

# Engenharia de Redes Neurais Artificiais

Aluno: Matheus Paul Lopuch

Exercícios das Aulas

01b\_Representacao\_Redes\_Neurnais\_Artificiais.ipynb

## Exercícios

1. Execute a célula de treino do *perceptron* algumas vezes e verifique os resultados. Depois, realize o mesmo com o treino do *perceptron* multicamadas. Houveram diferenças? Discuta quais são as diferenças e justifique.
2. Modifique a chamada do *Perceptron Multicamadas* para utilizar apenas um *perceptron* na camada oculta. Depois, utilize apenas dois *perceptrons*. Foi possível resolver o problema não-linear? Discuta a resposta e justifique.
3. Reproduza o código do *Multilayer Perceptron* utilizando Pytorch.

## Respostas:

1. O perceptron simples limita-se a problemas linearmente separáveis, enquanto o multicamadas alcança tarefas envolvendo relações complexas graças à presença de camadas ocultas e funções não-lineares de ativação como a sigmoide. Perceptron falha em separar corretamente as classes e o MLP consegue uma separação muito melhor, ajustando-se à complexidade da distribuição dos dados.

2. Com apenas um perceptron oculto, o modelo ainda realiza uma separação linear (não resolve bem problemas não-lineares). Com dois, consegue separar um pouco mais do problema, mas ainda não é plenamente eficaz no caso de conjuntos como XOR. A partir de três ou mais unidades ocultas, o MLP passa a representar bordas realmente não-lineares e solucionar bem o problema.

O número de perceptrons na camada oculta influencia a capacidade de aproximação da superfície de decisão. Um só é insuficiente para problemas complexos, mais unidades proporcionam representatividade capaz de capturar padrões não-lineares.

3.

```
import torch
import torch.nn as nn
import torch.optim as optim
```

# Dados

```
X = torch.tensor(X_data, dtype=torch.float32)
y = torch.tensor(y_data, dtype=torch.float32).view(-1, 1)
```

# Definição do modelo

```

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        return x

# Instância e treinamento
model = MLP(input_dim=X.shape[1], hidden_dim=3)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(1000):
    optimizer.zero_grad()
    output = model(X)
    loss = criterion(output, y)
    loss.backward()
    optimizer.step()

# Predição (exemplo)
with torch.no_grad():
    y_pred = (model(X) > 0.5).int()

```