Universidade Federal do Rio de Janeiro
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

Programa de Engenharia de Sistemas e Computação

CPS769 - Introdução à Inteligência Artificial e Aprendizagem Generativa

Prof. Dr. Edmundo de Souza e Silva (PESC/COPPE/UFRJ)
Profa. Dra. Rosa M. Leão (PESC/COPPE/UFRJ)
Participação Especial: Gaspare Bruno (Diretor Inovação, ANLIX)

Lista de Exercícios 1b

Luiz Henrique Souza Caldas
email: lhscaldas@cos.ufrj.br

10 de julho de 2024

Questão 1

O objetivo deste trabalho é entender como um perceptron com duas entradas e uma entrada de bias classifica pontos em um espaço 2-D. Você usará duas funções de ativação diferentes: ReLU e Sigmoid.

1. Implemente um perceptron com duas entradas e uma entrada de bias.
2. Gere um conjunto de dados de pontos em um espaço 2D. Os pontos devem ser classificados em duas classes com base em suas coordenadas.
3. Treine o perceptron em um conjunto de dados de pontos em um espaço 2-D (escolha).
4. Use duas funções de ativação diferentes (Rectified Linear Unit (ReLU) e Sigmoid) para classificar os pontos.
5. Visualize os limites de decisão para ambas as funções de ativação.

Responda às seguintes perguntas com base no programa Python que você deverá fazer, e em suas observações:

1. Explique o processo de geração de dados no programa. Como os pontos são classificados em duas classes?

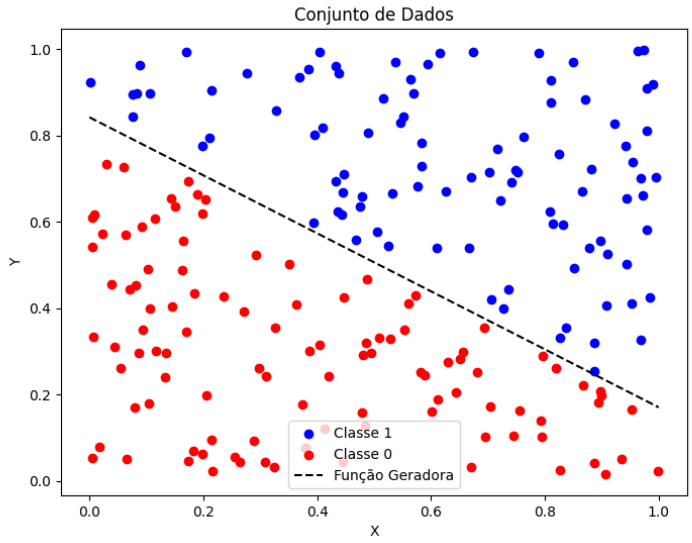
   Resposta:

   No programa, os dados de treinamento são gerados usando a classe DataGenerator (código no final deste relatório). Essa classe gera um conjunto de pontos aleatórios em um espaço 2D com coordenadas entre 0 e 1. Uma linha aleatória é gerada com uma inclinação (slope) e uma interceptação (intercept) aleatórias.

   Cada ponto é então classificado com base na posição relativa à linha. Especificamente, os pontos que estão acima da linha (onde a coordenada y é maior que a soma da inclinação vezes a coordenada x mais a interceptação) são classificados como pertencentes à classe 1, enquanto os pontos abaixo da linha são classificados como pertencentes à classe 0. Isso resulta em um conjunto de dados com pontos claramente divididos em duas classes.

   O *random seed* foi travado em 43 para garantir a repetibilidade do experimento (mesmo conjunto de dados a cada execução). O conjunto de dados utilizado pode ser visualizado na figura abaixo.

Figura 1: Conjunto de dados utilizado



2. Qual é o papel da função de ativação no perceptron? Compare as funções de ativação ReLU e Sigmoid.

   Resposta:

   A função de ativação no perceptron determina a saída do neurônio com base na soma ponderada de suas entradas. Ela introduz não-linearidade no modelo, permitindo que ele resolva problemas mais complexos.

   Comparação entre ReLU e Sigmoid:

   - Sigmoid: Retorna um valor entre 0 e 1, mapeando a soma ponderada de entradas para uma curva em forma de "S". É útil para problemas onde a saída precisa ser interpretada como uma probabilidade, mas pode sofrer com o desvanecimento do gradiente em redes profundas.
   - ReLU (Rectified Linear Unit): Retorna a entrada diretamente se for positiva; caso contrário, retorna zero. É computacionalmente eficiente e ajuda a resolver o problema do desvanecimento do gradiente, comum em redes profundas.

3. Treine o perceptron com funções de ativação ReLU e Sigmoid. Mostre os pesos finais para ambos os casos.

   Resposta:

   O treinamento do perceptron é feito utilizando o método train da classe Perceptron (código no final deste relatório). Durante o treinamento, para cada época, o algoritmo percorre todos os pontos de dados de treinamento, calcula a soma ponderada das entradas (incluindo um termo de bias), aplica a função de ativação (ReLU ou Sigmoid) para obter a previsão, e então calcula o erro como a diferença entre o rótulo real e a previsão. Utilizando a derivada

da função de ativação, os pesos são ajustados de acordo com a taxa de aprendizado para minimizar o erro. Este processo é repetido por um número especificado de épocas até que os pesos sejam suficientemente ajustados para classificar os pontos de dados corretamente.

Tabela 1: Pesos finais do perceptron para as funções de ativação ReLU e Sigmoid, exibidos com 5 casas decimais.

