

Atividade Computacional Redes Neurais 2025.1

Discente: João Victor Moreira Cardoso

Problema

Seja a função

$$f(\mathbf{x}) = \sum_{n=1}^D (x_n)^{2n} - \exp\left\{-\sum_{n=1}^D \epsilon_n (x_n)^{2n}\right\}$$

onde:

- $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_n \ \dots \ x_D]^T$, $x \in [-30, 30]$;
 - Cada $0 \leq \epsilon_n \leq 1$ é sorteado de uma distribuição uniforme \implies podemos criar um vetor $\epsilon = [\epsilon_1 \ \epsilon_2 \ \dots \ \epsilon_D]$ de mesma dimensão D ;
-

Para $D = 3$:

$$f(\mathbf{x}) = (x_1)^2 + (x_2)^4 + (x_3)^6 - \exp\{-[\epsilon_1(x_1)^2 + \epsilon_2(x_2)^2 + \epsilon_3(x_3)^6]\}$$

Imports

```
In [1]: import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import Normalize
from mpl_toolkits.mplot3d import Axes3D

from math import sqrt
from tqdm import tqdm

import pandas as pd
pd.options.display.max_colwidth = 1000
pd.options.display.max_columns = 1000
pd.options.display.max_rows = 200

import sklearn
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_validate, train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder, MinMaxScaler, StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import StackingClassifier, RandomForestClassifier
from sklearn.svm import SVC, LinearSVC, SVR
```

```

from sklearn.kernel_ridge import KernelRidge

from sklearn.metrics import mean_squared_error, r2_score
import random
from random import seed, randrange
import requests
import io

import matplotlib.patches as mpatches
import seaborn as sns
import pdflatex

from sklearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
from scipy.linalg import norm

import warnings
warnings.filterwarnings("ignore")

```

Dataset

(a) Sorteio dos valores de ϵ_n

```
In [2]: np.random.seed(42) # Para reproduzibilidade
D = 3 # Dimensão do vetor epsilon
n_points = 2000
epsilon = np.array(np.random.uniform(0,1,D))
print(f"Valores de epsilon_n sorteados: {epsilon}")
```

Valores de epsilon_n sorteados: [0.37454012 0.95071431 0.73199394]

(b)(i) Geração dos pontos da função $(\vec{x}_n, f(\vec{x}_n))$

```
In [3]: def f(x, epsilon):
    """Calcula a função f(x) conforme definida"""
    sum_x = np.sum([x_n**(2*(n+1)) for n, x_n in enumerate(x)]) # Termo da Soma
    exp_term = np.exp(-np.sum([epsilon[n] * (x_n**(2*(n+1))) for n, x_n in enumerate(x)]))
    return sum_x - exp_term

# Gerar 2000 pontos aleatórios no intervalo [-30, 30]
X = np.random.uniform(-30, 30, (n_points, D))
y = np.array([f(x, epsilon) for x in X])
```

(b)(ii) Informar os pontos da função $(\vec{x}_n, f(\vec{x}_n))$

```
In [4]: print(f'Vetor X:\n', X, '\n')
```

```
Vetor X:
[[ 5.91950905 -20.63888157 -20.64032878]
[-26.51498327  21.97056875   6.0669007 ]
[ 12.48435467 -28.76493034  28.19459113]
...
[-25.66720888  11.06612166   0.19322489]
[ 15.90893099 -0.88256196 -21.03710183]
[  8.89541724 -19.65681827  22.3436738 ]]
```

```
In [5]: print(f'Vetor y:\n', y, '\n')
```

```
Vetor y:
[7.75028721e+07 2.83573653e+05 5.03021363e+08 ... 1.56550243e+04
8.66795655e+07 1.24580062e+08]
```

(b)(iii) Visualização dos pontos da função ($\vec{x}_n, f(\vec{x}_n)$)

