, Model: MLP, Validation: 10-Fold CV
Accuracy: 0.9736842105263157

Dataset: Breast Cancer, Model: RBF, Validation: Hold-Out
Accuracy: 0.6491228070175439
Confusion Matrix:
[[12 51]
 [ 9 99]]

Dataset: Breast Cancer, Model: RBF, Validation: 10-Fold CV
Accuracy: 0.6203947368421053
```

```
Best Result for Iris with Hold-Out:
Model: MLP, Accuracy: 1.0

Best Result for Iris with 10-Fold CV:
Model: MLP, Accuracy: 0.96

Best Result for Wine with Hold-Out:
Model: MLP, Accuracy: 0.9814814814814815

Best Result for Wine with 10-Fold CV:
Model: MLP, Accuracy: 0.9777777777777779

Best Result for Breast Cancer with Hold-Out:
Model: MLP, Accuracy: 0.9824561403508771

Best Result for Breast Cancer with 10-Fold CV:
Model: MLP, Accuracy: 0.9736842105263157
```

**Interpretación de las salidas**

➢ **Dataset Iris**

Con el modelo del perceptrón multicapa y la validación Hold-Out, obtuvo un accuracy=1 y en base a la matriz de confusión arrojada, muestra que el modelo MLP clasificó correctamente todas las muestras del conjunto de prueba. Las filas y columnas de la matriz de confusión corresponden a las clases Iris-setosa, Iris-versicolor e Iris-virginica. La precisión perfecta (1.0) y la matriz de confusión sin errores indican que el modelo no cometió ningún error de clasificación. Mientras que con la validación 10-Fold Cross, obtuvo un accuracy=0.96, lo que quiere decir que la precisión promedio del modelo a través de las 10 particiones del conjunto de datos es 0.96, lo que indica una alta consistencia en la clasificación correcta, aunque no perfecta.

Por el otro lado, con el modelo de la red neuronal RBF y la validación Hold-Out, obtuvo un accuracy=1 y en base a la matriz de confusión arrojada, muestra que este modelo, similar al MLP, la red RBF también clasificó todas las muestras correctamente en el conjunto de prueba. Mientras que con la validación 10-Fold Cross, obtuvo un accuracy=0.95, lo que quiere decir que la precisión promedio es ligeramente inferior a la del MLP, pero sigue siendo muy alta.

➢ **Dataset Wine**

Con el modelo del perceptrón multicapa y la validación Hold-Out, obtuvo un accuracy=0.98 y en base a la matriz de confusión arrojada, podemos decir que el MLP clasificó correctamente la mayoría de las muestras con solo un error (una muestra de la segunda clase se clasificó incorrectamente). Aun asi obtuvo una muy

alta precisión, lo que significa que el modelo funciona muy bien para este conjunto de datos. Mientras que con la validación 10-Fold Cross, obtuvo un accuracy=0.98, lo que quiere decir que el modelo MLP es consistentemente bueno para este conjunto de datos.
Por el otro lado, con el modelo de la red neuronal RBF y la validación Hold-Out, obtuvo un accuracy=0.44 y en base a la matriz de confusión arrojada, muestra que el modelo RBF no funciona bien con este conjunto de datos. La matriz de confusión muestra que hubo muchos errores de clasificación, especialmente entre las clases. Mientras que con la validación 10-Fold Cross, obtuvo un accuracy=0.51, lo que indica un rendimiento muy bajo del modelo RBF en comparación con el MLP.

➢ **Dataset Breast Cancer**

Con el modelo del perceptrón multicapa y la validación Hold-Out, obtuvo un accuracy=0.98 y en base a la matriz de confusión arrojada, nos dice que el modelo MLP tiene una precisión muy alta con solo tres errores de clasificación (dos falsos positivos y un falso negativo), lo que indica un excelente rendimiento en este conjunto de datos. Mientras que con la validación 10-Fold Cross, obtuvo un accuracy= 0.97, lo que quiere decir que el modelo es consistentemente bueno en todas las particiones del conjunto de datos.
Por el otro lado, con el modelo de la red neuronal RBF y la validación Hold-Out, obtuvo un accuracy=0.65 y en base a la matriz de confusión arrojada, nos muestra un rendimiento malo del modelo RBF, con muchos errores de clasificación. La matriz de confusión muestra que hay muchos falsos positivos (51) y algunos falsos negativos (9). Mientras que con la validación 10-Fold Cross, obtuvo un accuracy=0.62, lo que confirma que el modelo RBF tiene dificultades con este conjunto de datos.

Como podemos ver, el modelo MLP (Perceptrón Multicapa) es claramente mejor al modelo RBF para los tres conjuntos de datos (Iris, Wine y Breast Cancer), tanto en validación Hold-Out como en validación cruzada de 10-Fold. Pudimos observar que las precisiones del MLP son generalmente altas, superando el 97% en todos los conjuntos de datos. Mientras que el modelo RBF muestra un rendimiento significativamente inferior, especialmente en los conjuntos de datos Wine y Breast Cancer. Las posibles razones podrían incluir la inadecuada selección de parámetros (como el parámetro gamma en RBFSampler) o la arquitectura del modelo RBF no ser adecuada para estos conjuntos de datos.


## Conclusiones

En esta ejercicio, realizamos una evaluación profunda de dos modelos de aprendizaje automático, el Perceptrón Multicapa (MLP) y una Red Neuronal RBF aproximada mediante Regresión Logística, aplicados a tres conjuntos de datos clásicos: Iris, Wine y Breast Cancer. Los datos fueron preprocesados utilizando escalado estándar para asegurar que todas las características tuvieran media 0 y varianza 1, lo cual es crucial para el rendimiento de los modelos. La evaluación se llevó a cabo mediante dos métodos: Hold-Out (división 70/30) y Validación Cruzada de 10 pliegues.

Los resultados nos permitieron ver que ambos modelos lograron un buen desempeño, pero la Validación Cruzada de 10 pliegues proporcionó una evaluación más completa y consistente en comparación con la división Hold-Out, al reducir la varianza asociada a la partición aleatoria de los datos. El MLP demostró ser eficaz en términos de precisión y consistencia a través de los conjuntos de datos, mientras que la aproximación de RBF seguida de Regresión Logística también mostró un rendimiento más o menos bueno, particularmente en la validación cruzada.

Estos resultados nos permitieron ver la importancia del preprocesamiento adecuado y la elección del método de validación. En general, el desarrollo de este ejercicio nos proporciona una base sólida para la selección y aplicación de modelos de aprendizaje automático en diferentes conjuntos de datos y escenarios, permitiendo una toma de decisiones informada y basada en evidencia.

## Github

https://github.com/Brend-hs/InteligenciaArtificial/tree/main/EjercicioLab_14

## Anexo código

