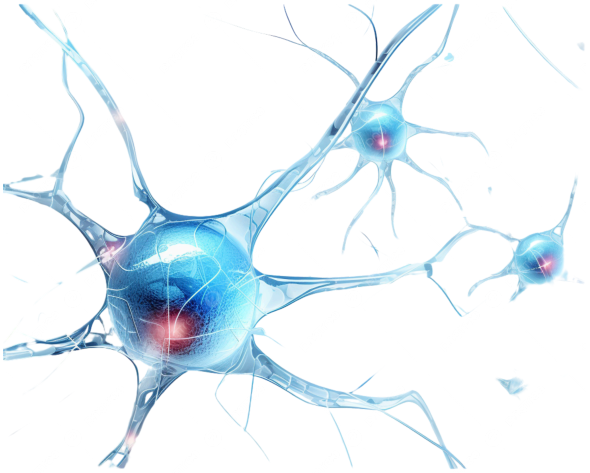
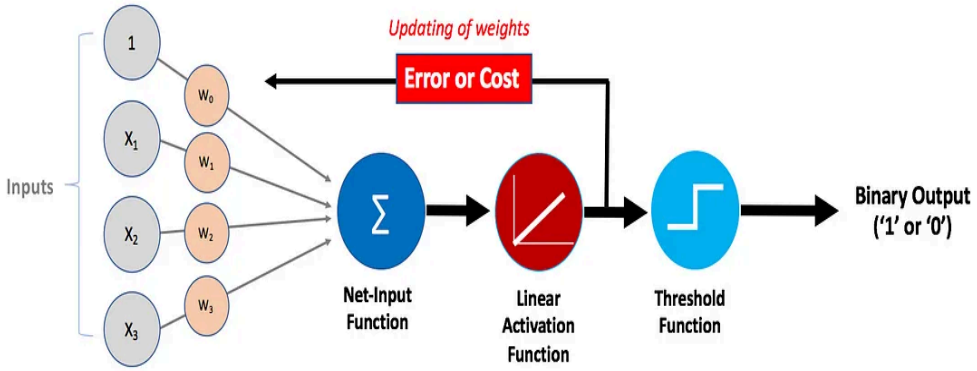
Aproximación de funciones mediante una Red Neuronal de Funciones de Base Radial





Javier Bolívar García-Izquierdo **Universidad de las Palmas de Gran Canaria**

Pablo Cabeza Lantigua **Inteligencia Artificial para Ciencia de Datos**

2ºGrado en Ciencia e Ingeniería de Datos

Año 2024-2025

Índice

Introducción

En este módulo práctico se nos pide desarrollar una Red Neuronal de Funciones de Base Radial (RBF) para aproximar la función:

$$f(x,\ y)\ =\ sen(\sqrt{x^2\ +\ y^2})$$ en el intervalo **-5 ≤ x ≤ 5, -5 ≤ y ≤ 5**.

El objetivo es construir y entrenar una red RBF capaz de aproximar esta función, utilizando un enfoque híbrido: la fase no supervisada para el cálculo de los centroides mediante el algoritmo k-means y la fase supervisada basada en el modelo ADALINE para el ajuste de los pesos de salida.

Se utilizarán funciones gaussianas como funciones de base radial. Además, se requiere analizar cómo afecta el número de unidades en la capa oculta al error cuadrático medio (MSE) de la red, en este documento se irá documentando el proceso y los resultados obtenidos.

Explicación teórica

El modelo ADALINE (ADAptive LInear NEuron), es una red neuronal de una sola capa. Su estructura es similar a la del Perceptrón, recibe un vector de entrada **X**, con pesos asociados **w**, y un sesgo **b**. El funcionamiento de ADALINE se basa en calcular una salida lineal:

$$y\ =\ w^T \cdot X\ +\ b$$

A diferencia del Perceptrón, que aplica una función de cuantización para producir una salida binaria durante el entrenamiento, ADALINE no discretiza la salida. Utiliza directamente el valor de salida (una predicción continua) para evaluar el error respecto al valor deseado y. Durante el entrenamiento, ADALINE emplea como criterio de aprendizaje el Error Cuadrático Medio (MSE):

