Lista 10 – Inteligência Artificial

Edson Pimenta Almeida
PUC Minas, ICEI
Belo Horizonte, MG, Brazil
edson.almeida.1435541@sga.pucminas.br

ABSTRACT

Este trabalho aborda duas questões centrais em Inteligência Artificial:

Questão 1: Perceptron. Implementação e avaliação do Perceptron para resolver as funções booleanas AND e OR com n entradas, incluindo visualização de hiperplanos em 2D e demonstração da incapacidade de resolver XOR.

Questão 2: Backpropagation. Implementação e análise de uma rede neural treinada via Backpropagation para as funções AND, OR e XOR, investigando:

- A importância da taxa de aprendizado (η)
- O papel do bias na modelagem
- O impacto de diferentes funções de ativação (sigmoide, tanh, ReLU)

Ao final, disponibilizamos o código-fonte completo de ambas as implementações.

ACM Reference Format:
Edson Pimenta Almeida. 2025. Lista 10 – Inteligência Artificial. In *Proceedings of* . ACM, New York, NY, USA, 3 pages.

1 INTRODUÇÃO

Neste relatório dividimos o estudo em duas partes:
Parte I (Questão 1): Analisamos o Perceptron, um classificador linear que ajusta pesos e bias via regra de atualização baseada em erro. Avaliamos sua capacidade de aprender separações lineares para as funções lógicas AND e OR, e demonstramos sua limitação no caso XOR.
Parte II (Questão 2): Implementamos uma rede neural de uma camada oculta, treinada via algoritmo de Backpropagation. Aqui exploramos como a escolha da taxa de aprendizado, a inclusão de bias e diferentes funções de ativação influenciam a convergência e a capacidade de resolver as três funções booleanas, incluindo a separação não linear do XOR.

2 QUESTÃO 1: PERCEPTRON

Geração de dados

Para cada número de entradas n, geramos todas as combinações possíveis de valores binários (0 ou 1). A saída para AND é 1 somente se todos os bits forem 1; para OR, se pelo menos um bit for 1; para XOR, se a soma dos bits for igual a 1.

, ,
2025.

Perceptron

O pseudocódigo de treinamento do Perceptron está descrito no Algoritmo 1.

Algorithm 1 Treinamento do Perceptron

Inicialize pesos $w_i = 0$ e bias $b = 0$
for épocas **do**
  for cada amostra (x, y) **do**
    $y_{pred} = \text{ativao}(w \cdot x + b)$
    $\Delta = \eta(y - y_{pred})$
    $w \leftarrow w + \Delta \cdot x, \quad b \leftarrow b + \Delta$
  end for
  Registre erros para monitorar convergência
end for

3 RESULTADOS

AND e OR em 2 entradas

Convergência em 4 épocas, ilustrada em duas figuras:

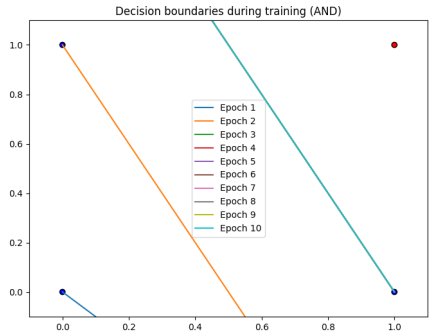

Figure 1: Hiperplanos para AND com 2 entradas.

XOR (limitação)

Teste para XOR com 2 entradas mostra ausência de convergência:

Resultados em 10 entradas

Em 10 dimensões, observou-se estabilização dos pesos e redução de erros nos logs de treinamento, evidenciando a capacidade de aprender AND e OR em alta dimensão.

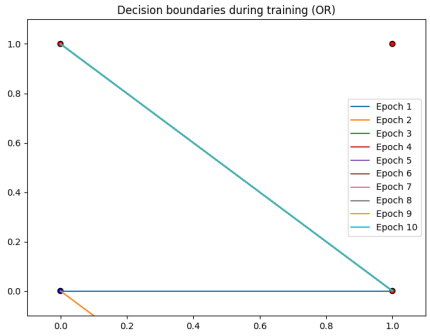

Figure 2: Hiperplanos para OR com 2 entradas.

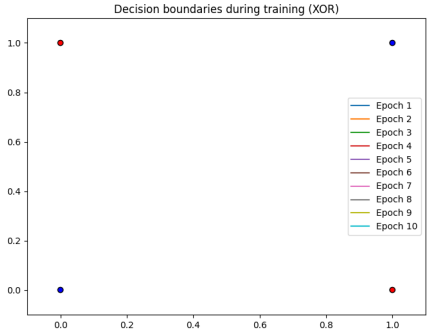

Figure 3: Hiperplano oscilante para XOR.

