Instituto Tecnológico de Aeronáutica

CT-213

Laboratório 6 - Aprendizado de Máquina

Aluno: Pedro Elardenberg Sousa e Souza
Professor: Marcos Ricardo Omena de Albuquerque Maximo

3 de dezembro de 2023

Conteúdo

1 Resumo

Neste Laboratório, foi treinado um algoritmo de visão computacional em *python*, utilizando aprendizado de máquina, para identificação de um campo de futebol de robôs. O algoritmo recebe uma imagem de um jogo de futebol de robôs e a segmenta em três cores: verde, branco e preto, as duas primeiras as cores mais abundantes nesse tipo de situação.

Para isso, foi feita uma rede neural em três camadas, sendo uma camada de entrada, com valores iniciais do problema, uma camada intermediária e uma camada de saída, com os resultados do problema. Foi utilizada uma classificação multi-classe para calcular a função de custo.

Na implementação, foram criados os métodos *Forward Propagation* para calcular as entradas e as ativações de cada neurônio e *Back Propagation* para calcular os pesos de cada neurônio na camada posterior e os viéses associados. As funções, então, foram testadas utilizando o código *test_neural_network.py*, a qual recebe um conjunto numérico e os conforma ao resultado de uma função predeterminada.

Uma vez implementados, os métodos foram utilizados pelo código *test_color_segmentation.py* para segmentar uma imagem correspondente a um momento de um jogo de futebol de robôs nas cores anteriormente descritas.

Os resultados foram então mostrados e discutidos neste relatório.

2 Indrodução

Em aprendizado de máquina, uma rede neural pode ser entendida como um conjunto de neurônios artificiais interligados de modo a tornar um conjunto inicial de valores, usualmente dados brutos, em um conjunto de valores tratados para a utilidade que nós queremos. Um neurônio artificial pode ser entendido como uma entidade que recebe os valores de uma camada anterior de neurônios e realiza uma operação matemática que gera um determinado resultado. A figura 1 representa o funcionamento de um neurônio artificial.

Uma camada de neurônios é um subconjunto em que cada elemento recebe valores dos neurônios da camada anterior e envia o resultado de sua operação para neurônios na camada seguinte. A figura 2 ilustra a ideia de camadas de uma rede neural.

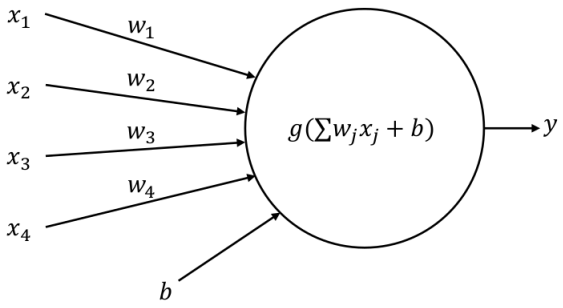

Figura 1: Representação de um neurônio artificial em uma rede neural[2]

Em que a conta realizada pelo neurônio é descrita por:

$$y = g\left(\sum w_j x_j + b\right) = g(\mathbf{w}^T\mathbf{x} + b) = g(z) \tag{1}$$

w: pesos
b: bias
x: entradas (features)
g: função de ativação

Para este problema, foi usada a função de ativação sigmóide $\sigma(x)$.

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{2}$$

Da equação 1, temos os hiperparâmetros **w** e b, os quais devem ser ajustados para cada neurônio de modo a gerar o resultado esperado ao fim da

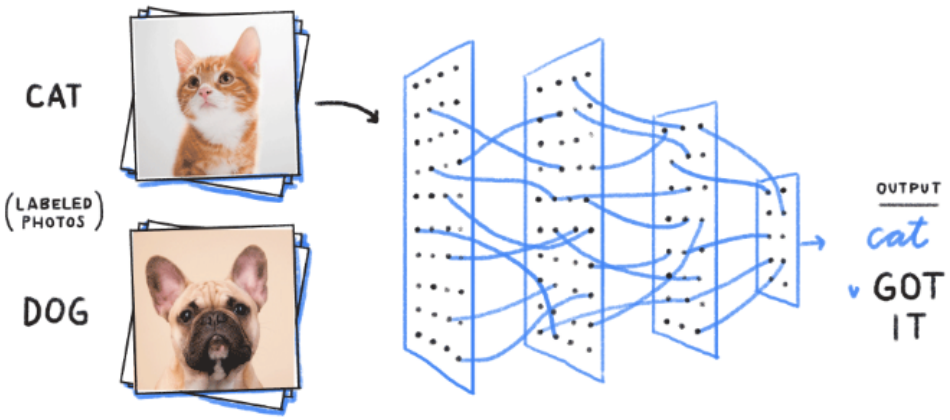

Figura 2: Ilustração do funcionamento de uma rede neural para diferenciação de uma imagem entre um cão e um gato[1]

