```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from itertools import product
import warnings
warnings.filterwarnings('ignore')

# Configurar estilo dos gráficos
plt.style.use('default')
sns.set_palette("husl")

class ActivationFunctions:
    """Classe com diferentes funções de ativação e suas derivadas"""

    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-np.clip(x, -250, 250)))

    @staticmethod
    def sigmoid_derivative(x):
        return x * (1 - x)

    @staticmethod
    def tanh(x):
        return np.tanh(x)

    @staticmethod
    def tanh_derivative(x):
        return 1 - x * x

    @staticmethod
    def relu(x):
        return np.maximum(0, x)

    @staticmethod
    def relu_derivative(x):
        return (x > 0).astype(float)

    @staticmethod
    def step(x):
        return (x >= 0).astype(int)

    @staticmethod
```

```python
    def step_derivative(x):
        return np.ones_like(x)

class DataGenerator:
    """Classe para gerar dados de treinamento para funções lógicas"""

    @staticmethod
    def generate_logic_data(function_name, n_inputs):
        """
        Gera dados de treinamento para funções lógicas

        Args:
            function_name: 'AND', 'OR', ou 'XOR'
            n_inputs: número de entradas booleanas

        Returns:
            X: matriz de entradas (2^n_inputs x n_inputs)
            y: vetor de saídas (2^n_inputs,)
        """
        # Gerar todas as combinações possíveis de entradas
        combinations = list(product([0, 1], repeat=n_inputs))
        X = np.array(combinations)
        y = np.zeros(len(combinations))

        for i, inputs in enumerate(combinations):
            if function_name == 'AND':
                y[i] = int(all(inputs))
            elif function_name == 'OR':
                y[i] = int(any(inputs))
            elif function_name == 'XOR':
                y[i] = int(sum(inputs) % 2)

        return X, y

class Perceptron:
    """Implementação do algoritmo Perceptron"""

    def __init__(self, n_inputs, learning_rate=0.1, use_bias=True,
random_seed=42):
        """
        Inicializa o Perceptron

        Args:
```

```
            n_inputs: número de entradas
            learning_rate: taxa de aprendizado
            use_bias: se deve usar bias
            random_seed: semente para reprodutibilidade
        """
        np.random.seed(random_seed)
        self.n_inputs = n_inputs
        self.learning_rate = learning_rate
        self.use_bias = use_bias

        # Inicializar pesos aleatoriamente
        weight_size = n_inputs + (1 if use_bias else 0)
        self.weights = np.random.uniform(-1, 1, weight_size)

        # Histórico de treinamento
        self.training_history = []
        self.weight_history = []

    def _add_bias(self, X):
        """Adiciona coluna de bias se necessário"""
        if self.use_bias:
            return np.column_stack([X, np.ones(X.shape[0])])
        return X

    def predict(self, X):
        """Faz predições usando função degrau"""
        X_with_bias = self._add_bias(X)
        return ActivationFunctions.step(np.dot(X_with_bias,
self.weights))

    def train(self, X, y, max_epochs=100, verbose=False):
        """
        Treina o Perceptron

        Args:
            X: matriz de entradas
            y: vetor de saídas esperadas
            max_epochs: número máximo de épocas
            verbose: se deve imprimir progresso

        Returns:
            dict com resultados do treinamento
        """
```

```
        X_with_bias = self._add_bias(X)

        for epoch in range(max_epochs):
            total_error = 0
            n_errors = 0

            for i in range(len(X)):
                # Forward pass
                prediction =
ActivationFunctions.step(np.dot(X_with_bias[i], self.weights))
                error = y[i] - prediction

                # Backward pass (atualização dos pesos)
                if error != 0:
                    self.weights += self.learning_rate * error *
X_with_bias[i]
                    n_errors += 1

                total_error += abs(error)

