UNIVERSIDADE FEDERAL
DE MINAS GERAIS

ATIVIDADE AVALIATIVA
TREINAMENTO RBF

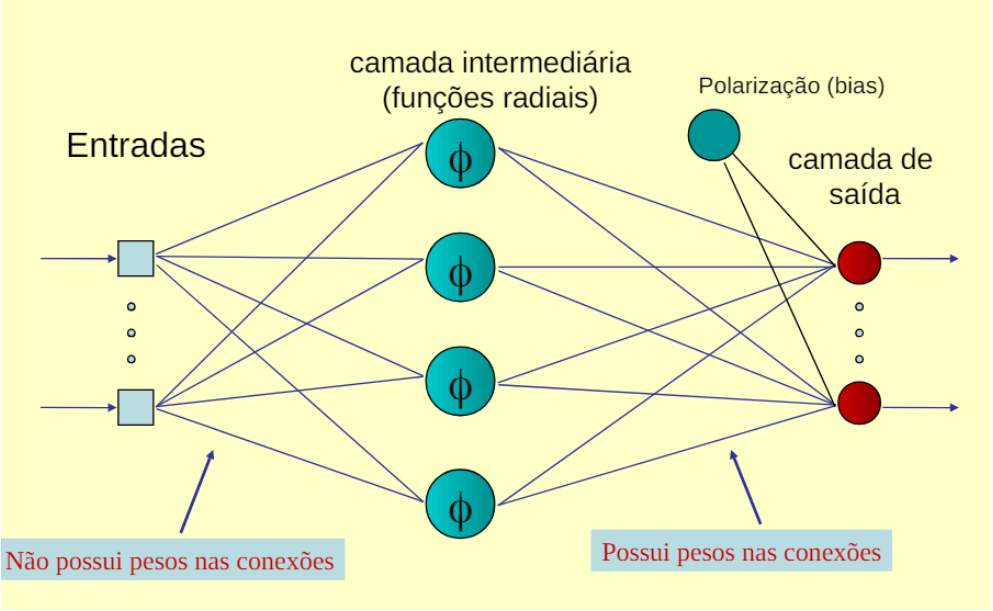
ALUNO: FELIPPE VELOSO MARINHO
MATRÍCULA: 2021072260
DISCIPLINA: REDES NEURAIS ARTIFICIAIS

Redes Neurais RBF (Radial Basis Function Neural Networks) são funções de base radial, não-lineares que podem ser utilizadas como funções-base em qualquer tipo de modelo de regressão não-linear (linear ou não-linear nos parâmetros) e como função de ativação de qualquer tipo de rede multicamada. As redes com função de base radial apresentam 3 diferenças das redes MLP:

1) Elas sempre apresentam uma única camada intermediária;

2) Os neurônios de saída são sempre lineares;

3) Os neurônios da camada intermediária têm uma função de base radial como função de ativação, ao invés de uma função sigmoidal ou outras.



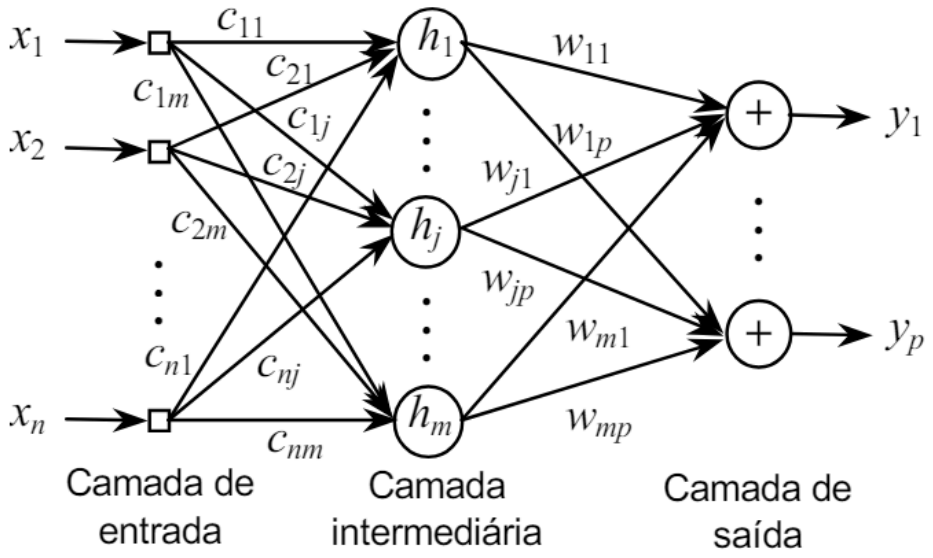


Figura 3 – Rede RBF com múltiplas saídas.