rede neural. Para tanto, utiliza-se otimização. Como a equação 2 é derivável, pode-se usar descida de gradiente como método de otimização.

$$\theta_{n+1} = \theta_n + \alpha \frac{\partial J(\theta_n)}{\partial \theta_n} \tag{3}$$

α : taxa de aprendizagem
$\theta = [\mathbf{w}\ b]^T$: vetor com os pesos da rede
J: função de custo para a descida do gradiente

Em que a função de custo é a função de entropia binária (ou *cross entropy*) calculada como

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} -[y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)})] \tag{4}$$

$y^{(i)}$: resultado do i-ésimo exemplo de treinamento. Vetor de valores necessários para otimizar a função
$\hat{y}^{(i)}$: valor da saída do neurônio g(z), o qual é esperado que resulte um valor próximo de $y^{(i)}$

Iterando sobre a equação 3 n vezes, pode-se enfim ter os hiperparâmetros da equação 1 otimizados para o problema posto.

3 Análise dos Resultados

Este laboratório possui dois casos de treino, *test_neural_network.py* e *test_color_segmentation.py* e um código de implementação da rede neural, *neural_network.py*.

O primeiro caso recebe um *array* de duplas de números $[x_1\ x_2]$ e retorna um valor 0 ou 1, a depender da função utilizada para o treinamento da rede neural. O objetivo é usar esse caso mais simples para ajustar as operações feitas na rede neural, de modo a realizar corretamente o treinamento do segundo caso.

3.1 Funcionamento da rede neural

A rede neural de *neural_network.py* é uma rede com uma única camada escondida. E recebe a quantidade de *inputs*, a quantidade de neurônios da camada 1, a quantidade de outputs que ela deve gerar e a taxa de aprendizado α. Gera-se valores aleatórios iniciais para os pesos $\boldsymbol{w}$ e valores nulos para os *biases* b da equação 1. A rede possui quatro métodos:

3.1.1 *forward_propagation(inputs)*

Esta função recebe os *inputs* e devolve os valores de entrada e saída de cada neurônio, seguindo o modelo da equação 1, respectivamente, z e $g(z)$, aqui chamado de função de ativação a, dada pela equação 2. Assim, a rede é dada por:

$$\begin{aligned} a[0] &= z[0] = inputs \\ z[1] &= w[1]a[0] + b[1] \\ a[1] &= \sigma(z[1]) \end{aligned} \tag{5}$$

3.1.2 *back_propagation(inputs, expected_outputs)*

Essa função recebe os *inputs* e os *outputs* esperados e, após ter os valores dos gradientes dos pesos e *biases* da função *compute_gradient_back_propagation*, recalcula esses pesos e *biases*, seguindo a equação 3.

3.1.3 *compute_gradient_back_propagation(inputs, expected_outputs)*

Essa função calcula as derivadas parciais da equação 3 da seguinte forma:

$$\begin{aligned}
&\frac{\partial L}{\partial w[2]} = \delta_2 \times a[1] \qquad && \frac{\partial L}{\partial w[1]} = \delta_1 \times a[0] \\
&\frac{\partial L}{\partial b[2]} = \delta_2 && \frac{\partial L}{\partial b[1]} = \delta_1 \\
&\delta_2 = (a[2] - y) && \delta_1 = w[2] \times \delta_2 \times \sigma'(z[1])
\end{aligned} \tag{6}$$

Em que L é a função dentro do somatório da equação 4 e y são os *outputs* esperados do problema.

3.1.4 compute_cost(inputs, expected_outputs)

Essa função recebe os valores de a e z de *forward_propagation* e realiza o cálculo do custo descrito na equação 4.