            # Salvar histórico
            predictions = self.predict(X)
            accuracy = np.mean(predictions == y)

            self.training_history.append({
                'epoch': epoch,
                'total_error': total_error,
                'n_errors': n_errors,
                'accuracy': accuracy
            })
            self.weight_history.append(self.weights.copy())

            if verbose and epoch % 10 == 0:
                print(f"Época {epoch}: Erro = {total_error}, Acurácia =
{accuracy:.3f}")

            # Convergência (sem erros)
            if total_error == 0:
                if verbose:
                    print(f"Convergiu na época {epoch}")
                break

        final_predictions = self.predict(X)
```

```python
        final_accuracy = np.mean(final_predictions == y)

        return {
            'converged': total_error == 0,
            'epochs': epoch + 1,
            'final_accuracy': final_accuracy,
            'final_weights': self.weights.copy(),
            'history': self.training_history
        }

class MLP:
    """Implementação de Rede Neural Multicamadas com Backpropagation"""

    def __init__(self, input_size, hidden_size, output_size=1,
learning_rate=0.1,
                 activation='sigmoid', use_bias=True, random_seed=42):
        """
        Inicializa a MLP

        Args:
            input_size: número de entradas
            hidden_size: número de neurônios na camada escondida
            output_size: número de saídas
            learning_rate: taxa de aprendizado
            activation: função de ativação ('sigmoid', 'tanh', 'relu')
            use_bias: se deve usar bias
            random_seed: semente para reprodutibilidade
        """
        np.random.seed(random_seed)
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.use_bias = use_bias

        # Definir funções de ativação
        if activation == 'sigmoid':
            self.activation = ActivationFunctions.sigmoid
            self.activation_derivative =
ActivationFunctions.sigmoid_derivative
        elif activation == 'tanh':
            self.activation = ActivationFunctions.tanh
```

```
            self.activation_derivative =
ActivationFunctions.tanh_derivative
        elif activation == 'relu':
            self.activation = ActivationFunctions.relu
            self.activation_derivative =
ActivationFunctions.relu_derivative

        # Inicializar pesos (Xavier initialization)
        input_weights_size = input_size + (1 if use_bias else 0)
        hidden_weights_size = hidden_size + (1 if use_bias else 0)

        self.weights_input_hidden = np.random.uniform(
            -np.sqrt(6/(input_size + hidden_size)),
            np.sqrt(6/(input_size + hidden_size)),
            (input_weights_size, hidden_size)
        )

        self.weights_hidden_output = np.random.uniform(
            -np.sqrt(6/(hidden_size + output_size)),
            np.sqrt(6/(hidden_size + output_size)),
            (hidden_weights_size, output_size)
        )

        # Histórico de treinamento
        self.training_history = []

    def _add_bias(self, X):
        """Adiciona coluna de bias se necessário"""
        if self.use_bias:
            return np.column_stack([X, np.ones(X.shape[0])])
        return X

    def forward(self, X):
        """Forward pass através da rede"""
        # Camada de entrada para escondida
        X_with_bias = self._add_bias(X)
        hidden_input = np.dot(X_with_bias, self.weights_input_hidden)
        hidden_output = self.activation(hidden_input)

        # Camada escondida para saída
        hidden_with_bias = self._add_bias(hidden_output)
        output_input = np.dot(hidden_with_bias,
self.weights_hidden_output)
```

```
        output = self.activation(output_input)

        return {
            'hidden_input': hidden_input,
            'hidden_output': hidden_output,
            'output_input': output_input,
            'output': output,
            'X_with_bias': X_with_bias,
            'hidden_with_bias': hidden_with_bias
        }