Métodos de treinamentos

Geralmente os modelos de treinamento propostos na literatura são divididos em duas partes:

- Definição dos centros, da forma e da dispersão das funções de base radial, normalmente baseada em aprendizado não supervisionado ou computação evolutiva.
- Adaptação supervisionada dos pesos da camada de saída, responsáveis pela combinação linear das ativações da camada intermediária, empregando técnicas como pseudo-inversão.

Dentre outros métodos, o que será feito no treinamento do exercício proposto será:

1 - Selecionar número de nodos na camada intermediária de acordo com informações sobre o conjunto de dados;

2 - Selecionar parâmetros para cada uma das funções radiais (centros e raios);

3 - Obter conjunto de pesos para a camada de saída;

Problema analisado

Abaixo será feita a documentação da implementação e treinamento de uma RBF (Radial Basis Function Neural Networks) em um problema de classificação de uma amostra de dados distribuídas em quatro gaussianas

com as classes mostradas de maneira alternada, assim como o Problema XOR.

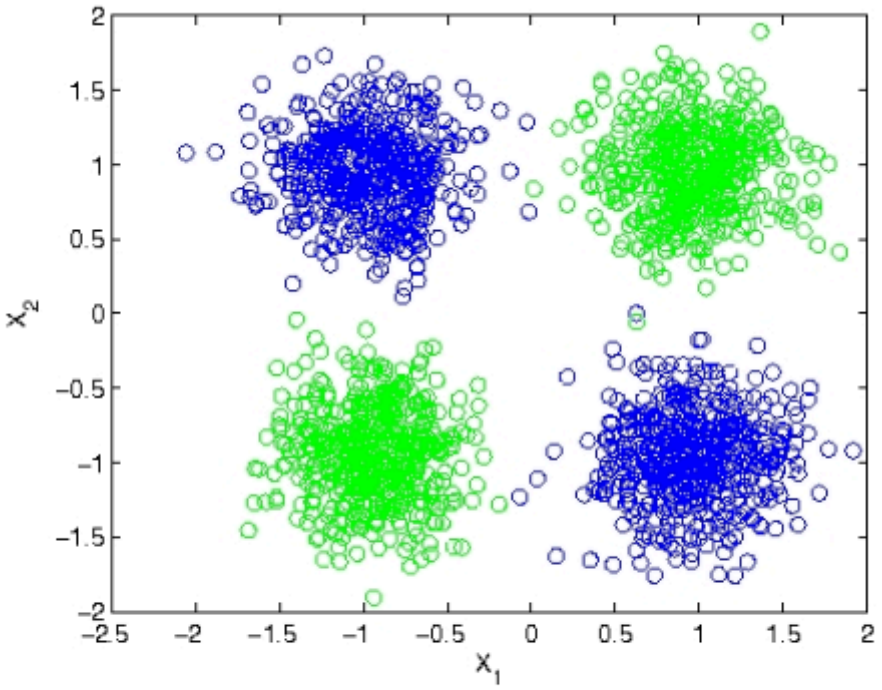


Figure 1: Problema XOR

```
def generate_xor_data(n_samples, mean, std):
    np.random.seed(42)

    # Generate Gaussian blobs for XOR
    X1 = np.random.normal(loc=mean, scale=std, size=(n_samples, 2))
    X2 = np.random.normal(loc=[mean[0], -mean[1]], scale=std,
size=(n_samples, 2))
    X3 = np.random.normal(loc=[-mean[0], mean[1]], scale=std,
size=(n_samples, 2))
    X4 = np.random.normal(loc=[-mean[0], -mean[1]], scale=std,
size=(n_samples, 2))

    # Assign labels to create XOR problem
    y1 = np.zeros(n_samples)
    y2 = np.ones(n_samples)
    y3 = np.ones(n_samples)
    y4 = np.zeros(n_samples)

    # Concatenate all data points
    X = np.vstack((X1, X2, X3, X4))
```

```
    y = np.hstack((y1, y2, y3, y4))

    return X, y
```

O Problema também tinha como condição que os dados fossem separados em um conjunto de treinamento com 90% dos dados e um conjunto de testes com 10% dos dados de forma aleatória. Faça 10 simulações e calcule a acurácia média e o desvio padrão das soluções. Gere um relatório contendo os resultados das simulações e o gráfico com o melhor hiperplano de separação. Use a função de base radial gaussiana que está no slide da aula.