$$E\ =\ \frac{1}{m}\sum_{i=1}^{n}(t_i\ -\ \phi(y))^2$$

Este error se minimiza mediante descenso del gradiente, actualizando los pesos de manera iterativa y no analítica, donde p es la tasa de aprendizaje:

$$w(k\ +\ 1)\ =\ w(k)\ -\ p(y\ -\ \widehat{y})x$$

$$b(k\ +\ 1)\ =\ b(k)\ -\ p(y\ -\ \widehat{y})$$

La principal diferencia entre ADALINE y el Perceptrón reside en cómo se actualizan sus pesos. El Perceptrón sólo actualiza cuando se produce un error de clasificación, y utiliza una función escalón que limita el aprendizaje a errores binarios. En cambio, ADALINE trabaja con errores continuos, lo que le permite ajustar los pesos incluso cuando la clasificación es correcta pero la salida no es óptima.

Esto hace que ADALINE tenga un comportamiento más estable durante el entrenamiento, siendo más robusto. El Perceptrón, en estos casos, puede estancarse al no encontrar la separación perfecta.

Explicación código modificado

A continuación vamos a explicar las modificaciones que hemos realizado al código original para cumplir la práctica asignada. Recomendamos ver los códigos directamente en el entorno de desarrollo.

- **Adaline.py**
  - **Código completo**

```python
import numpy as np

from AdalineSerializer import AdalineSerializer

class Adaline(object):

    def __init__(self, eta=0.01, pathModels='.//models'):
        self.W = list()
        self.errorForEachEpoch = list()
        self.eta = eta
        self.pathModels = pathModels

    def initRandomW(self, X):
        W_ = list()
        totalMean_ = np.mean(X)
        totalStd_ = np.std(X)
        means_ = np.mean(X, axis=0)
        stds_ = np.std(X, axis=0)
        ramdon_seed = np.random.RandomState()
        W_.append(ramdon_seed.normal(loc=totalMean_, scale=totalStd_))
        for mean, std in zip(means_, stds_):
            W_.append(ramdon_seed.normal(loc=mean, scale=std))
        return np.array(W_)

    def fit(self, trainingX, trainingY, epochs=50):
        nameModel_ = None
        DSerializer = AdalineSerializer(self.pathModels)
        if not self.W:
            self.W = self.initRandomW(trainingX)
        if self.errorForEachEpoch:
            bestError = min(self.errorForEachEpoch)
```

```
        else:
            bestError = np.inf

        for _ in range(epochs):
            randomIndices_ = np.array(range(trainingX.shape[0]))
            np.random.shuffle(randomIndices_)
                    for xi, target in zip(trainingX[randomIndices_, :],
trainingY[randomIndices_]):
                net_input = self._net_input(xi)
                activation_function = self._activation(net_input)
                errors = target - activation_function
                self.W[1:] = self.W[1:] + self.eta * errors * xi
                self.W[0] = self.W[0] + self.eta * errors
            error_ = self.error(trainingX, trainingY)
            self.errorForEachEpoch.append(error_)
            if error_ < bestError:
                bestError = error_
                                nameModel_ = DSerializer.writeModel(self.W,
self.errorForEachEpoch)
                                  print("Adaline::[{}] a better model is
written".format(nameModel_))
        return nameModel_

    def _net_input(self, X):
        return np.dot(X, self.W[1:]) + self.W[0]

    def _activation(self, p_net_input):
        return p_net_input

    def _quantization(self, p_activation):
        return np.where(p_activation >= 0.0, 1, -1)

    def predict(self, X):
        net_in = self._net_input(X)
        activation = self._activation(net_in)
        output = self._quantization(activation)
        return output

    def error(self, X, Y):
        error_ = 0
        for xi, target in zip(X, Y):
            net_input  = self._net_input(xi)
            activation = self._activation(net_input)
            error_ += ((target - activation) ** 2)
        return error_ / X.shape[0]

    def loadModel(self, pathModels, nameModel):
        PSerializer = AdalineSerializer(pathModels)
        self.W, self.errorForEachEpoch = PSerializer.readModel(nameModel)
```