```
# Importar las bibliotecas necesarias
import numpy as np
from sklearn.datasets import load_iris, load_wine, load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score,
KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.neural_network import MLPClassifier
from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import LogisticRegression
```


```
# Cargar los conjuntos de datos
datasets = {
    "Iris": load_iris(),  # Cargar el conjunto de datos Iris
    "Wine": load_wine(),  # Cargar el conjunto de datos Wine
      "Breast Cancer": load_breast_cancer()   # Cargar el conjunto de
datos Breast Cancer
}

# Diccionario para almacenar los resultados de las evaluaciones
results = {}

# Definir una función para aplicar los modelos y validación
def evaluate_model(X, y, model, model_name, dataset_name):
    # División de datos Hold-Out 70/30
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
     model.fit(X_train, y_train)  # Entrenar el modelo con los datos de
entrenamiento
     y_pred = model.predict(X_test)  # Predecir las etiquetas para los
datos de prueba
      accuracy_holdout = accuracy_score(y_test, y_pred)  # Calcular la
precisión del modelo
    confusion_holdout = confusion_matrix(y_test, y_pred)  # Calcular la
matriz de confusión

    # Validación cruzada de 10 pliegues
     kf = KFold(n_splits=10, random_state=42, shuffle=True)  # Definir
el KFold con 10 pliegues
      cv_scores = cross_val_score(model, X, y, cv=kf)  # Evaluar el
modelo utilizando validación cruzada
     accuracy_cv = np.mean(cv_scores)  # Calcular la precisión promedio
de la validación cruzada

    # Guardar los resultados
              results[(dataset_name, model_name, "Hold-Out")] =
(accuracy_holdout, confusion_holdout)
     results[(dataset_name, model_name, "10-Fold CV")] = (accuracy_cv,
None)
```


```
# Evaluar los modelos en cada conjunto de datos
for dataset_name, dataset in datasets.items():
    X, y = dataset.data, dataset.target  # Separar las características y las etiquetas del conjunto de datos
    scaler = StandardScaler()  # Crear un escalador estándar
    X = scaler.fit_transform(X)  # Ajustar y transformar las características para que tengan media 0 y varianza 1