3.2 Rede neural de teste *neural_network.py*

Esse código testa a implementação da rede neural com algumas funções de classificação simples (com uma classe):

- sum_gt_zero(): função que classifica se a soma das duas entradas é maior que zero.
- xor(): função inspirada na operação de ou exclusivo que classifica se as duas entradas tem o mesmo sinal.

Os inputs são 200 duplas $[x_1\ x_2]$ com valores aleatórios entre -5 e 5. A rede neural, portanto, tem 2 neurônios iniciais (*inputs*). Possui também 10 neurônios na camada escondida e 1 *output*, que é o resultado 0 ou 1 dado por uma das funções acima. A taxa de aprendizado α é igual a 6 e o número de iterações de treinamento, ou seja, quantas vezes o programa chama a função *back_propagation*, é igual a 1000.

Os resultados, para as duas funções de classificação, são dados pelas imagens abaixo:

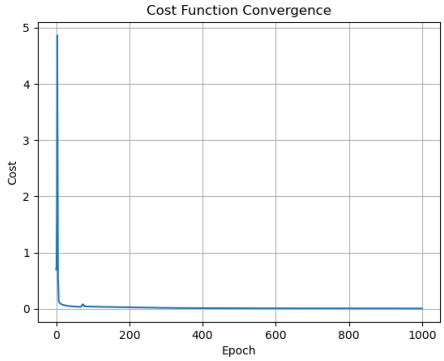

Figura 3: Convergência da rede neural para a função sum_gt_zero()

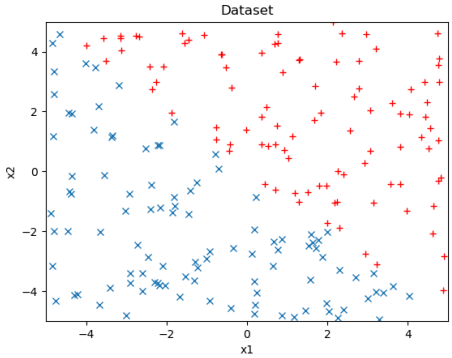

Figura 4: *Dataset* da função sum_gt_zero()

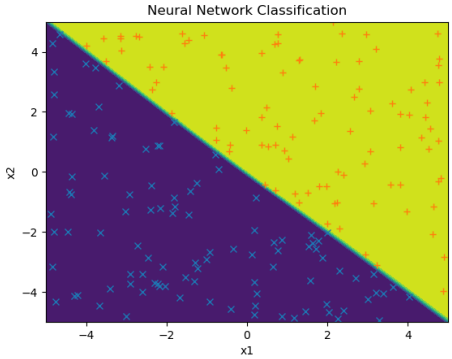

Figura 5: Resultado da classificação da rede neural para a função sum_gt_zero()

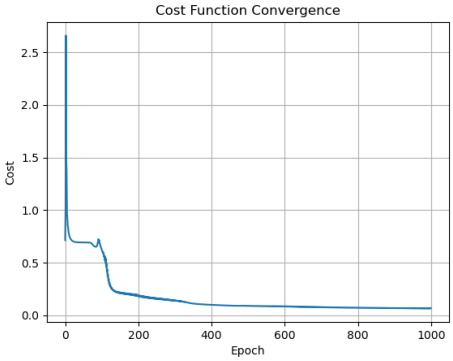

Figura 6: Convergência da rede neural para a função xor()

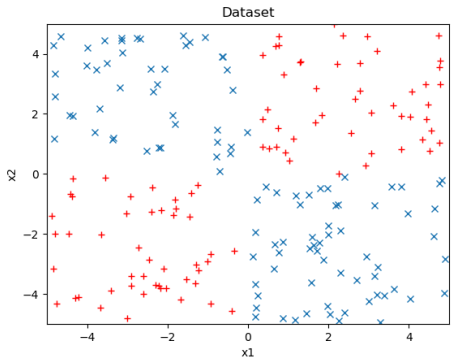

Figura 7: *Dataset* da função xor()

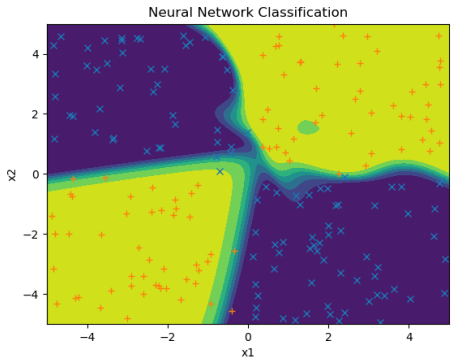

Figura 8: Resultado da classificação da rede neural para a função xor()