    def backward(self, X, y, forward_result):
        """Backward pass (backpropagation)"""
        batch_size = X.shape[0]

        # Erro na camada de saída
        output_error = y.reshape(-1, 1) - forward_result['output']
        output_delta = output_error *
self.activation_derivative(forward_result['output'])

        # Gradiente para pesos da camada escondida para saída
        hidden_output_grad =
np.dot(forward_result['hidden_with_bias'].T, output_delta) / batch_size

        # Erro na camada escondida
        if self.use_bias:
            hidden_error = np.dot(output_delta,
self.weights_hidden_output[:-1].T)
        else:
            hidden_error = np.dot(output_delta,
self.weights_hidden_output.T)

        hidden_delta = hidden_error *
self.activation_derivative(forward_result['hidden_output'])

        # Gradiente para pesos da camada de entrada para escondida
        input_hidden_grad = np.dot(forward_result['X_with_bias'].T,
hidden_delta) / batch_size

        # Atualizar pesos
        self.weights_hidden_output += self.learning_rate *
hidden_output_grad
```

```
        self.weights_input_hidden += self.learning_rate *
input_hidden_grad

    def train(self, X, y, max_epochs=1000, verbose=False):
        """Treina a MLP"""
        for epoch in range(max_epochs):
            # Forward pass
            forward_result = self.forward(X)

            # Calcular erro e acurácia
            predictions = (forward_result['output'] >
0.5).astype(int).flatten()
            accuracy = np.mean(predictions == y)
            mse = np.mean((y.reshape(-1, 1) - forward_result['output'])
** 2)

            # Salvar histórico
            self.training_history.append({
                'epoch': epoch,
                'mse': mse,
                'accuracy': accuracy
            })

            if verbose and epoch % 100 == 0:
                print(f"Época {epoch}: MSE = {mse:.6f}, Acurácia =
{accuracy:.3f}")

            # Verificar convergência
            if accuracy == 1.0:
                if verbose:
                    print(f"Convergiu na época {epoch}")
                break

            # Backward pass
            self.backward(X, y, forward_result)

        return {
            'epochs': epoch + 1,
            'final_accuracy': accuracy,
            'final_mse': mse,
            'history': self.training_history
        }
```

```
    def predict(self, X):
        """Faz predições"""
        forward_result = self.forward(X)
        return (forward_result['output'] > 0.5).astype(int).flatten()

class Visualizer:
    """Classe para visualizações"""

    @staticmethod
    def plot_2d_decision_boundary(model, X, y, title,
model_type='perceptron'):
        """Plota boundary de decisão para problemas 2D"""
        if X.shape[1] != 2:
            print("Visualização disponível apenas para 2 entradas")
            return

        plt.figure(figsize=(10, 8))

        # Criar grid para visualizar boundary
        h = 0.01
        x_min, x_max = -0.5, 1.5
        y_min, y_max = -0.5, 1.5
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))

        # Fazer predições no grid
        grid_points = np.c_[xx.ravel(), yy.ravel()]
        if model_type == 'perceptron':
            Z = model.predict(grid_points)
        else:
            Z = model.predict(grid_points)
        Z = Z.reshape(xx.shape)

        # Plotar boundary
        plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)
        plt.contour(xx, yy, Z, colors='black', linestyles='--',
linewidths=1)

        # Plotar pontos de dados
        colors = ['red' if label == 0 else 'blue' for label in y]
        plt.scatter(X[:, 0], X[:, 1], c=colors, s=100,
edgecolors='black')
```

```
        # Adicionar labels aos pontos
        for i, (x, y_coord) in enumerate(X):
            plt.annotate(f'({int(x)},{int(y_coord)})→{int(y[i])}',
                         (x, y_coord), xytext=(5, 5), textcoords='offset
points')

        plt.xlim(x_min, x_max)
        plt.ylim(y_min, y_max)
        plt.xlabel('Entrada 1')
        plt.ylabel('Entrada 2')
        plt.title(title)