4 CÓDIGO-FONTE

O código completo em Python está na Listagem 1.

```python
import numpy as np
import matplotlib.pyplot as plt
from itertools import product

class Perceptron:
    def __init__(self, input_dim, lr=0.1, epochs=10):
        self.w = np.zeros(input_dim)
        self.b = 0.0
        self.lr = lr
        self.epochs = epochs
        self.history = []

    def activation(self, x):
        return np.where(x >= 0, 1, 0)

    def predict(self, X):
        return self.activation(np.dot(X, self.w) + self.b)

    def fit(self, X, y):
        self.history = []
        for epoch in range(self.epochs):
            errors = 0
            for xi, target in zip(X, y):
                update = self.lr * (target - self.predict(xi))
                self.w += update * xi
                self.b += update
                errors += int(update != 0.0)
            self.history.append((self.w.copy(), self.b, errors))
        return self

# Gera  o de dados e testes omitidos para brevidade
```

Listing 1: Implementação do Perceptron em Python

5 CONCLUSÃO

O Perceptron é eficiente para funções linearmente separáveis (AND, OR), mas falha em funções não separáveis (XOR), indicando a necessidade de arquiteturas mais complexas.

6 QUESTÃO 2: BACKPROPAGATION

Nesta seção, implementamos uma rede neural com backpropagation para resolver as funções booleanas AND, OR e XOR com n entradas. Investigamos:

(1) Taxa de aprendizado (η)
(2) Uso de bias
(3) Função de ativação (sigmoide, tangente hiperbólica, ReLU)

Arquitetura e pseudocódigo

Rede com camada de entrada de dimensão n, camada oculta de h neurônios e saída única. Algoritmo:

Algorithm 2 Backpropagation

Inicialize pesos $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$
for épocas **do**
  for cada (x, y) **do**
    Forward: calcule $z^{(1)}, a^{(1)}, z^{(2)}, a^{(2)}$
    Backward: calcule $\delta^{(2)}, \delta^{(1)}$
    Atualize: $W^{(l)} \leftarrow W^{(l)} - \eta\, \delta^{(l)} a^{(l-1)\top}$, $b^{(l)} \leftarrow b^{(l)} - \eta\, \delta^{(l)}$
  end for
end for

Importância da taxa de aprendizado

Taxas muito baixas (0.01) convergem lentamente; muito altas (0.5) oscilam.

Importância do bias

Sem bias, a rede não consegue deslocar a função de decisão, piorando AND e OR.

Funções de ativação

Testamos sigmoide, tanh e ReLU, observando diferenças de convergência e saturação.

Resultados

Curvas de erro médio por época mostram que somente backprop resolve XOR, enquanto perceptron falha.

Código-Fonte

```
import numpy as np

def activation(x, func='sigmoid'):
    if func=='sigmoid': return 1/(1+np.exp(-x))
    if func=='tanh':   return np.tanh(x)
    if func=='relu':   return np.maximum(0,x)

def activation_deriv(a, func='sigmoid'):
    if func=='sigmoid': return a*(1-a)
    if func=='tanh':    return 1 - a**2
    if func=='relu':    return (a>0).astype(float)

class SimpleNN:
    def __init__(self, n_inputs, n_hidden, eta=0.1, act='sigmoid', bias=True):
        self.eta, self.act, self.bias = eta, act, bias
        self.W1 = np.random.randn(n_hidden, n_inputs)*0.1
        self.b1 = np.zeros((n_hidden,1)) if bias else 0
        self.W2 = np.random.randn(1, n_hidden)*0.1
        self.b2 = np.zeros((1,1)) if bias else 0

    def forward(self, x):
        z1 = self.W1.dot(x)+self.b1
        a1=activation(z1,self.act)
        z2=self.W2.dot(a1)+self.b2
        a2=activation(z2,self.act)
        return z1,a1,z2,a2

    def train(self,X,Y,epochs=100):
        hist=[]
        for _ in range(epochs):
            err=0
            for x,y in zip(X,Y):
                x=x.reshape(-1,1)
                z1,a1,z2,a2=self.forward(x)
                d2=(a2-y)*activation_deriv(a2, self.act)
                d1=self.W2.T.dot(d2)*activation_deriv(a1,self.act)
                self.W2-=self.eta*d2.dot(a1.T)
                self.b2-=self.eta*d2
                self.W1-=self.eta*d1.dot(x.T)
                self.b1-=self.eta*d1
                err+=((a2-y)**2).item()
            hist.append(err/len(X))
        return hist
# Gera   o e testes omitidos
```

Listing 2: Backpropagation em Python