Vê-se, ao comparar as imagens 4 com 5 e 7 com 8, que os resultados foram obtidos satisfatoriamente para as duas funções. A função de classificação sum_gt_zero(), por ser mais simples, convergiu mais rapidamente. Observando a imagem 3, nota-se que os parâmetros estavam otimizados antes da centésima iteração. Para a função xor(), o resultado convergiu em torno da iteração 300, como se vê na imagem 6.

3.3 Rede neural de segmentação de cores *test_color_segmentation.py*

Esse código realiza o aprendizado da segmentação de cores e exibe o resultado. Os inputs são os pixels de uma imagem do futebol de robôs (figura 10) e o programa segmenta as cores em branco, verde e preto, que são as principais cores dessa categoria. Foram utilizadas 600 iterações para o treinamento da rede, como mostra a figura 9.

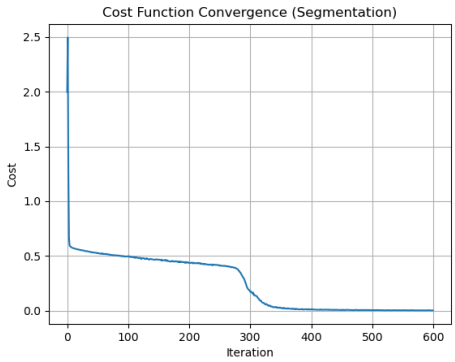

Figura 9: Convergência da rede neural para a segmentação de cores

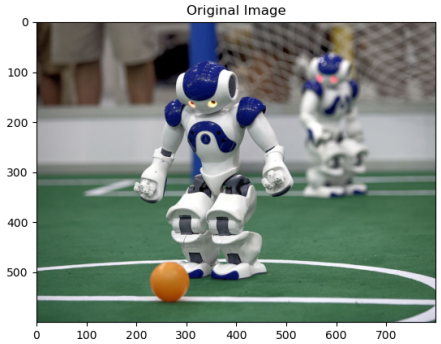

Figura 10: Imagem modelo para a segmentação de cores

O resultado da segmentação portanto, é mostrado na figura 11.

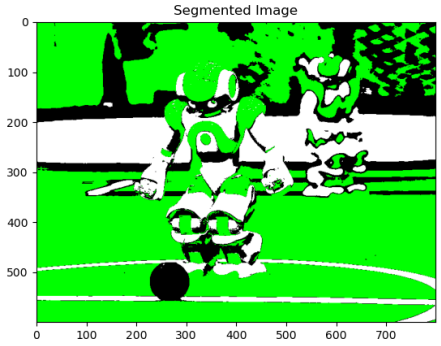

Figura 11: Imagem segmentada pelo algoritmo em *test_color_segmentation.py*

4 Conclusão

Com este laboratório, foi possível exemplificar a eficácia e o poder dos algoritmos de rede neural utilizando *back propagation* como uma forma de otimização de parâmetros.

Observação: um dos grandes desafios da implementação foi fazer a correta operação matricial, uma vez que os *arrays* de entrada no neurônio z, função de ativação a, pesos w, *biases* b e seus respectivos gradientes possuem dimensões que podem variar entre eles e entre si próprios em cada espaço do vetor (o *array* $z[0]$ possui dimensões diferentes de $z[1]$, por exemplo). Para isso, as dimensões dos vetores para os casos de *test_neural_network.py* foram

adicionadas como comentário no código de *neural_network.py* para maior compreensão dos cálculos.

Referências

[1] Simple image classification using convolutional neural network — deep learning in python, 2017. Acessado em 26 de Novembro de 2023. URL: `https://becominghuman.ai/building-an-image-classifier-using-deep-learning-in-python-totally-from-a-beginners-perspective-be8dbaf22dd8`.

[2] Marcos Ricardo Omena de Albuquerque Maximo. Ct-213 - aula 7 - aprendizado de máquina. Apostila, 2020. Curso CT-213 - Inteligência Artificial para Robótica Móvel, Instituto Tecnológico de Aeronáutica.