Instituto Tecnológico de Aeronáutica – ITA
Inteligência Artificial para Robótica Móvel – CT-213
Professor: Marcos Ricardo Omena de Albuquerque Maximo

Laboratório 6 – Redes Neurais

Observação: antes de começar a fazer esse laboratório, é necessário ter o `pillow` no seu ambiente do Anaconda. Se ainda não o tem, basta fazer como já fez anteriormente para outras dependências.

1. Introdução

Nesse laboratório, seu objetivo é implementar uma rede neural de 3 camadas (1 camada de entrada, 1 camada escondida e 1 camada de saída) para realizar segmentação de cores para o futebol de robôs. A Figura 1 ilustra esse processo, em que na Figura 1(a) tem-se a imagem original e na 1(b) a imagem após a segmentação.

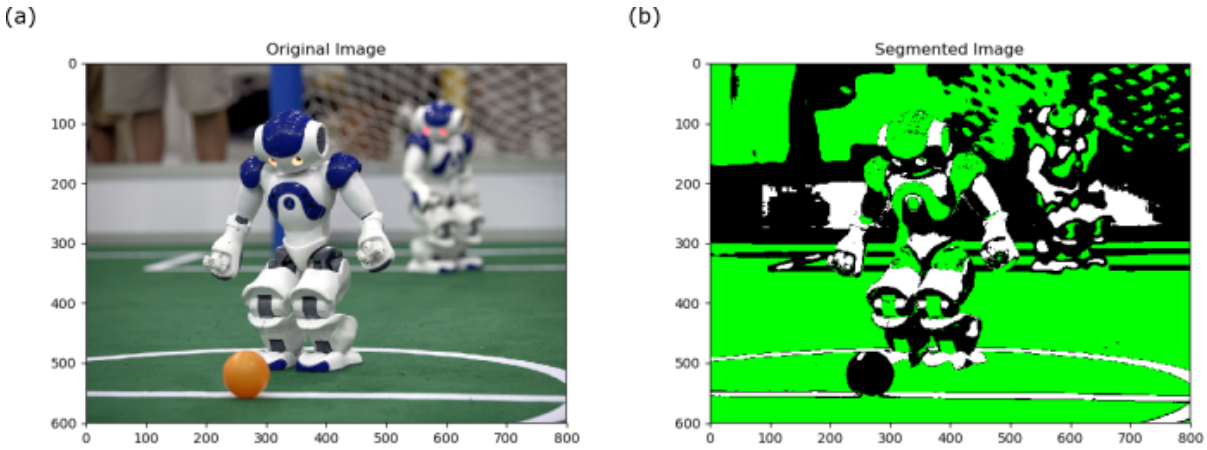

Figura 1: segmentação de cores.

2. Descrição do Problema

O problema a ser resolvido é a implementação de uma rede neural de três camadas para realizar a segmentação de cores para o futebol de robôs. Para isso, será necessário configurar essa rede neural para realizar classificação multi-classe. Você deve implementar os algoritmos de Forward Propagation (inferência) quanto Back Propagation (treinamento) para essa rede. Como trata-se de um problema de classificação de multi-classe, deve-se usar uma *loss function* de regressão logística multi-classe:

$$L(y^{(i)}, \hat{y}^{(i)}) = -\sum_{c=1}^{C} [(1 - y_c^{(i)}) log(1 - \hat{y}_c^{(i)}) + y_c^{(i)} log(\hat{y}_c^{(i)})]$$

Como de costume, a função de custo é dada pela média dos *losses* dos exemplos de treinamento:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)})$$

Considerando uma rede neural de 3 camadas (1 camada escondida) e com função de ativação sigmóide $\sigma(.)$ em todos os neurônios, chega-se às seguintes equações para o algoritmo de Back Propagation nessa rede:

$$\frac{\partial L}{\partial w^{[2]}_{ck}} = \delta^{[2]}_{c} a^{[1]}_{k}$$

$$\frac{\partial L}{\partial b^{[2]}_{c}} = \delta^{[2]}_{c}$$

$$\frac{\partial L}{\partial w^{[1]}_{kj}} = \delta^{[1]}_{k} a^{[0]}_{j}$$

$$\frac{\partial L}{\partial b^{[1]}_{k}} = \delta^{[1]}_{k}$$

em que:

$$\delta^{[2]}_{c} = (\hat{y}_c - y_c)$$

$$\delta^{[1]}_{k} = \sum_{c=1}^{C} w^{[2]}_{ck} \delta^{[2]}_{c} \sigma'(z^{[1]}_{k})$$

A adaptação dessas equações para vetorização fica como exercício. Perceba que essas equações são muito semelhantes às mostradas nos slides, porém há algumas diferenças, especialmente no cálculo do erro na última camada, dado que se utiliza aqui *loss function* de regressão logística, ao invés de erro quadrático. Durante a implementação do algoritmo de Back Propagation, é importante lembrar também que a derivada da função sigmóide é dada por:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Note ainda que o $L(y^{(i)}, \hat{y}^{(i)})$ se refere a apenas um exemplo de treinamento. Desse modo, lembrando que a rede neural deve ser atualizada com o gradiente de $J(\theta)$, você deve calcular as derivadas parciais de $L(y^{(i)}, \hat{y}^{(i)})$ para cada exemplo i e tirar a média. Para simplificar, não será implementado nenhum mecanismo de regularização na rede neural.