Bidimensional

```
# Dividir em treino (1500), validação (250) e teste (250)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=500, random_
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, r

print("\n(b) Distribuição dos pontos:")
print(f"Treinamento: {X_train.shape[0]} pontos")
print(f"Validação: {X_val.shape[0]} pontos")
print(f"Teste: {X_test.shape[0]} pontos")

# Normalizar os valores de y para o intervalo [0,1]
norm = Normalize(vmin=min(y_train.min(), y_val.min(), y_test.min()),
                  vmax=max(y_train.max(), y_val.max(), y_test.max()))

plt.figure(figsize=(12, 6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=norm(y_train), cmap='viridis', alpha
plt.scatter(X_val[:, 0], X_val[:, 1], c=norm(y_val), cmap='viridis', marker='s',
plt.scatter(X_test[:, 0], X_test[:, 1], c=norm(y_test), cmap='viridis', marker=''

# Adicionar barra de cores
cbar = plt.colorbar()
cbar.set_label('Valor de f(x) (normalizado)')

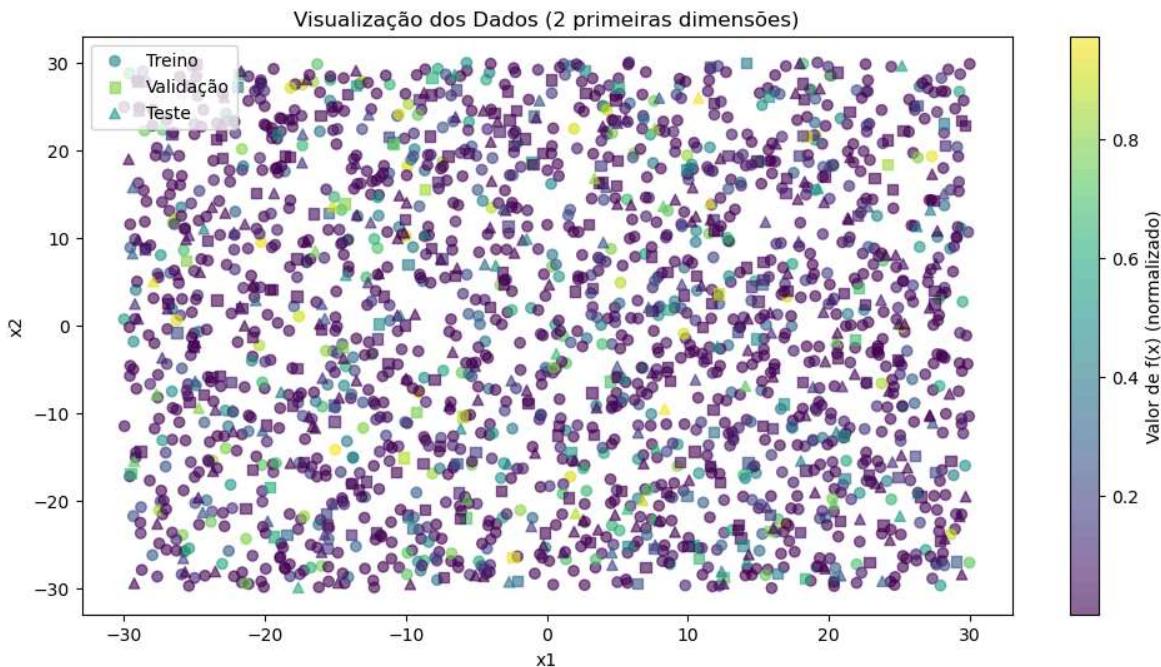
plt.title("Visualização dos Dados (2 primeiras dimensões)")
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.show()
```

(b) Distribuição dos pontos:

Treinamento: 1500 pontos

Validação: 250 pontos

Teste: 250 pontos



Tridimensional

```
In [7]: # Configuração do gráfico 3D
fig = plt.figure(figsize=(14, 10))
ax = fig.add_subplot(111, projection='3d')

# Plot dos pontos de treinamento
sc_train = ax.scatter(X_train[:, 0], X_train[:, 1], X_train[:, 2],
                      c=y_train, cmap='viridis', alpha=0.6,
                      label='Treino (1500 pts)', marker='o')

# Plot dos pontos de validação
sc_val = ax.scatter(X_val[:, 0], X_val[:, 1], X_val[:, 2],
                     c=y_val, cmap='viridis', alpha=0.6,
                     label='Validação (250 pts)', marker='s')

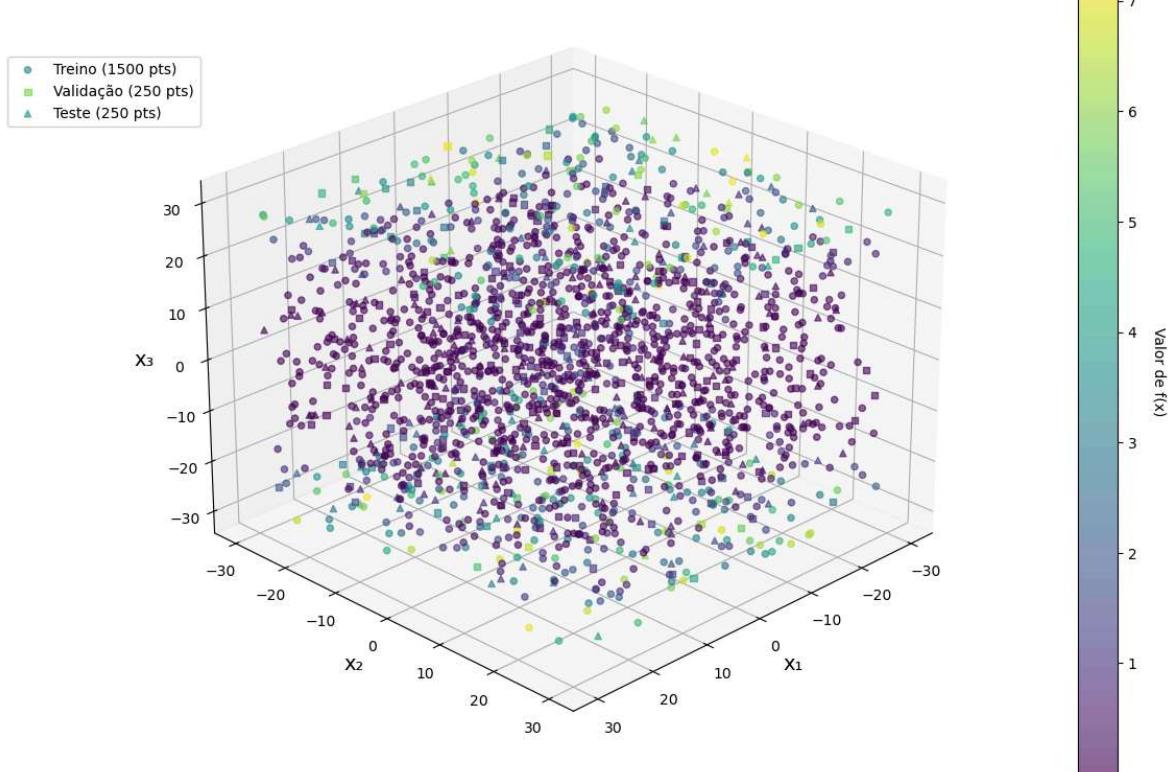
# Plot dos pontos de teste
sc_test = ax.scatter(X_test[:, 0], X_test[:, 1], X_test[:, 2],
                      c=y_test, cmap='viridis', alpha=0.6,
                      label='Teste (250 pts)', marker='^')

# Configurações do gráfico
ax.set_title('Visualização 3D dos Dados Gerados com $f(x)$ como cor', fontsize=14)
ax.set_xlabel('x1', fontsize=14)
ax.set_ylabel('x2', fontsize=14)
ax.set_zlabel('x3', fontsize=14)

# Adicionando barra de cores
cbar = fig.colorbar(sc_train, ax=ax, pad=0.1)
cbar.set_label('Valor de f(x)', rotation=270, labelpad=20)

# Ajustando a legenda
ax.legend(loc='upper left', bbox_to_anchor=(-0.2, 0.9))

# Ajustando o ângulo de visualização
ax.view_init(elev=25, azim=45)
plt.show()
```

Visualização 3D dos Dados Gerados com $f(x)$ como cor

Treinamento

Como estamos tratando de um problema de regressão, uma vez que não temos as classes dos pontos bem definidas, temos que ajustar as métricas e os modelos para atuarem como regressores.