Função de Ativação	**Peso** w_0	**Peso** w_1	**Peso** w_2
ReLU	0.79833	1.49001	-0.75895
Sigmoid	4.52860	7.80608	-6.28154

Observação: Tempo de execução (ReLU): 3.84350 segundos. Tempo de execução (Sigmoid): 2.14966 segundos.

4. Trace os limites de decisão para ambas as funções de ativação. Descreva quaisquer diferenças que você observar.

   Resposta:

5. Como as funções de ativação ReLU e Sigmoid afetam a capacidade do perceptron de classificar os pontos?

   Resposta:

6. Como o número de iterações para a aprendizagem afeta o desempenho do perceptron e o limite de decisão?

   Resposta:

7. Quais são algumas limitações potenciais do uso de um perceptron de camada única para tarefas de classificação? Sugira possíveis melhorias.

   Resposta:

8. Seria possível fazer o treinamento das lista anterior apenas aumentando o número de neurônios de 1 para N? Explique de acordo com os artigos que você leu.

   Resposta:

Código

O código abaixo encontra-se no repositório https://github.com/lhscaldas/cps769-ai-gen, bem como o arquivo LaTex com o relatório e os códigos e arquivos LaTex das outras listas desta disciplina.

Código 1: código completo utilizado

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import time
4
```

```
class DataGenerator:
    def __init__(self, num_points=100, seed=None):
        self.num_points = num_points
        if seed is not None:
            np.random.seed(seed)
        self.data = np.random.rand(num_points, 2)
        self.labels = np.zeros(num_points)
        self.slope, self.intercept = self._generate_random_line()
        self._classify_points()

    def _generate_random_line(self):
        slope = np.random.uniform(-1, 1)
        intercept = np.random.uniform(0, 1)
        return slope, intercept

    def _classify_points(self):
        for i, (x, y) in enumerate(self.data):
            if y > self.slope * x + self.intercept:
                self.labels[i] = 1
            else:
                self.labels[i] = 0

    def get_data_and_labels(self):
        return self.data, self.labels

    def plot_data(self):
        plt.figure(figsize=(8, 6))
        plt.scatter(self.data[self.labels == 1][:, 0], self.data[self.labels ==
            1][:, 1], color='blue', label='Classe 1')
        plt.scatter(self.data[self.labels == 0][:, 0], self.data[self.labels ==
            0][:, 1], color='red', label='Classe 0')

        x_vals = np.array([0, 1])
        y_vals = self.slope * x_vals + self.intercept
        plt.plot(x_vals, y_vals, color='black', linestyle='--', label='Função
           Geradora')

        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Conjunto de Dados')
        plt.legend()
        plt.show()

class Perceptron:
    def __init__(self, input_size, learning_rate=0.01, activation_function='
       sigmoid'):
        if activation_function == 'relu':
            self.weights = np.random.randn(input_size + 1) * np.sqrt(2 /
               input_size)  # He initialization
        else:
            self.weights = np.random.randn(input_size + 1)  # Initialize
               weights with small random numbers
        self.learning_rate = learning_rate
        self.activation_function = activation_function
```

```
53
54      def activation(self, x):
55          if self.activation_function == 'relu':
56              return np.maximum(0, x)
57          elif self.activation_function == 'sigmoid':
58              return 1 / (1 + np.exp(-x))
59
60      def activation_derivative(self, x):
61          if self.activation_function == 'relu':
62              return np.where(x > 0, 1, 0)
63          elif self.activation_function == 'sigmoid':
64              sigmoid_x = self.activation(x)
65              return sigmoid_x * (1 - sigmoid_x)
66
67      def predict(self, inputs):
68          inputs_with_bias = np.append(inputs, 1)
69          weighted_sum = np.dot(self.weights, inputs_with_bias)
70          return self.activation(weighted_sum)
71
72      def train(self, training_data, labels, epochs=100):
73          for _ in range(epochs):
74              for inputs, label in zip(training_data, labels):
75                  inputs_with_bias = np.append(inputs, 1)
76                  weighted_sum = np.dot(self.weights, inputs_with_bias)
77                  prediction = self.activation(weighted_sum)
78                  error = label - prediction
79                  derivative = self.activation_derivative(weighted_sum)
80                  update = self.learning_rate * error * derivative *
                        inputs_with_bias
81                  self.weights += update
82          return self.weights
83
84  # Example usage
85  if __name__ == "__main__":
86      data_gen = DataGenerator(num_points=200, seed=43)
87      training_data, labels = data_gen.get_data_and_labels()
88
89      data_gen.plot_data()
90
91      epocas = 1000
92
93      # Training with ReLU
94      perceptron_relu = Perceptron(input_size=2, learning_rate=0.01,
           activation_function='relu')
95      start_time_relu = time.time()
96      weights_relu = perceptron_relu.train(training_data, labels, epochs=epocas)
97      end_time_relu = time.time()
98      time_elapsed_relu = end_time_relu - start_time_relu
99      print(f"Pesos finais (ReLU): {weights_relu}")
100     print(f"Tempo de execução (ReLU): {time_elapsed_relu:.5f} segundos")
101
102     # Training with Sigmoid
103     perceptron_sigmoid = Perceptron(input_size=2, learning_rate=0.01,
           activation_function='sigmoid')
```

```
    start_time_sigmoid = time.time()
    weights_sigmoid = perceptron_sigmoid.train(training_data, labels, epochs=
        epocas)
    end_time_sigmoid = time.time()
    time_elapsed_sigmoid = end_time_sigmoid - start_time_sigmoid
    print(f"Pesos finais (Sigmoid): {weights_sigmoid}")
    print(f"Tempo de execução (Sigmoid): {time_elapsed_sigmoid:.5f} segundos")
```