- **Explicación**
  - **Clase con constructor**

La clase Adaline implementa el modelo ADALINE, explicado anteriormente, para entrenamiento supervisado. En el constructor se inicializan los pesos W como lista vacía, una lista para almacenar el error por época, la tasa de aprendizaje y la ruta para guardar los modelos.

```
class Adaline(object):

    def __init__(self, eta=0.01, pathModels='.//models'):
        self.W = list()
        self.errorForEachEpoch = list()
        self.eta = eta
        self.pathModels = pathModels
```

  - **Método initRandomW(self, X)**

Este método inicializa los pesos del modelo con valores aleatorios usando una distribución normal. Para ello, calcula la media y desviación estándar global de todos los datos, así como por cada característica individual.

Luego genera un peso inicial para el sesgo (W[0]) y pesos para cada entrada (W[1:]), distribuidos alrededor de estas medias con las desviaciones estándar calculadas.

```
def initRandomW(self, X):
    W_ = list()
    totalMean_ = np.mean(X)
    totalStd_ = np.std(X)
    means_ = np.mean(X, axis=0)
    stds_ = np.std(X, axis=0)
    ramdon_seed = np.random.RandomState()
    W_.append(ramdon_seed.normal(loc=totalMean_, scale=totalStd_))
    for mean, std in zip(means_, stds_):
        W_.append(ramdon_seed.normal(loc=mean, scale=std))
    return np.array(W_)
```

- **Método fit(self, trainingX, trainingY, epochs=50)**

Este método entrena el modelo usando la regla de actualización basada en descenso del gradiente, como se explicó en la sección teórica previa.

Recordemos que el ADALINE calcula una salida lineal a partir del vector de entrada y sus pesos, y actualiza estos pesos para minimizar el error cuadrático medio (MSE) entre la salida predicha y el valor esperado.

En el código, para cada muestra, se calcula el error continuo y se ajustan los pesos y el sesgo proporcionalmente a ese error y a la tasa de aprendizaje. Esta actualización de los pesos es la implementación práctica del descenso del gradiente, que permite que la red aprenda ajustando sus parámetros para reducir el error.

Al final de cada época, se calcula el MSE sobre todo el conjunto de entrenamiento y se guarda el modelo si mejora el error, asegurando que podamos conservar la mejor versión del modelo durante el proceso de aprendizaje.

```
def fit(self, trainingX, trainingY, epochs=50):
    nameModel_ = None
    DSerializer = AdalineSerializer(self.pathModels)
    if not self.W:
        self.W = self.initRandomW(trainingX)
    if self.errorForEachEpoch:
        bestError = min(self.errorForEachEpoch)
    else:
        bestError = np.inf

    for _ in range(epochs):
        randomIndices_ = np.array(range(trainingX.shape[0]))
        np.random.shuffle(randomIndices_)
                for xi, target in zip(trainingX[randomIndices_, :],
trainingY[randomIndices_]):
            net_input = self._net_input(xi)
            activation_function = self._activation(net_input)
            errors = target - activation_function
            self.W[1:] = self.W[1:] + self.eta * errors * xi
            self.W[0] = self.W[0] + self.eta * errors
        error_ = self.error(trainingX, trainingY)
        self.errorForEachEpoch.append(error_)
        if error_ < bestError:
            bestError = error_
                            nameModel_ = DSerializer.writeModel(self.W,
self.errorForEachEpoch)
                             print("Adaline::[{}] a better model is
written".format(nameModel_))
    return nameModel_
```

- **Método _net_input(self, X)**

Calcula la suma ponderada de las entradas más el sesgo. Recibe un vector X y calcula el producto con los pesos de entrada, sumándole el peso del sesgo.

```
def _net_input(self, X):
    return np.dot(X, self.W[1:]) + self.W[0]
```

- **Método _activation(self, p_net_input)**

Función de activación de la neurona ADALINE, que en este caso es la función identidad (salida lineal). Recibe la entrada neta y la devuelve sin cambios.

```
def _activation(self, p_net_input):
    return p_net_input
```

