

Relatório do Laboratório 7:

Redes Neurais

Isabelle Ferreira de Oliveira
CT-213 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

Resumo—Esse relatório documenta a implementação de uma rede neural de duas camadas para realizar a segmentação de cores para o futebol de robôs. Para isso, foi necessário configurar essa rede neural para realizar classificação multi-classe, implementando os algoritmos de Forward Propagation (inferência) e Back Propagation (treinamento) para essa rede.

Index Terms—Redes neurais, segmentação de cores, Forward Propagation, Back Propagation

I. INTRODUÇÃO

Redes neurais são sistemas de computação que podem reconhecer padrões escondidos, agrupar dados e classificá-los, além de, com o tempo, aprender e melhorar continuamente [3].

Uma rede neural é formada por camadas de neurônios artificiais (baseado no sistema nervoso humano e neurônios naturais), e eles são os responsáveis por realizar as contas que levam de uma entrada a uma saída esperada (para o caso de aprendizado supervisionado). A Figura 1 apresenta um exemplo de rede neural com duas camadas (a camada de entrada geralmente não é considerada nesse tipo de contagem). É possível notar no exemplo que a camada 1 tem 3 neurônios e a camada 2 tem 1 neurônio.

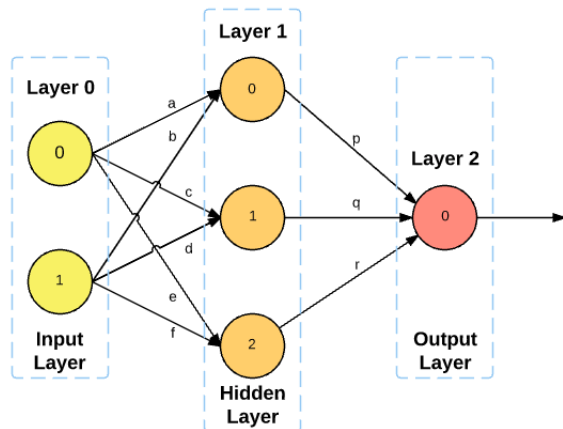


Figura 1. Exemplo de rede neural, com duas camadas (1 camada de entrada, 1 camada escondida e 1 camada de saída), como a trabalhada nesse laboratório. Essa imagem de exemplo foi apresentada no site [2]

Agora analisando um neurônio em particular, como o apresentado na Figura 2, a conta realizada para o aprendizado é que

se encontra no interior do círculo, ou seja: $g(\sum w_j x_j + b) = y$, no qual w são os pesos; b , os bias; x , as entradas; e g , a função de ativação. O objetivo do treinamento é ajustar w e b para aproximar alguma função $y(x)$ de acordo com os valores fornecidos como *outputs* esperados.

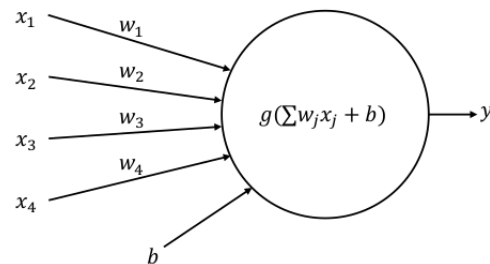


Figura 2. Exemplo de neurônio. Essa imagem de exemplo foi apresentada no roteiro [1]

O processo de inferência é realizado por meio do cálculo da equação apresentada no círculo do neurônio da Figura 2, em um algoritmo chamado *Forward Propagation*. Já o processo de aprendizado é realizado pelo algoritmo de *Back Propagation*, que calcula os pesos e *biases* a partir das saídas esperadas e das decidas de gradientes calculados para esses pesos e *bias*. Já esses gradientes são calculados por meio das saídas esperadas, as entradas iniciais e os resultados intermediários calculados por meio do *Forward Propagation*.

Os pseudo-códigos geral dos algoritmos *Forward Propagation*, *Back Propagation* e o cálculo das decidas de gradientes para pesos e *biases* podem ser visto a seguir. Em seguida, será apresentado como esses algoritmos foram implementados no contexto do laboratório.

```
# Forward Propagation
def forward_propagation(input):
    z[1] = weights[1]input + biases[1]
    a[1] = g[1](z[1])
    z[2] = weights[2]a[1] + biases[2]
    a[2] = g[2](z[2])
    return z, a
```

```
# Gradients' Computation
```

```

def compute_gradient_back_propagation(inputs,
    expected_outputs):
    for i in range(len(inputs)):
        input = inputs[i]
        expected_output = expected_outputs[i]

        z, a = forward_propagation(input)

        dz[2] = a[2] - expected_output
        weights_gradient[2] += dz[2] * a[1].T
        biases_gradient[2] += dz[2]

        dz[1] = weights[2].T * dz[2] *
            derivate(g[1](z[1]))
        weights_gradient[1] += dz[1] * input.T
        biases_gradient[1] += dz[1]

    return weights_gradient, biases_gradient