- Modelos:
 - MLP-BP de Três Camadas: usaremos o [MLPRegressor da Scikit-Learn](#)
 - RBF de Três Camadas: construí uma rotina própria de RBF com k-means
 - SVM com Kernel: usaremos o [SVR da Scikit-Learn](#)
- Métricas:
 - Mean Squared Error: usaremos o [MSE da Scikit-Learn](#) para analisar o quanto o modelo está errando
 - R^2 Score: usaremos o [\$R^2\$ da Scikit-Learn](#) para analisar o desempenho global dos modelos

Função para avaliação de modelos

```
In [8]: def evaluate_model(model, X_train, y_train, X_val, y_val, X_test, y_test):
    """Avalia um modelo e retorna métricas"""
    model.fit(X_train, y_train)

    # Previsões
    y_pred_train = model.predict(X_train)
    y_pred_val = model.predict(X_val)
    y_pred_test = model.predict(X_test)
```

```

# Métricas
mse_train = mean_squared_error(y_train, y_pred_train)
mse_val = mean_squared_error(y_val, y_pred_val)
mse_test = mean_squared_error(y_test, y_pred_test)

r2_train = r2_score(y_train, y_pred_train)
r2_val = r2_score(y_val, y_pred_val)
r2_test = r2_score(y_test, y_pred_test)

return {
    'mse': {'train': mse_train, 'val': mse_val, 'test': mse_test},
    'r2': {'train': r2_train, 'val': r2_val, 'test': r2_test}
}, y_pred_train,y_pred_val,y_pred_test

```

Normalização dos Dados

Fazemos isso para que os valores numéricos não estourem. Para tal, usamos o `StandardScaler` que atua como um z-score:

$$z = \frac{x - \mu}{\sigma}$$

```
In [9]: scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

y_train_scaled = scaler.fit_transform(y_train.reshape(-1, 1)).ravel()
y_val_scaled = scaler.transform(y_val.reshape(-1, 1)).ravel()
y_test_scaled = scaler.transform(y_test.reshape(-1, 1)).ravel()
```

(c) MLP-BP de três camadas

Implementei um MLP de 3 camadas em que:

- Camada escondida contém apenas 20 neurônios;
- Função de ativação para o cálculo do Net é a 'ReLU' ≡ Rectified Linear Unit Function
- Regra de Atualização dos Pesos: 'adam'
- Número Máximo de Iterações: 5000

```
In [10]: print("\n(c) Treinando MLP-BP de três camadas...")
mlp = make_pipeline(
    StandardScaler(),
    MLPRegressor(hidden_layer_sizes=(20,), activation='relu',
                 solver='adam', max_iter=5000, random_state=42)
)

mlp_metrics, y_mlp_pred_train,y_mlp_pred_val,y_mlp_pred_test = evaluate_model(ml
print("MSE - Treino:", mlp_metrics['mse']['train'], "| Validação:", mlp_metrics[
print("R² - Treino:", mlp_metrics['r2']['train'], "| Validação:", mlp_metrics['r
```

(c) Treinando MLP-BP de três camadas...