        plt.grid(True, alpha=0.3)
        plt.show()

    @staticmethod
    def plot_training_history(history, title):
        """Plota histórico de treinamento"""
        epochs = [h['epoch'] for h in history]

        if 'total_error' in history[0]:  # Perceptron
            errors = [h['total_error'] for h in history]
            accuracies = [h['accuracy'] for h in history]

            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

            ax1.plot(epochs, errors, 'b-', linewidth=2)
            ax1.set_xlabel('Época')
            ax1.set_ylabel('Erro Total')
            ax1.set_title('Erro vs Época')
            ax1.grid(True, alpha=0.3)

            ax2.plot(epochs, accuracies, 'g-', linewidth=2)
            ax2.set_xlabel('Época')
            ax2.set_ylabel('Acurácia')
            ax2.set_title('Acurácia vs Época')
            ax2.grid(True, alpha=0.3)
            ax2.set_ylim([0, 1.1])

        else:  # MLP
            mse = [h['mse'] for h in history]
            accuracies = [h['accuracy'] for h in history]

            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
```

```python
            ax1.plot(epochs, mse, 'r-', linewidth=2)
            ax1.set_xlabel('Época')
            ax1.set_ylabel('MSE')
            ax1.set_title('MSE vs Época')
            ax1.grid(True, alpha=0.3)
            ax1.set_yscale('log')

            ax2.plot(epochs, accuracies, 'g-', linewidth=2)
            ax2.set_xlabel('Época')
            ax2.set_ylabel('Acurácia')
            ax2.set_title('Acurácia vs Época')
            ax2.grid(True, alpha=0.3)
            ax2.set_ylim([0, 1.1])

        plt.suptitle(title)
        plt.tight_layout()
        plt.show()

    @staticmethod
    def plot_weight_evolution(weight_history, title):
        """Plota evolução dos pesos durante treinamento"""
        if not weight_history:
            return

        plt.figure(figsize=(12, 6))

        weight_history = np.array(weight_history)
        epochs = range(len(weight_history))

        for i in range(weight_history.shape[1]):
            plt.plot(epochs, weight_history[:, i], linewidth=2,
                    label=f'Peso {i+1}' if i <
weight_history.shape[1]-1 else 'Bias')

        plt.xlabel('Época')
        plt.ylabel('Valor do Peso')
        plt.title(title)
        plt.legend()
        plt.grid(True, alpha=0.3)
        plt.show()

class ExperimentRunner:
```

```
    """Classe para executar experimentos sistemáticos"""

    @staticmethod
    def run_perceptron_experiments():
        """Executa experimentos completos com Perceptron"""
        print("="*60)
        print("EXPERIMENTOS COM PERCEPTRON")
        print("="*60)

        functions = ['AND', 'OR', 'XOR']
        n_inputs_list = [2, 3, 4, 5]

        results = {}

        for func in functions:
            print(f"\n🔍 TESTANDO FUNÇÃO {func}")
            print("-" * 40)

            results[func] = {}

            for n_inputs in n_inputs_list:
                print(f"\nNúmero de entradas: {n_inputs}")

                # Gerar dados
                X, y = DataGenerator.generate_logic_data(func,
n_inputs)
                print(f"Dados gerados: {len(X)} amostras")

                # Treinar Perceptron
                perceptron = Perceptron(n_inputs, learning_rate=0.1,
use_bias=True)
                train_result = perceptron.train(X, y, max_epochs=100,
verbose=False)

                # Testar
                predictions = perceptron.predict(X)
                accuracy = np.mean(predictions == y)

                results[func][n_inputs] = {
                    'accuracy': accuracy,
                    'converged': train_result['converged'],
                    'epochs': train_result['epochs']
                }
```

```
                print(f"Acurácia: {accuracy:.1%}")
                print(f"Convergiu: {'Sim' if train_result['converged']
else 'Não'}")
                print(f"Épocas: {train_result['epochs']}")

                # Visualizar para 2 entradas
                if n_inputs == 2:
                    title = f"Perceptron - {func} (2 entradas)"