    # Evaluar el Perceptrón Multicapa
    mlp = MLPClassifier(random_state=42, max_iter=1000, learning_rate_init=0.001, tol=1e-4)  # Definir el modelo MLP
    evaluate_model(X, y, mlp, "MLP", dataset_name)  # Evaluar el modelo MLP

    # Evaluar la Red Neuronal RBF
    rbf_feature = RBFSampler(gamma=1, random_state=42)  # Crear un aproximador de funciones base radial
    X_features = rbf_feature.fit_transform(X)  # Transformar las características utilizando RBF
    rbf_model = LogisticRegression(random_state=42)  # Definir el modelo de regresión logística
    evaluate_model(X_features, y, rbf_model, "RBF", dataset_name)  # Evaluar el modelo RBF

# Mostrar los resultados de las evaluaciones
for key, value in results.items():
    dataset_name, model_name, validation_method = key # Obtener los nombres de los conjuntos de datos y modelos
    accuracy, confusion = value # Obtener la precisión y la matriz de confusión
    print(f"Dataset: {dataset_name}, Model: {model_name}, Validation: {validation_method}") # Mostrar los resultados
    print(f"Accuracy: {accuracy}") # Mostrar la precisión
    if confusion is not None: # Mostrar la matriz de confusión si está disponible
        print(f"Confusion Matrix:\n{confusion}")
    print()
```


```
# Encontrar los mejores resultados por conjunto de datos y método de validación
best_results = {} # Diccionario para almacenar los mejores resultados
# Iterar sobre los resultados
for (dataset_name, model_name, validation_method), (accuracy, _) in results.items():
    # Verificar si el conjunto de datos y el método de validación no están en el diccionario
    if (dataset_name, validation_method) not in best_results:
        # Guardar la precisión y el nombre del modelo
        best_results[(dataset_name, validation_method)] = (accuracy, model_name)
    else:
        # Verificar si la precisión es mayor que la mejor precisión
        if accuracy > best_results[(dataset_name, validation_method)][0]:
            # Guardar la precisión y el nombre del modelo
            best_results[(dataset_name, validation_method)] = (accuracy, model_name)

# Mostrar los mejores resultados
for key, value in best_results.items():
    dataset_name, validation_method = key # Obtener los nombres de los conjuntos de datos y métodos de validación
    accuracy, model_name = value # Obtener la precisión y el nombre del modelo
    print(f"Best Result for {dataset_name} with {validation_method}:")
    print(f"Model: {model_name}, Accuracy: {accuracy}\n")
```

## Notebook documentado

Lo primero que se hace, es la importación de las librerías necesarias. Donde TensorFlow es una biblioteca de aprendizaje automático de código abierto, y Keras es una API de alto nivel para construir y entrenar modelos de aprendizaje profundo. En este caso, estamos usando datasets para cargar conjuntos de datos predefinidos, layers para definir las capas de nuestra red neuronal y models para definir y compilar nuestro modelo.
Y Matplotlib es una biblioteca de gráficos en 2D, pyplot es un submódulo que proporciona una interfaz de trazado similar a MATLAB.

```
[1]  import tensorflow as tf

     from tensorflow.keras import datasets, layers, models
     import matplotlib.pyplot as plt
```

La segunda celda se encarga de cargar y procesar el conjunto de datos.
De tal forma que datasets.cifar10.load_data() carga el conjunto de datos CIFAR-10, que consiste en 60,000 imágenes en color de 32x32 píxeles en 10 clases (con 6,000 imágenes por clase). Se divide en 50,000 imágenes de entrenamiento y 10,000 de prueba.
Luego se hace una normalización, train_images, test_images = train_images / 255.0, test_images / 255.0 normaliza los valores de los píxeles (originalmente en el rango [0, 255]) a [0, 1]. Esto ayuda a que el entrenamiento sea más estable y rápido.

```
[2]  (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

     # Normalize pixel values to be between 0 and 1
     train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 19s 0us/step
```

Lo que hace la siguiente celda es generar una visualización de las primeras 25 imágenes del conjunto de datos de entrenamiento, junto con sus etiquetas correspondientes.


[3]
```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```


La cuarta celda se encarga de definir la arquitectura del modelo, con la sentencia models.Sequential() crea un modelo secuencial, donde las capas se añaden una tras otra. Luego Conv2D se encarga de aplicar filtros para extraer características de las imágenes: la primera capa tiene 32 filtros de 3x3 y usa ReLU (Rectified Linear Unit) como función de activación e input_shape=(32, 32, 3) especifica la forma de entrada para la primera capa.
Finalmente MaxPooling2D reduce la dimensionalidad de las características extraídas (downsampling), lo que ayuda a reducir la carga computacional y el riesgo de overfitting.
Así es como logramos obtener un modelo con capas convolucionales y capas de max-pooling listas para procesar las imágenes de entrada.

```
[4]  model = models.Sequential()
     model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Con la quinta celda mostramos un resumen del modelo