```

```

# Back Propagation
def back_propagation(inputs,
    expected_outputs):
    weights_gradient, biases_gradient =
        compute_gradient_back_propagation(inputs,
            expected_outputs)

    weights[1] -= alpha * weights_gradient[1]
    biases[1] -= alpha * biases_gradient[1]
    weights[2] -= alpha * weights_gradient[2]
    biases[2] -= alpha * biases_gradient[2]

```

No pseudocódigo acima, J é a função para medir a qualidade das soluções candidatas; $m0$ e $C0$ são a média e a matriz de covariância iniciais da população que será gerada, respectivamente; m e C são, de forma análoga, a média e a matriz de covariância atuais da população que será gerada, respectivamente; e μ é o tamanho da população considerada como "melhores soluções até então".

II. IMPLEMENTAÇÃO DO ALGORITMO

Para a implementação da rede neural, era necessário preencher a função *forward_propagation()*, *compute_gradient_back_propagation()* e *back_propagation()* da classe *NeuralNetwork*. Essas funções a se completar estavam no código base fornecido [1].

Recebendo os valores de *input*, a função *forward_propagation()* foi implementada conforme apresentado no pseudo-código da Introdução, com o detalhe que a função de ativação g de todos os neurônios utilizada foi a *sigmoid*. Além disso, tanto a função *sigmoid* quanto sua derivada já eram fornecidos pelo código base.

Já na implementação de *back_propagation()*, também foi implementado conforme o sugerido no pseudo-código da Introdução, ou seja, atualizando os valores de pesos e *biases* subtraindo de seu valor um fator de aprendizado α vezes os gradientes de pesos e *biases*, respectivamente. Esses valores de gradientes são aqueles calculados em *compute_gradient_back_propagation()*, supondo que essa função já foi corretamente implementado. Foi utilizado para α o mesmo que o sugerido pelo código base.

Finalmente, para implementar a função *compute_gradient_back_propagation()*, também foi feito de forma bastante semelhante ao apresentado na Introdução, com o detalhe de usar *numpy.multiply* para multiplicar termo a termo os vetores referentes aos pesos e a derivada da função de ativação.

Para testar o funcionamento dessas implementações, foi inicialmente alterado o valor da variável *classification_function* (entre "sum_gt_zero" no arquivo *test_neural_network.py* do código base, gerando imagens dos resultados da classificação de cada *dataset* para cada uma dessas funções (função "soma > 0" e função "xor") usando rede neural.

Já tendo ciência da correta implementação dos algoritmos de aprendizado, foi feito por fim um teste da segmentação de cores, executando o código do arquivo *test_color_segmentation.py*, conforme indicado no roteiro [1]. Os gráficos de resultados tanto dessa segmentação de cores, quanto das funções anteriores, foram apresentados nas Figuras 3 a 11.

III. RESULTADOS E CONCLUSÕES

A. Teste das Funções soma > 0 e xor

O aprendizado com a rede neural foi executado para as duas funções já citadas na seção anterior. Os resultados dessas execuções foram satisfatórios e saíram conforme o esperado, comprovando o correto funcionamento da implementação e a validade da utilização de rede neural com aprendizado supervisionado no aprendizado dessas funções. Esses resultados foram apresentados nas Figuras de 3 a 8.

Vale reparar que, nas Figuras 3 e 6, os custos não são sempre decrescentes com o passar das iterações. Isso, entretanto, não configura erro, uma vez que os ruídos nos *dataset* acabam interferindo durante o aprendizado. O resultado continua correto uma vez que a tendência geral é a diminuição desse custo, até a convergência. O mesmo vale, inclusive, para o gráfico da Figura 9, que será melhor discutido na subseção seguinte.

A comparação entre as Figuras 4 e 5 e entre as Figuras 7 e 8 demonstra que a classificação do *dataset* aconteceu de maneira satisfatória.

B. Teste da segmentação de cores

O custo na Figura 9 segue a tendência geral de diminuição e convergência com o passar das iterações, apesar de alguns picos já explicados na subseção anterior. Esse resultado é satisfatório e pode ser melhor observado na comparação entre as Figuras 10 e 11, que demonstra que a rede neural realmente conseguiu aprender a classificar as cores verdes e branco com considerável acerto. Vale notar também as cores não identificadas, como a bola laranja, que foi deixada como preta (cor representativa de impossibilidade de identificar).