- **Método _quantization(self, p_activation)**

Función de cuantización para transformar la salida continua en una salida binaria. Devuelve 1 si la activación es mayor o igual a 0, y -1 en caso contrario.

```
def _quantization(self, p_activation):
    return np.where(p_activation >= 0.0, 1, -1)
```

- **Método predict(self, X)**

Recibe un conjunto de entradas X, calcula la entrada, la activa y luego la cuantiza para dar la predicción final.

```
def predict(self, X):
    net_in = self._net_input(X)
    activation = self._activation(net_in)
    output = self._quantization(activation)
    return output
```

- **Método error(self, X, Y)**

Calcula el error medio cuadrático (MSE) entre las salidas predichas y las salidas reales. Para cada muestra, calcula la diferencia cuadrática entre el valor objetivo y la activación.

```
def error(self, X, Y):
    error_ = 0
    for xi, target in zip(X, Y):
        net_input  = self._net_input(xi)
        activation = self._activation(net_input)
        error_ += ((target - activation) ** 2)
    return error_ / X.shape[0]
```

- **Método loadModel(self, pathModels, nameModel)**

Carga un modelo previamente guardado desde disco usando la clase AdalineSerializer. Restaura los pesos W y el historial de error por época para poder continuar o evaluar.

```
def loadModel(self, pathModels, nameModel):
    PSerializer = AdalineSerializer(pathModels)
    self.W, self.errorForEachEpoch = PSerializer.readModel(nameModel)
```

- **RBFVisualizer.py**
  - **Código completo**

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

class RBFVisualizer:
   def __init__(self, input_dimension, output_dimension,
learning_rate=0.01, save_path='.//models'):
       self.input_dimension = input_dimension
       self.output_dimension = output_dimension
       self.learning_rate = learning_rate
       self.save_path = save_path

   def evaluate_hidden_layers(self, hidden_layers, X, Y, epochs):
       mse_scores = []

       for units in hidden_layers:
           print(f"\n-- Running model with {units} hidden layers")
           model = RBFNN(numberInputs=self.input_dimension,
numberHideUnits=units, numberOutputs=self.output_dimension,
                         eta=self.learning_rate, pathModels=self.save_path)

           model.fit(X, Y, numberEpochs=epochs)
           predictions = model.predict(X)
           mse = np.mean((Y - predictions) ** 2)
           print(f"-- MSE at {units} layers: {mse:.4f}")
           mse_scores.append(mse)

       self.plot_results(hidden_layers, mse_scores, epochs)

   def plot_results(self, hidden_layers, mse_values, epochs):
       plt.figure(figsize=(8, 5))
       plt.plot(hidden_layers, mse_values, marker='s', linestyle='--',
color='green')
       plt.xlabel("Hidden Nodes")
       plt.ylabel("Mean Squared Error")
       plt.title(f"Performance of RBFNN, {epochs} Epochs,
{self.learning_rate} Eta")
       plt.xticks(hidden_layers)
       plt.grid(True)
       plt.tight_layout()
       plt.show()
```

- **Explicación**
  - **Clase con constructor**

La clase RBFVisualizer se crea para evaluar y visualizar cómo varía el rendimiento de una red neuronal RBF al modificar el número de neuronas en la capa oculta. En el constructor inicializamos las dimensiones de entrada y salida, la tasa de aprendizaje, y una ruta para guardar los modelos entrenados. Estos parámetros quedan guardados como atributos del objeto para usarlos en los métodos posteriores.

```
class RBFVisualizer:
            def     __init__(self,     input_dimension,     output_dimension,
learning_rate=0.01, save_path='.//models'):
       self.input_dimension = input_dimension
       self.output_dimension = output_dimension
       self.learning_rate = learning_rate
       self.save_path = save_path
```

  - **Método evaluate_hidden_layers(self, hidden_layers, X, Y, epochs)**

Este método permite probar la red neuronal con diferentes cantidades de neuronas en la capa oculta. Recibe una lista con los números de neuronas a evaluar (hidden_layers), los datos de entrada X y salida Y, y la cantidad de épocas para entrenar.

Por cada valor de neuronas, crea un modelo RBF, lo entrena, calcula las predicciones y el error cuadrático medio (MSE). Luego guarda el MSE para comparar el rendimiento entre configuraciones. Finalmente, llama a un método para graficar los resultados.

Visualizar el MSE en función de las neuronas ocultas es útil para detectar si el modelo tiene capacidad insuficiente (alto error con pocas neuronas) o sobreajuste (error alto con muchas neuronas), ayudando así a encontrar una arquitectura equilibrada.