A única diferença entre os requisitos e o modelo desenvolvido foi justamente na utilização de superfícies de separação. No enunciado, é pedido para que sejam utilizados hiperplano, porém como um problema de classificação em RBF comum foram utilizadas gaussianas ou hiper-elipsóides para particionar os conjuntos de dados.

- Redes RBF *X* MLP
  - MLP utiliza hiperplanos para particionar espaço de entradas (camada intermediária)
    - Definidos por funções da forma $f(\sum w_i x_i)$
  - RBF utiliza hiper-elipsóides para particionar o espaço de entradas (camada intermediária)
    - Definidos por funções da forma $\phi(||x_i - \mu_i||)$
    - Distância do vetor de entrada a um centro

A atividade também foi utilizada como forma de aprendizado da biblioteca SKLEARN em Python. Biblioteca muito popular em Machine Learning.

Para gerar o modelo utilizando a biblioteca sklearn foi criado uma classe para o modelo RBF:

```
class RBFNetwork:
    def __init__(self, n_clusters, sigma):
        self.n_clusters = n_clusters
        self.sigma = sigma
```

```
        self.kmeans = KMeans(n_clusters=self.n_clusters)
        self.scaler = StandardScaler()
        self.classifier = LogisticRegression()

    def _rbf(self, X, centers):
            return np.exp(-np.linalg.norm(X[:, np.newaxis] - centers,
axis=2) ** 2)

    def fit(self, X, y):
        X_scaled = self.scaler.fit_transform(X)
        self.kmeans.fit(X_scaled)
        centers = self.kmeans.cluster_centers_
        transformed_X = self._rbf(X_scaled, centers)
        self.classifier.fit(transformed_X, y)

    def predict(self, X):
        X_scaled = self.scaler.transform(X)
        centers = self.kmeans.cluster_centers_
        transformed_X = self._rbf(X_scaled, centers)
        return self.classifier.predict(transformed_X)

    def get_params(self):
            return self.kmeans.cluster_centers_, self.classifier.coef_,
self.classifier.intercept_
```

Essa classe, assim como explicado nos slides possuem na camada oculta, neurônios que aplicam uma função radial (RBF) às entradas. A função gaussiana foi definida no enunciado, sendo uma adaptação da função em R dada no slide. Ela foi definida pelo objeto _rbf da classe responsável pela rede.

O kmeans é um método de segregar em torno de centros (centróides) diversos dados. No código ele é utilizado através da função importada pelo sklearn, onde passamos no parâmetro o número de clusters que o kmeans deve encontrar (4).

O StndartScaler do sklearn é utilizado para normalizar os dados, garantindo a variância com média entre 0 e 1.

Por fim, o LogisticRegression é outro método do sklearn responsável pela classificação na camada de saída. Após a aplicação da função gaussiana na

camada oculta, os dados transformados são passados para a regressão logística para classificação.