MSE - Treino: 0.022905393168026773 | Validação: 0.02348598790812087 | Teste: 0.031864816193016565

R^2 - Treino: 0.9770946068319732 | Validação: 0.9787385769190651 | Teste: 0.9712385445357153

(d) RBF de três camadas

Construí uma RBF de forma que:

- Usa K-means para encontrar centros representativos nos dados em que $K = 50$ centros;
- Calcula a ativação de cada neurônio RBF usando função Gaussiana:
 $\varphi(\|\vec{x}_n - c_i\|) = \exp(-\|\vec{x}_n - c_i\|^2/(2\sigma^2))$, o que faz com que cada neurônio responda mais intensamente a pontos próximos ao seu centro;
- Camada de saída com combinação linear das ativações RBF;
- Usa pseudoinversa para resolver o sistema linear (mais estável que inversão direta);

```
In [11]: from scipy.spatial.distance import cdist
from sklearn.base import BaseEstimator, RegressorMixin

print("\n(d) Treinando RBF de três camadas...")

class RBFNetwork(BaseEstimator, RegressorMixin):
    def __init__(self, n_centers=50, sigma=1.0):
        self.n_centers = n_centers # Quantidade de Neurônios na Camada Oculta
        self.sigma = sigma # Parâmetro de Largura das funções RBF

    def _rbf_function(self, X, centers):
        # Usando cdist para calcular as distâncias euclidianas
        return np.exp(-cdist(X, centers, 'squared_euclidean') / (2 * self.sigma**2))

    def fit(self, X, y):
        # Camada 1: Encontrar centros com K-means
        kmeans = KMeans(n_clusters=self.n_centers, random_state=42)
        kmeans.fit(X)
        self.centers_ = kmeans.cluster_centers_

        # Camada 2: Calcular saídas RBF
        rbf_features = self._rbf_function(X, self.centers_)

        # Camada 3: Regressão Linear (usando pseudoinversa para estabilidade numérica)
        self.weights_ = np.linalg.pinv(rbf_features) @ y
        return self

    def predict(self, X):
        rbf_features = self._rbf_function(X, self.centers_)
        return rbf_features @ self.weights_

rbf = make_pipeline(
    StandardScaler(),
    RBFNetwork()
)

rbf_metrics, y_rbf_pred_train, y_rbf_pred_val, y_rbf_pred_test = evaluate_model(rbf)
```

```

print("MSE - Treino:", rbf_metrics['mse']['train'], "| Validação:", rbf_metrics['mse']['val'], "| Teste:", rbf_metrics['mse']['test'])
print("R² - Treino:", rbf_metrics['r2']['train'], "| Validação:", rbf_metrics['r2']['val'], "| Teste:", rbf_metrics['r2']['test'])

(d) Treinando RBF de três camadas...
MSE - Treino: 0.13515109264031638 | Validação: 0.13925535290069407 | Teste: 0.15757398711348578
R² - Treino: 0.8648489073596837 | Validação: 0.8739347484172555 | Teste: 0.8577723723481726

```

(e) SVM com Kernel

Implementei um SVM Regressor (SVR) de tal forma que:

- Kernel do SVR é RBF;
 - Kernel Gaussiano: $\mathbf{K}(x, x') = \exp(-\gamma * \|x - x'\|^2)$
 - γ : controla a "largura";
 - $\|x - x'\|^2$: é a distância euclidiana ao quadrado.
 - Com essa Transformação de Núcleo, conseguimos mapear o problema para um espaço de dimensão superior e fazer a regressão
- gamma: Controla o alcance da influência de cada ponto de treinamento
 - Valores baixos: decisão mais suave (considera pontos mais distantes);
 - Valores altos: decisão mais complexa (considera pontos mais próximos);

```

In [12]: print("\n(e) Treinando SVM com Kernel...")
svm = make_pipeline(
    StandardScaler(),
    SVR(kernel='rbf', gamma=0.2)
)

svm_metrics, y_svm_pred_train,y_svm_pred_val,y_svm_pred_test = evaluate_model(svm)
print("MSE - Treino:", svm_metrics['mse']['train'], "| Validação:", svm_metrics['mse']['val'], "| Teste:", svm_metrics['mse']['test'])
print("R² - Treino:", svm_metrics['r2']['train'], "| Validação:", svm_metrics['r2']['val'], "| Teste:", svm_metrics['r2']['test'])

(e) Treinando SVM com Kernel...
MSE - Treino: 0.2015156436318011 | Validação: 0.22930130905565943 | Teste: 0.2406103668256992
R² - Treino: 0.7984843563681989 | Validação: 0.7924178380778762 | Teste: 0.782823026256176