```
def evaluate_hidden_layers(self, hidden_layers, X, Y, epochs):
   mse_scores = []

   for units in hidden_layers:
       print(f"\n-- Running model with {units} hidden layers")
                       model    =    RBFNN(numberInputs=self.input_dimension,
numberHideUnits=units, numberOutputs=self.output_dimension,
                     eta=self.learning_rate, pathModels=self.save_path)

       model.fit(X, Y, numberEpochs=epochs)
       predictions = model.predict(X)
       mse = np.mean((Y - predictions) ** 2)
       print(f"-- MSE at {units} layers: {mse:.4f}")
       mse_scores.append(mse)

   self.plot_results(hidden_layers, mse_scores, epochs)
```

- **Método plot_results(self, hidden_layers, mse_values, epochs)**

Este método se encarga de graficar la relación entre el número de neuronas en la capa oculta y el error cuadrático medio obtenido. Configura el gráfico con etiquetas, título, estilo de línea y cuadrícula para facilitar la interpretación visual de los resultados.

```
def plot_results(self, hidden_layers, mse_values, epochs):
   plt.figure(figsize=(8, 5))
   plt.plot(hidden_layers, mse_values, marker='s', linestyle='--',
color='green')
   plt.xlabel("Hidden Nodes")
   plt.ylabel("Mean Squared Error")
   plt.title(f"Performance of RBFNN, {epochs} Epochs, {self.learning_rate}
Eta")
   plt.xticks(hidden_layers)
   plt.grid(True)
   plt.tight_layout()
   plt.show()
```

- **DataGenerator.py**
  - **Código completo**

```
import numpy as np

class DataGenerator:
   def __init__(self, num_samples=625, range_min=-5, range_max=5):
       self.num_samples = num_samples
       self.range_min = range_min
       self.range_max = range_max

   def generate(self):
                   x   =   np.random.uniform(self.range_min,   self.range_max,
self.num_samples)
                   y   =   np.random.uniform(self.range_min,   self.range_max,
self.num_samples)
       features = np.column_stack((x, y))
       output = np.sin(np.sqrt(x**2 + y**2)).reshape(-1, 1)
       return features, output
```

  - **Explicación**
    - **Clase con constructor**

Definimos la clase DataGenerator que servirá para generar el conjunto de datos. En el constructor se inicializan tres parámetros que controlan la cantidad de datos y el rango de valores para x e y. Estos parámetros se guardan para usarlos luego en la generación de datos.

```
import numpy as np

class DataGenerator:
   def __init__(self, num_samples=625, range_min=-5, range_max=5):
       self.num_samples = num_samples
       self.range_min = range_min
       self.range_max = range_max
```

- **Método generate(self)**

El método genera muestras aleatorias uniformes para x e y dentro del rango definido. Luego combina estos vectores en una matriz de características donde cada fila representa un punto (x,y). Calcula el valor de la función objetivo $f(x,y) = sen(\sqrt{x^2 + y^2})$ para cada punto y lo devuelve junto con las características, listo para usarse en entrenamiento.

```
def generate(self):
    x = np.random.uniform(self.range_min, self.range_max, self.num_samples)
    y = np.random.uniform(self.range_min, self.range_max, self.num_samples)
    features = np.column_stack((x, y))
    output = np.sin(np.sqrt(x**2 + y**2)).reshape(-1, 1)
    return features, output
```

- **Main.py**
  - **Código completo y explicación**