Tendo em vista o que foi apresentado, pode-se notar, por fim, que esses algoritmos realmente se demonstraram eficazes em encontrar os pesos e *biases* dessa rede neural, além de demonstrar a capacidade de aprendizado de uma rede neural para esses problemas de classificação.

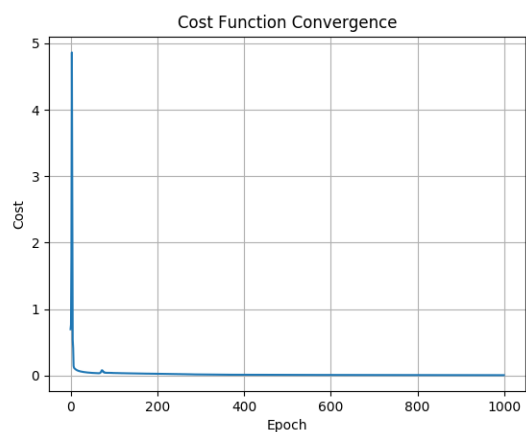


Figura 3. Otimização da função de Esfera Transladada usando a estratégia evolutiva SES. O resultado encontrado é o ponto vermelho.

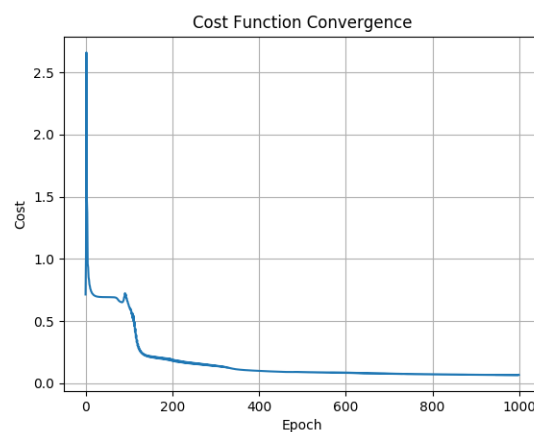


Figura 6. Otimização da função de Esfera Transladada usando a estratégia evolutiva SES. O resultado encontrado é o ponto vermelho.

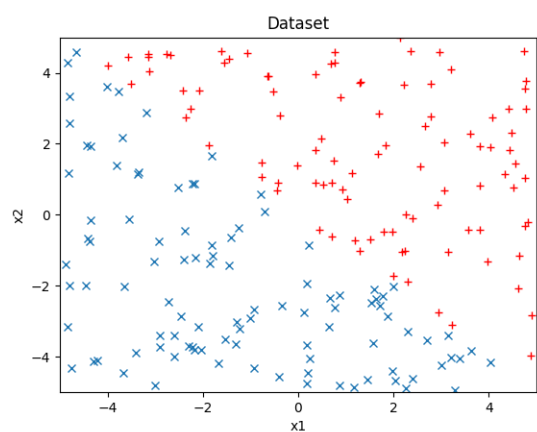


Figura 4. Otimização da função de Esfera Transladada usando a estratégia evolutiva CMA-ES. O resultado encontrado é o ponto vermelho.

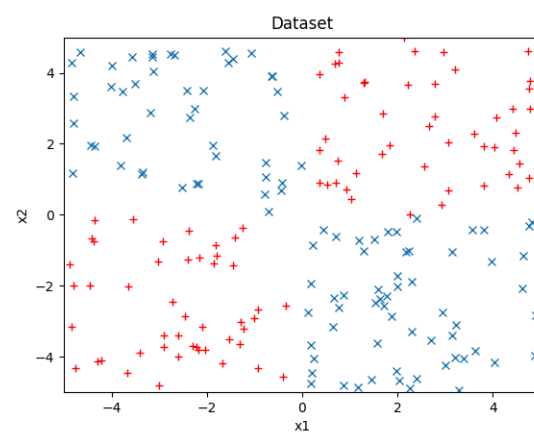


Figura 7. Otimização da função de Esfera Transladada usando a estratégia evolutiva CMA-ES. O resultado encontrado é o ponto vermelho.