```

(f) Comparação dos resultados no conjunto de teste

```

In [13]: print("\nComparação dos modelos:")
print("\nMLP-BP:")
print(f"MSE Teste: {mlp_metrics['mse']['test']:.4f} | R² Teste: {mlp_metrics['r2']['test']:.4f}")

print("\nRBF:")
print(f"MSE Teste: {rbf_metrics['mse']['test']:.4f} | R² Teste: {rbf_metrics['r2']['test']:.4f}")

print("\nSVM:")
print(f"MSE Teste: {svm_metrics['mse']['test']:.4f} | R² Teste: {svm_metrics['r2']['test']:.4f}")

# Plotar comparação
models = ['MLP-BP', 'RBF', 'SVM']
mse_test = [mlp_metrics['mse']['test'], rbf_metrics['mse']['test'], svm_metrics['mse']['test']]
r2_test = [mlp_metrics['r2']['test'], rbf_metrics['r2']['test'], svm_metrics['r2']['test']]

```

```

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.bar(models, mse_test, color=['blue', 'green', 'red'])
plt.title('Comparação de MSE no Conjunto de Teste')
plt.ylabel('MSE')

plt.subplot(1, 2, 2)
plt.bar(models, r2_test, color=['blue', 'green', 'red'])
plt.title('Comparação de R2 no Conjunto de Teste')
plt.ylabel('R2')

plt.tight_layout()
plt.show()

```

Comparação dos modelos:

MLP-BP:

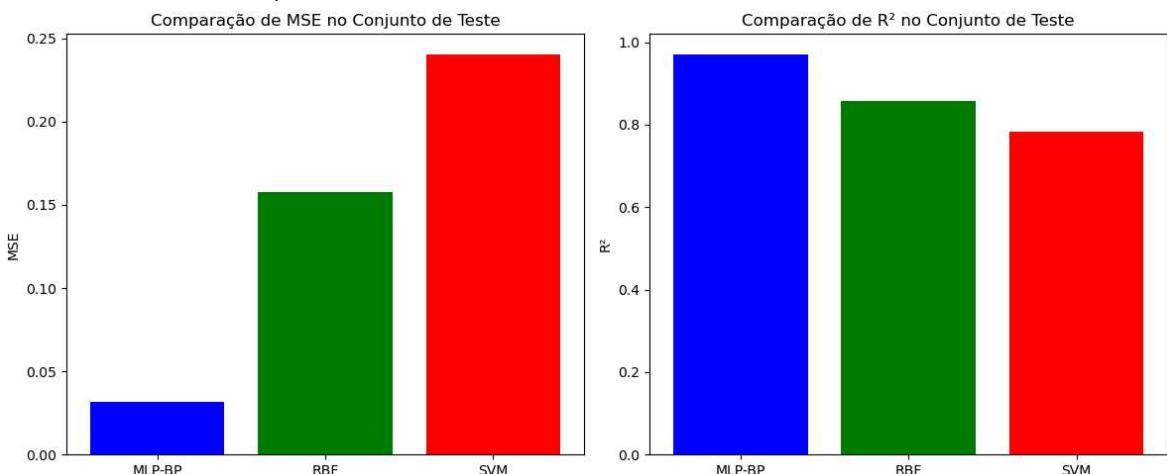
MSE Teste: 0.0319 | R² Teste: 0.9712

RBF:

MSE Teste: 0.1576 | R² Teste: 0.8578

SVM:

MSE Teste: 0.2406 | R² Teste: 0.7828



Gerando gráficos de comparação para MLP, RBF e SVM

```

In [15]: fi,ax=plt.subplots(1,3,sharex=False,figsize=(16,5))

# Gráfico MLP
ax[0].scatter(y_test_scaled, y_mlp_pred_test, alpha=0.6, label='MLP')
ax[0].plot([min(y_test_scaled), max(y_test_scaled)], [min(y_test_scaled), max(y_test_scaled)])
ax[0].set_xlabel("Valor Real")
ax[0].set_ylabel("Valor Preditivo")
ax[0].set_title("MLP: Real vs Preditivo")
ax[0].grid(linestyle='dashed')

# Gráfico RBF
ax[1].scatter(y_test_scaled, y_rbf_pred_test, alpha=0.6, label='RBF', color='orange')
ax[1].plot([min(y_test_scaled), max(y_test_scaled)], [min(y_test_scaled), max(y_test_scaled)])
ax[1].set_xlabel("Valor Real")
ax[1].set_title("RBF: Real vs Preditivo")
ax[1].grid(linestyle='dashed')

# Gráfico SVM

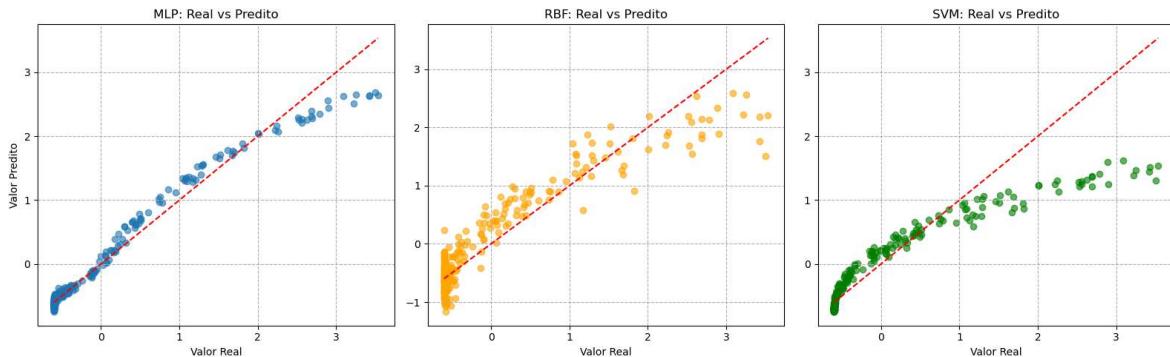
```

```

ax[2].scatter(y_test_scaled, y_svm_pred_test, alpha=0.6, label='SVM', color='green')
ax[2].plot([min(y_test_scaled), max(y_test_scaled)], [min(y_test_scaled), max(y_test_scaled)])
ax[2].set_xlabel("Valor Real")
ax[2].set_title("SVM: Real vs Predito")
ax[2].grid(linestyle='dashed')

plt.tight_layout()
plt.show()

```



Relatório

1. O MLP-BP apresentou o melhor desempenho geral, com menor MSE e maior R^2 no conjunto de teste. Isso era esperado, pois redes neurais multicamadas são aproximadores universais de funções. O principal destaque vai para a rápida convergência do modelo que com menos robustez que os outros modelos (converge com menos de 1000 iterações e possui apenas 20 neurônios na camada oculta), consegue ter um desempenho superior aos outros dois. Contudo, conseguiríamos ajustar os outros modelos para que desempenhassem de forma parecida (ponto 4. deste breve relatório), porém aumentando suas complexidades.
2. O RBF teve desempenho intermediário. Ele funciona bem para problemas com características radialmente simétricas, mas pode ter dificuldade com funções mais complexas.
3. O SVM com kernel RBF teve o "pior" desempenho neste problema. SVMs são poderosos para classificação binária, mas podem ser menos eficientes para problemas de regressão complexos como este.
4. Há modos de aumentar as respectivas acurárias (aqui, vistas pelo score R^2). O problema passa a ser analisar se não há overfitting dos modelos ao aumentar seus desempenhos:
 - MLP-BP: Aumentar o número de neurônios na camada escondida
 - RBF: Aumentar o número de centros na região de decisão
 - SVM com Kernel RBF: Aumentar o parâmetro gamma que controla o alcance da influência dos pontos no problema
5. A função proposta parece ser melhor aproximada pela arquitetura do MLP-BP.