```
from DataGenerator import DataGenerator
from RBFVisualizer import RBFVisualizer

if __name__ == '__main__':
    num_samples = 625
    epochs = 150
    learning_rate = 0.03
    hidden_layers = [3,6,9,12,15,18,21,24,27,30]

    data_gen = DataGenerator(num_samples=num_samples)
    X, y = data_gen.generate()

        visualizer = RBFVisualizer(input_dimension=2, output_dimension=1,
learning_rate=learning_rate,
                                   save_path='.//models')
    visualizer.evaluate_hidden_layers(hidden_layers, X, y, epochs)
```

  - **Explicación**

Este archivo es el punto de entrada que coordina todo el proceso del proyecto. Primero, se importan las clases necesarias, DataGenerator para crear los datos y RBFVisualizer para entrenar y evaluar la red.

- **Bloque principal**

En el bloque principal se definen los parámetro **num_samples**, que establece cuántos puntos de datos se generan. **epochs** determina cuántas veces se entrenará la red. **learning_rate** fija la velocidad de aprendizaje durante el ajuste. Por último, **hidden_layers** es una lista con distintos tamaños de la capa oculta que queremos probar.

```
if __name__ == '__main__':
    num_samples = 625
    epochs = 150
    learning_rate = 0.03
    hidden_layers = [3,6,9,12,15,18,21,24,27,30]
```

- **data_gen:**

Luego, se instancia el generador de datos y se generan los conjuntos de entradas (X) y salidas (y) aplicando la función.