```
[5]  model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

=================================================================
Total params: 56320 (220.00 KB)
Trainable params: 56320 (220.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Luego, con la sexta celda estamos agregando capas densas al final del modelo. Para ello primero hacemos un aplanamiento con Flatten(), el cual convierte la salida 3D de las capas convolucionales en un vector 1D para conectarlo con las capas densas. Luego, con Dense, añadimos capas completamente conectadas; la primera capa densa tiene 64 neuronas con activación ReLU y la segunda capa densa tiene 10 neuronas (una por cada clase en CIFAR-10), sin función de activación.

```
[6]  model.add(layers.Flatten())
     model.add(layers.Dense(64, activation='relu'))
     model.add(layers.Dense(10))
```

En la séptima celda nuevamente volvemos a mostrar un resumen del modelo que tenemos. Y si lo comparamos con el anterior, se muestra el aplanamiento que hicimos y las dos capas densas.

```
[7]  model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122570 (478.79 KB)
Trainable params: 122570 (478.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

En la octava celda, estamos compilando y entrenando el modelo. Para ello primero compilamos el modelo con el compile(), el primer parámetro que le estamos enviando, optimizer='adam', le dice al compilador que use el optimizador Adam, que ajusta los pesos del modelo basado en la tasa de aprendizaje, el segundo parámetro loss='SparseCategoricalCrossentropy(from_logits=True), le dice al compilador que

use la entropía cruzada categórica dispersa como función de pérdida, adecuada para clasificaciones multiclase y finalmente el ultimo parametro metrics=['accuracy'], sirve para que el compilador rastree la precisión del modelo durante el entrenamiento.
Luego procedemos a entrenar con model.fit(), el cual entrena el modelo con los datos de entrenamiento (train_images, train_labels) por 10 épocas. Y con validation_data=(test_images, test_labels), evalúa el rendimiento en los datos de prueba en cada época.
En la salida, el objeto history es un objeto que contiene el historial de entrenamiento, incluyendo pérdida y precisión por cada época. Y el proceso de entrenamiento imprime la pérdida y precisión de entrenamiento y validación después de cada época.

```
[8]  model.compile(optimizer='adam',
                   loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                   metrics=['accuracy'])