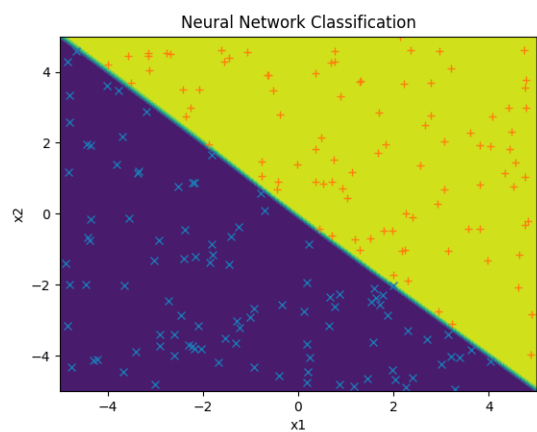


Figura 5. Otimização da função de Ackley usando a estratégia evolutiva SES. O resultado encontrado é o ponto vermelho.

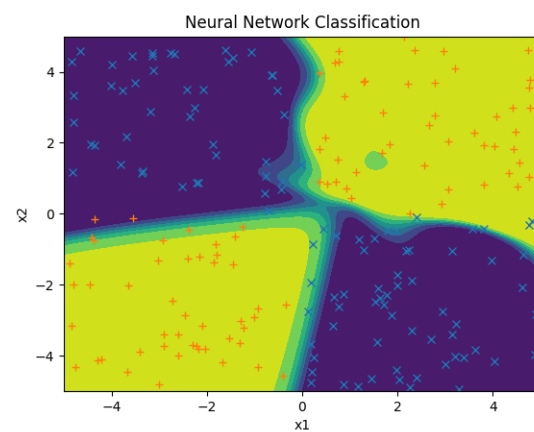


Figura 8. Otimização da função de Ackley usando a estratégia evolutiva SES. O resultado encontrado é o ponto vermelho.

REFERÊNCIAS

- [1] M. Maximo, "Roteiro: Laboratório 5 - Estratégias Evolutivas". Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.
- [2] Towards Data Science, "Neural Net from scratch". Acessado em <https://towardsdatascience.com/neural-net-from-scratch-using-numpy-71a31f6e3675>.
- [3] SAS, "Redes Neurais: O que são e qual a sua importância?". Acessado em https://www.sas.com/pt_br/insights/analytics/neural-networks.html.

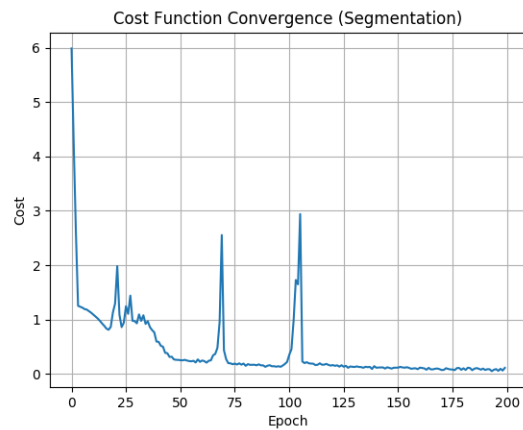


Figura 9. Otimização da função de Esfera Transladada usando a estratégia evolutiva SES. O resultado encontrado é o ponto vermelho.

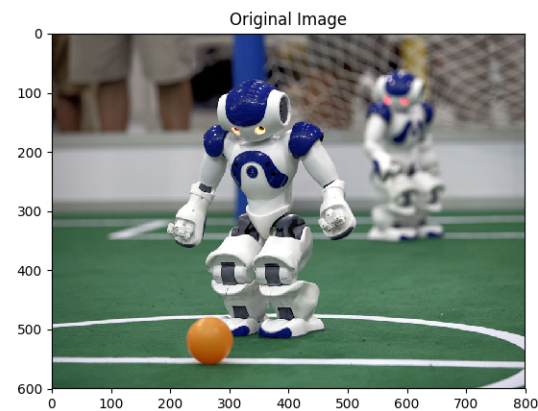


Figura 10. Otimização da função de Esfera Transladada usando a estratégia evolutiva CMA-ES. O resultado encontrado é o ponto vermelho.

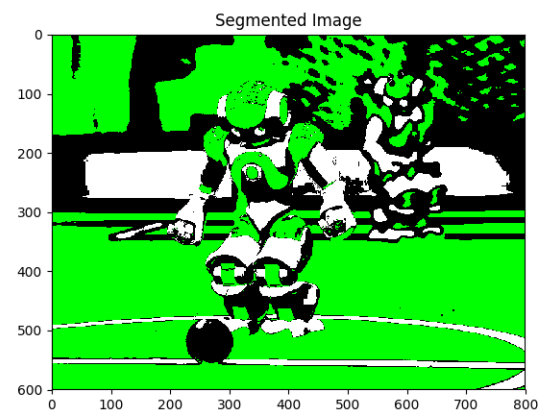


Figura 11. Otimização da função de Ackley usando a estratégia evolutiva SES. O resultado encontrado é o ponto vermelho.