```
data_gen = DataGenerator(num_samples=num_samples)
   X, y = data_gen.generate()
```

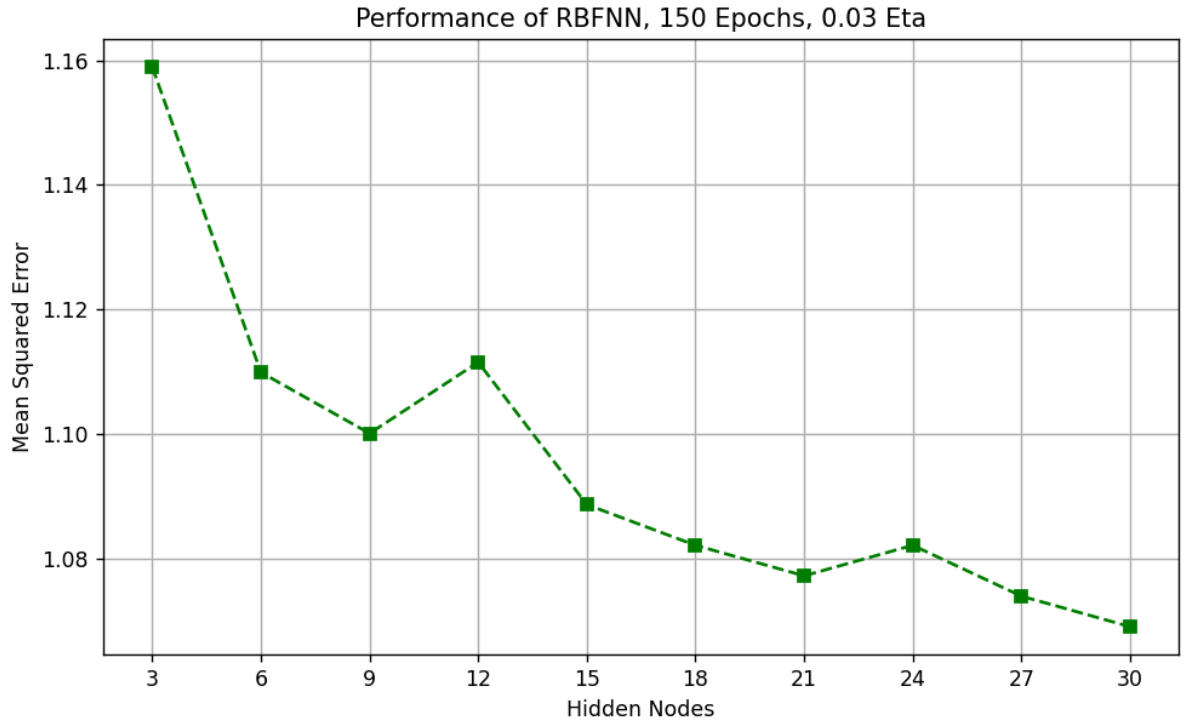
- **visualizador Red RBF**

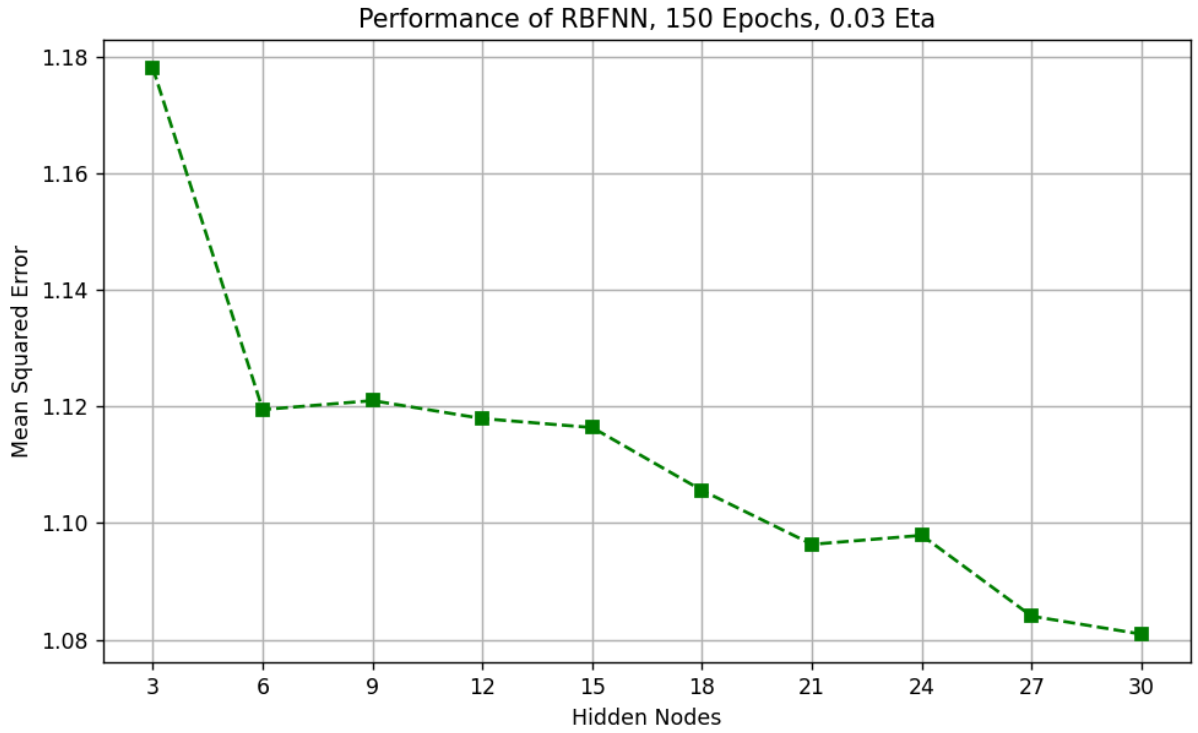
Después, se crea el visualizador de la red RBF con las dimensiones de entrada y salida, la tasa de aprendizaje y la ruta para guardar los modelos. Finalmente, se llama al método **evaluate_hidden_layers** para entrenar la red con cada tamaño de capa oculta definido, evaluar su desempeño y mostrar los resultados.

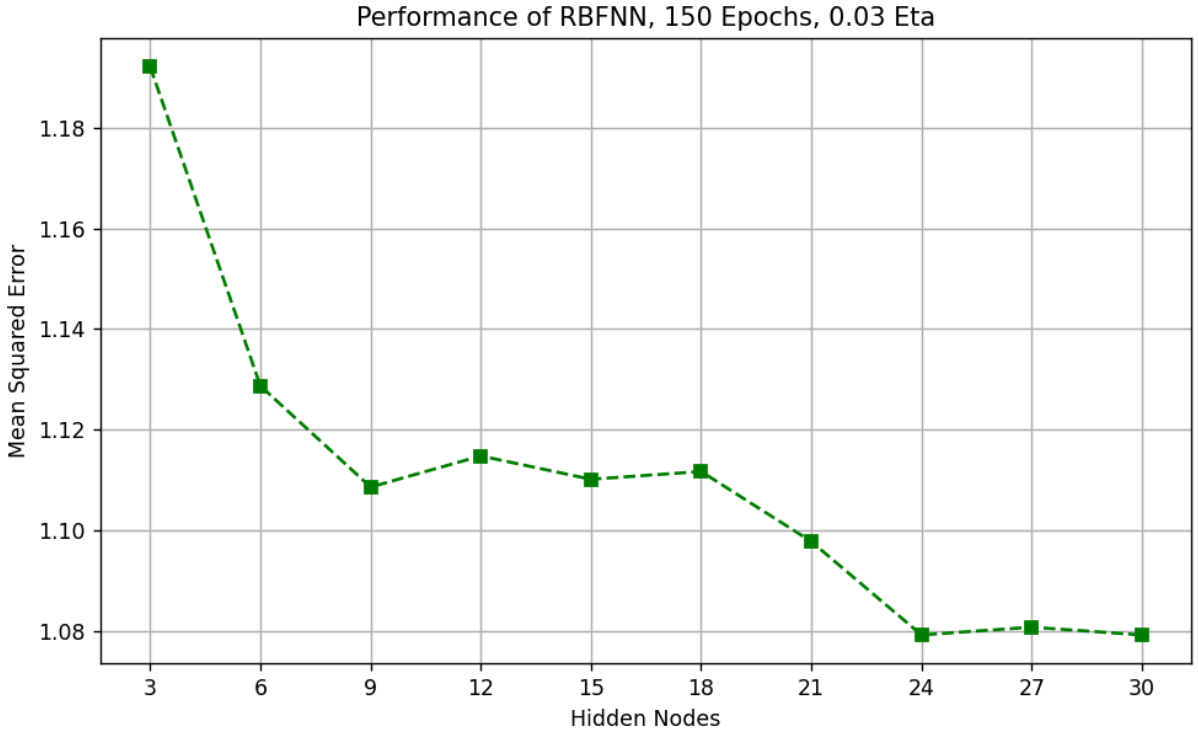
```
   visualizer    =    RBFVisualizer(input_dimension=2,    output_dimension=1,
learning_rate=learning_rate,
                              save_path='.//models')
   visualizer.evaluate_hidden_layers(hidden_layers, X, y, epoc
```

Análisis de resultados

Como se puede ver en las siguientes tres gráficas, hay una tendencia clara, el error cuadrático medio (MSE) va bajando a medida que se incrementa el número de nodos ocultos. Esta tendencia se repite en las tres ejecuciones, todas realizadas con 150 épocas y una tasa de aprendizaje de 0.03







Este descenso del MSE se debe a que, al añadir más nodos ocultos, la red tiene más capacidad para ajustarse a los datos. Cada nodo en la capa oculta funciona como un "centro" que responde a una zona específica del espacio de entrada. Por tanto, cuantos más nodos hay, mejor se cubre ese espacio y más fácil le resulta a la red captar los patrones y detalles de la función que está tratando de aproximar.

Aunque en algunos puntos se ven pequeñas subidas o bajadas en el MSE, esto es normal y puede deberse a factores aleatorios del entrenamiento, como cómo se inicializan los centros con k-means o cómo se ajustan los pesos con ADALINE. Lo importante es que, en general, el error sigue bajando con más nodos.

Al final, se ve claramente que las mejores configuraciones están entre 24 y 30 nodos, donde se alcanza el MSE más bajo de forma estable. Eso indica que ese rango es una buena opción para este problema, ya que ofrece una buena precisión sin que el modelo se complique demasiado. Eso sí, en otros casos, usar demasiados nodos podría llevar a sobreajuste.

En resumen, añadir más nodos ocultos mejora el rendimiento de la RBFNN porque le da más herramientas para representar bien la función que queremos aprender.

Conclusión

En este trabajo se ha demostrado que aumentar el número de nodos ocultos en una RBFNN mejora su capacidad para aproximar funciones, reduciendo el error cuadrático medio (MSE). Aunque existen pequeñas variaciones debido al entrenamiento, se ha observado que configuraciones con entre 27 y 30 nodos ofrecen el mejor rendimiento. Por tanto, una red más compleja puede ser beneficiosa siempre que se controle el riesgo de sobreajuste.