                    Visualizer.plot_2d_decision_boundary(perceptron, X,
y, title)

Visualizer.plot_training_history(train_result['history'],
                                                    f"Treinamento
{title}")

Visualizer.plot_weight_evolution(perceptron.weight_history,
                                                    f"Evolução dos Pesos
- {title}")

        # Resumo dos resultados
        print("\n" + "="*60)
        print("RESUMO DOS RESULTADOS - PERCEPTRON")
        print("="*60)

        for func in functions:
            print(f"\n{func}:")
            for n_inputs in n_inputs_list:
                result = results[func][n_inputs]
                status = "✅" if result['accuracy'] == 1.0 else "❌"
                print(f"  {n_inputs} entradas: {status}
{result['accuracy']:.1%} "
                      f"({result['epochs']} épocas)")

        return results

    @staticmethod
    def run_mlp_experiments():
        """Executa experimentos completos com MLP"""
        print("\n" + "="*60)
        print("EXPERIMENTOS COM MLP (BACKPROPAGATION)")
        print("="*60)
```

```python
        functions = ['AND', 'OR', 'XOR']
        activations = ['sigmoid', 'tanh', 'relu']
        learning_rates = [0.01, 0.1, 0.5]

        results = {}

        # Teste 1: Importância da função de ativação
        print("\n🧠 TESTE 1: IMPORTÂNCIA DA FUNÇÃO DE ATIVAÇÃO")
        print("-" * 50)

        for func in functions:
            print(f"\nFunção {func}:")
            results[func] = {}

            X, y = DataGenerator.generate_logic_data(func, 2)

            for activation in activations:
                mlp = MLP(input_size=2, hidden_size=4, 
learning_rate=0.1,
                         activation=activation, use_bias=True)
                train_result = mlp.train(X, y, max_epochs=1000, 
verbose=False)

                predictions = mlp.predict(X)
                accuracy = np.mean(predictions == y)

                results[func][activation] = {
                    'accuracy': accuracy,
                    'epochs': train_result['epochs']
                }

                print(f"  {activation:8}: {accuracy:.1%} 
({train_result['epochs']} épocas)")

                # Visualizar XOR com sigmoid (mais interessante)
                if func == 'XOR' and activation == 'sigmoid':
                    title = f"MLP - XOR com {activation}"
                    Visualizer.plot_2d_decision_boundary(mlp, X, y, 
title, 'mlp')
                    
Visualizer.plot_training_history(train_result['history'], 
                                                   f"Treinamento 
{title}")
```

```
        # Teste 2: Importância da taxa de aprendizado
        print(f"\n⚡ TESTE 2: IMPORTÂNCIA DA TAXA DE APRENDIZADO")
        print("-" * 50)

        X, y = DataGenerator.generate_logic_data('XOR', 2)

        for lr in learning_rates:
            mlp = MLP(input_size=2, hidden_size=4, learning_rate=lr,
                     activation='sigmoid', use_bias=True)
            train_result = mlp.train(X, y, max_epochs=1000,
verbose=False)

            predictions = mlp.predict(X)
            accuracy = np.mean(predictions == y)

            print(f"Taxa {lr:4.2f}: {accuracy:.1%}
({train_result['epochs']} épocas)")

        # Teste 3: Importância do bias
        print(f"\n🎯 TESTE 3: IMPORTÂNCIA DO BIAS")
        print("-" * 50)

        for use_bias in [True, False]:
            mlp = MLP(input_size=2, hidden_size=4, learning_rate=0.1,
                     activation='sigmoid', use_bias=use_bias)
            train_result = mlp.train(X, y, max_epochs=1000,
verbose=False)

            predictions = mlp.predict(X)
            accuracy = np.mean(predictions == y)

            bias_status = "Com bias" if use_bias else "Sem bias"
            print(f"{bias_status:9}: {accuracy:.1%}
({train_result['epochs']} épocas)")

        return results

    @staticmethod
    def demonstrate_xor_limitation():
        """Demonstra por que Perceptron não resolve XOR"""
        print("\n" + "="*60)
        print("DEMONSTRAÇÃO: POR QUE PERCEPTRON NÃO RESOLVE XOR?")