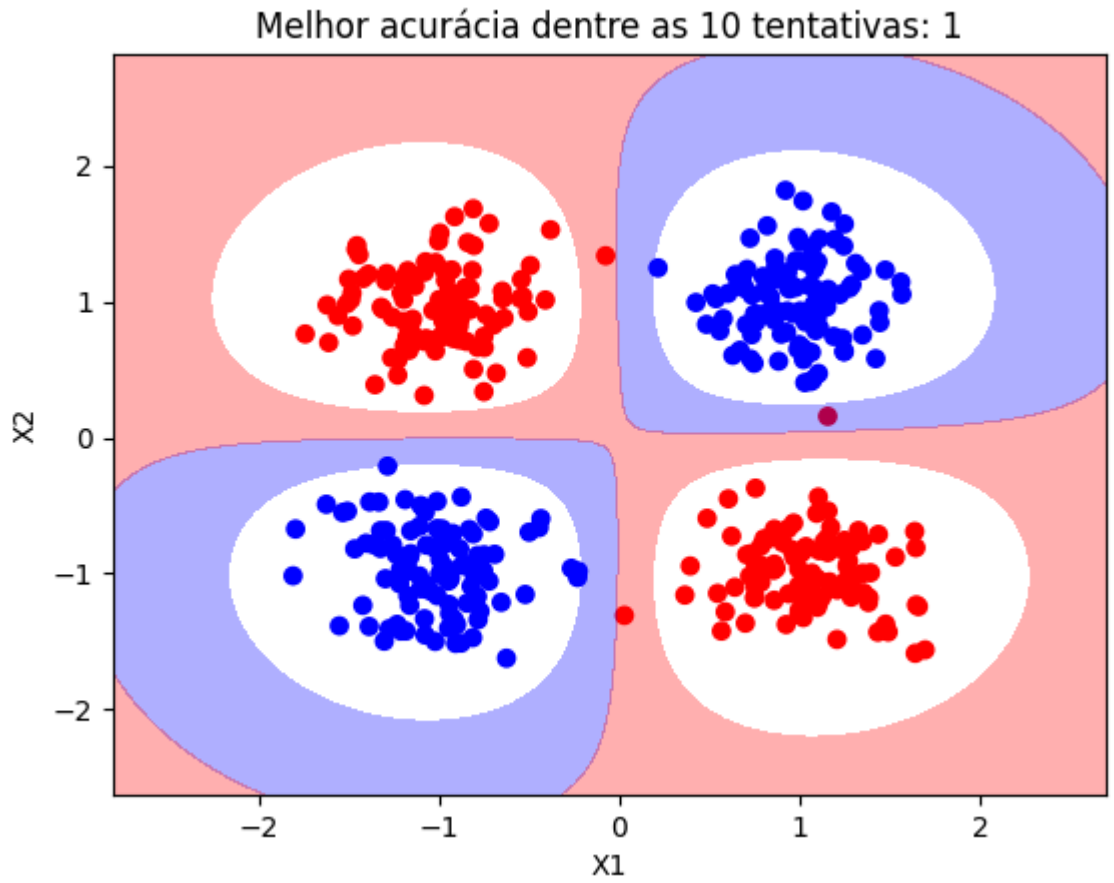
```
for _ in range(n_simulations):
    # Split data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.1, random_state=None)

    # Train RBF Network
    rbf_network = RBFNetwork(n_clusters=4, sigma=-sigma)
    rbf_network.fit(X_train, y_train)

    # Predict on test set
    y_pred = rbf_network.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

    print('-----------------------------------')
    print('Simulação ', _ + 1)

    # Print accuracy for each simulation
    print(f'Acurácia: {accuracy * 100:.2f}%')

    # Print desvio padrão for each simulation
        print(f'Desvio Padrão da Acurácia: {np.std(accuracies) *
100:.2f}%')

    # Save the best model
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_accuracy_number = _ + 1
                        best_centers, best_coef, best_intercept =
rbf_network.get_params()
```

Depois disso é percorrido o total de simulações pedidos (10), os dados são separados para teste. A acurácia é calculada para cada simulação e o melhor modelo é salvo para ser plotado.

Por fim, esses são os resultados:

```
-----------------------------------
```

```
Simulação  1
Acurácia: 100.00%
Desvio Padrão da Acurácia: 0.00%
-----------------------------------
Simulação  2
Acurácia: 100.00%
Desvio Padrão da Acurácia: 0.00%
-----------------------------------
Simulação  3
Acurácia: 100.00%
Desvio Padrão da Acurácia: 0.00%
-----------------------------------
Simulação  4
Acurácia: 100.00%
Desvio Padrão da Acurácia: 0.00%
-----------------------------------
Simulação  5
Acurácia: 100.00%
Desvio Padrão da Acurácia: 0.00%
-----------------------------------
Simulação  6
Acurácia: 97.50%
Desvio Padrão da Acurácia: 0.93%
-----------------------------------
Simulação  7
Acurácia: 100.00%
Desvio Padrão da Acurácia: 0.87%
-----------------------------------
Simulação  8
Acurácia: 97.50%
Desvio Padrão da Acurácia: 1.08%
-----------------------------------
Simulação  9
Acurácia: 100.00%
Desvio Padrão da Acurácia: 1.04%
-----------------------------------
Simulação  10
Acurácia: 100.00%
Desvio Padrão da Acurácia: 1.00%
Mean Accuracy: 99.50%
Standard Deviation: 1.00%
```



O plot realizado foi somente o da melhor acurácia dentre os 10 testes.