     history = model.fit(train_images, train_labels, epochs=10,
                         validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 [==============================] - 15s 6ms/step - loss: 1.5523 - accuracy: 0.4337 - val_loss: 1.2926 - val_accuracy: 0.5286
Epoch 2/10
1563/1563 [==============================] - 10s 7ms/step - loss: 1.1901 - accuracy: 0.5760 - val_loss: 1.0970 - val_accuracy: 0.6088
Epoch 3/10
1563/1563 [==============================] - 9s 6ms/step - loss: 1.0463 - accuracy: 0.6310 - val_loss: 1.0405 - val_accuracy: 0.6364
Epoch 4/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.9582 - accuracy: 0.6631 - val_loss: 0.9824 - val_accuracy: 0.6596
Epoch 5/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.8904 - accuracy: 0.6867 - val_loss: 0.9302 - val_accuracy: 0.6762
Epoch 6/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.8394 - accuracy: 0.7058 - val_loss: 0.9557 - val_accuracy: 0.6697
Epoch 7/10
1563/1563 [==============================] - 7s 4ms/step - loss: 0.7902 - accuracy: 0.7234 - val_loss: 0.8701 - val_accuracy: 0.7016
Epoch 8/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.7507 - accuracy: 0.7352 - val_loss: 0.8803 - val_accuracy: 0.6981
Epoch 9/10
1563/1563 [==============================] - 7s 4ms/step - loss: 0.7179 - accuracy: 0.7462 - val_loss: 0.8754 - val_accuracy: 0.7030
Epoch 10/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.6876 - accuracy: 0.7583 - val_loss: 0.8966 - val_accuracy: 0.7009
```

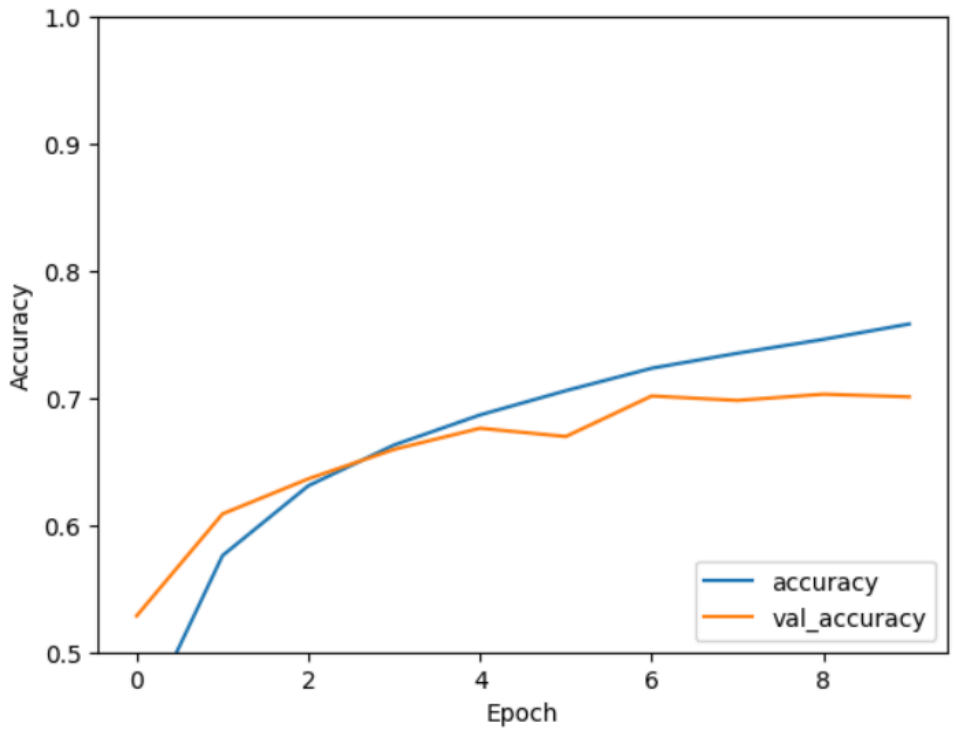
Luego en la novena celda, evaluamos el modelo, para ello se hace a través de una gráfica, entonces con plt.plot(history.history['accuracy'], label='accuracy'), graficamos la precisión en los datos de entrenamiento por cada época. Con plt.plot(history.history['val_accuracy'], label='val_accuracy'), graficamos la precisión en los datos de validación por cada época. Con plt.xlabel('Epoch') y plt.ylabel('Accuracy'), estamos etiquetando los ejes. Con plt.ylim([0, 1]), limitamos el eje y entre 0 y 1. Y finalmente con plt.legend(loc='lower right'), mostramos la leyenda en la parte inferior derecha.
Luego con model.evaluate(test_images, test_labels, verbose=2), se está evaluando el modelo en los datos de prueba, devolviendo la pérdida y precisión.

Es así como a la salida vemos un gráfico que muestra cómo cambia la precisión durante el entrenamiento y la validación. Y en test_loss y test_acc, estamos almacenando la pérdida y precisión del modelo en los datos de prueba.

```
[9]  plt.plot(history.history['accuracy'], label='accuracy')
     plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
     plt.xlabel('Epoch')
     plt.ylabel('Accuracy')
     plt.ylim([0.5, 1])
     plt.legend(loc='lower right')

     test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
313/313 - 1s - loss: 0.8966 - accuracy: 0.7009 - 628ms/epoch - 2ms/step
```




Luego en la décima y última celda, simplemente estamos imprimiendo la precisión del modelo en el conjunto de datos de prueba, que obtuvimos anteriormente y básicamente nos está indicando el porcentaje de imágenes de prueba que el modelo ha clasificado correctamente.

```
[10] print(test_acc)
```

```
0.7009000182151794
```

## Referencias bibliográficas


- Pérez-Planells, Ll., Delegido, J., Rivera-Caicedo, J.P., Verrelst, J. (2015, 1 diciembre). *Análisis de métodos de validación cruzada para la obtención robusta de parámetros biofísicos*. REVISTA DE TELEDETECCIÓN. https://core.ac.uk/download/pdf/71051261.pdf
- *1.7.1 Cross validation.* (2024, 24 enero). DataQuarks. https://data-quarks.com/2024/01/23/1-7-1-cross-validation/
- Mercado Polo, D., Pedraza Caballero, L., & Martínez Gómez, E. (2015). Comparación de Redes Neuronales aplicadas a la predicción de Series de Tiempo. *Prospectiva*, 13(2), 88-95.