```

```
        print("="*60)

        X, y = DataGenerator.generate_logic_data('XOR', 2)

        print("\nTabela verdade XOR:")
        print("Entrada 1 | Entrada 2 | Saída")
        print("-" * 30)
        for i, (x1, x2) in enumerate(X):
            print(f"    {int(x1)}      |      {int(x2)}      |
{int(y[i])}")

        print("\nAnálise geométrica:")
        print("- Pontos (0,0) e (1,1) devem resultar em 0")
        print("- Pontos (0,1) e (1,0) devem resultar em 1")
        print("- Não existe uma linha reta que separe esses pontos!")

        # Tentar treinar Perceptron várias vezes
        print("\nTentativas de treinamento do Perceptron:")
        accuracies = []

        for i in range(5):
            perceptron = Perceptron(2, learning_rate=0.1,
use_bias=True,
                                  random_seed=i)
            train_result = perceptron.train(X, y, max_epochs=100,
verbose=False)
            predictions = perceptron.predict(X)
            accuracy = np.mean(predictions == y)
            accuracies.append(accuracy)
            print(f"Tentativa {i+1}: {accuracy:.1%}")

        print(f"\nAcurácia média: {np.mean(accuracies):.1%}")
        print("Conclusão: Perceptron falha consistentemente no XOR")

        # Comparar com MLP
        print("\nComparação com MLP:")
        mlp = MLP(input_size=2, hidden_size=3, learning_rate=0.1,
                 activation='sigmoid', use_bias=True)
        train_result = mlp.train(X, y, max_epochs=1000, verbose=False)
        predictions = mlp.predict(X)
        accuracy = np.mean(predictions == y)

        print(f"MLP: {accuracy:.1%} ({train_result['epochs']} épocas)")
```

```python
        print("Conclusão: MLP resolve XOR facilmente!")

def main():
    """Função principal para executar todos os experimentos"""
    print("🚀 INICIANDO EXPERIMENTOS COMPLETOS")
    print("Implementação: Perceptron vs Backpropagation")
    print("Autor: Sistema de IA")
    print("Data:", "2024")

    # Executar todos os experimentos
    runner = ExperimentRunner()

    # Experimentos com Perceptron
    perceptron_results = runner.run_perceptron_experiments()

    # Experimentos com MLP
    mlp_results = runner.run_mlp_experiments()

    # Demonstração da limitação do XOR
    runner.demonstrate_xor_limitation()

    print("\n" + "="*60)
    print("CONCLUSÕES FINAIS")
    print("="*60)

    print("\nPERCEPTRON:")
    print("- Resolve problemas linearmente separáveis (AND, OR)")
    print("- Falha em problemas não-lineares (XOR)")
    print("- Convergência rápida quando possível")
    print("- Algoritmo simples e eficiente")

    print("\nMLP (BACKPROPAGATION):")
    print("- Resolve qualquer problema (AND, OR, XOR)")
    print("- Camada escondida permite separação não-linear")
    print("- Sigmoid e Tanh funcionam bem para problemas pequenos")
    print("- ReLU é mais rápida mas pode ter problemas de 'morte'")
    print("- Taxa de aprendizado é crítica para convergência")
    print("- Bias é essencial para flexibilidade")

    print("\nRECOMENDAÇÕES:")
    print("- Use Perceptron para problemas simples e lineares")
    print("- Use MLP para problemas complexos e não-lineares")
    print("- Sigmoid/Tanh: bons para classificação binária")
```

```python
    print("- ReLU: melhor para redes maiores")
    print("- Taxa de aprendizado: comece com 0.1")
    print("- Sempre use bias para maior flexibilidade")

if __name__ == "__main__":
    main()
```