Para a segmentação de cores, para simplificar, usar-se-á apenas duas classes de cores: verde e branco, que são as cores mais abundantes no ambiente do futebol de robôs. No dataset, há algumas outras cores (como o laranja da bola). Durante o treinamento da rede, essas outras cores serão marcadas como sendo da classe 0 [0, 0], i.e. como se representassem uma classe de cor indefinida. Então, na segmentação final, as cores indefinidas são mostradas como preto. Também, por simplicidade, o dataset está usando espaço de cor RGB.

No dataset de cores fornecido, há 78990 exemplos. Como realizar um passo do Back Propagation num dataset tão grande seria muito lento, utiliza-se o conceito de mini-batch, logo a rede é treinada com Descida de Gradiente Estocástica. O mini-batch usado consiste de 100 exemplos aleatórios de treinamento de cada classe (incluindo a classe de cor indefinida). No caso do dataset considerado, há muito mais pixels verdes do que de outras cores, pois o campo do futebol de robôs é verde. Com isso, se o batch de treinamento mantivesse a mesma distribuição de cores que o dataset original, a rede daria preferência para aprender o verde em

detrimento das demais cores. Esse problema de desbalanceamento de exemplos de treinamento é comum em redes neurais. Finalmente, destaca-se que se aplica normalização nos valores de entrada para facilitar o treinamento: normalmente, os valores de cada canal de cor RGB está na faixa 0-255. Como esses são valores altos, de modo que fica ruim para uma rede com função de ativação sigmóide, usou-se como dados de entrada para a rede os valores R/255, G/255 e B/255.

3. Código Base

O código base já implementa o teste da rede neural e o treinamento e teste da segmentação de cores. Segue uma breve descrição dos arquivos fornecidos:

- `neural_network.py`: implementação da rede neural. Nessa classe, você deve implementar os métodos `forward_propagation()`, `compute_gradient_back_propagation()` e `back_propagation()`.
- `utils.py`: funções utilitárias.
- `test_neural_network.py`: 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.
- `test_color_segmentation.py`: realiza o aprendizado da segmentação de cores e exibe o resultado.

Destaca-se que os treinamentos das redes neurais podem demorar alguns minutos. Os hiperparâmetros definidos em cada *script* já foram ajustados pelo professor para o problema, mas fique à vontade para alterá-los caso sinta necessidade.

4. Tarefas

4.1. Implementação e Teste da Rede Neural

Sua primeira tarefa consiste em implementar as operações de Forward Propagation e Back Propagation da rede neural. O Forward Propagation da rede em questão segue as mesmas equações mostradas nos slides. Já o Back Propagation segue as equações mostradas na Seção 2, que são diferentes das mostradas nos slides, por conta de estarmos usando uma função de custo diferente aqui.

Não há necessidade de implementar as operações da rede neural da forma mais eficiente possível (a não ser que os treinamentos estejam demorando excessivamente nos seus testes). Caso sua implementação tenha ficado proibitivamente lenta, a ponto de não conseguir realizar as tarefas propostas, fique à vontade para conversar com o professor. Além disso, sua implementação pode se aproveitar fortemente das restrições impostas na rede: três camadas e funções de ativação sigmóide em todos os neurônios.

Após a implementação da rede, utilize o script `test_neural_network.py` para testar a sua rede. Teste primeiramente com a função de classificação `sum_gt_zero()` e

posteriormente com a `xor()`. Para alterar a função utilizada, basta alterar a variável `classification_function`. Em ambos os casos, usa-se 10 neurônios na camada escondida e taxa de aprendizado de 6. Adicione todos os gráficos auto-gerados no seu relatório e os discuta.

Observação: na notação usada nos slides, a camada $l = 0$ da rede neural é a camada de entrada, de modo que não há pesos nem biases associados a ela. Desse modo, para que a notação no código fique igual à dos slides, define-se uma posição 0 com `None` nas listas que definem as matrizes de peso e os vetores de biases. Verifique como as variáveis `weights` e `biases` são inicializadas no código para entender melhor essa observação.

4.2. Segmentação de Cores

Para realizar o aprendizado e teste da segmentação de cores, utilize o *script* `test_color_segmentation.py`. Para essa rede, usou-se 20 neurônios na camada escondida e taxa de aprendizado 6. Coloque em seu relatório as figuras geradas e discuta os resultados obtidos.

5. Entrega

A entrega consiste do código e de um relatório, submetida através do Google Classroom. Modificações nos arquivos do código base são permitidas, desde que o nome e a interface dos scripts "main" não sejam alterados. A princípio, não há limitação de número de páginas para o relatório, mas pede-se que seja sucinto. O relatório deve conter:

- Breve descrição em alto nível da sua implementação.
- Figuras que comprovem o funcionamento do seu código.

Por limitações do Google Classroom (e por motivo de facilitar a automatização da correção), entregue seu laboratório com todos os arquivos num único arquivo **.zip** (**não** utilize outras tecnologias de compactação de arquivos) com o seguinte padrão de nome: "**<login_email_google_education>_labX.zip**". Por exemplo, no meu caso, meu login Google Education é **marcos.maximo**, logo eu entregaria o lab 6 como "**marcos.maximo_lab6.zip**". **Não** crie subpastas para os arquivos da sua entrega, **deixe todos os arquivos na "raiz" do .zip**. Os relatórios devem ser entregues em formato **.pdf**.

6. Dicas

- Não é obrigatório implementar a rede neural com vetorização, mas é altamente recomendado. A diferença de desempenho é significativa: com implementação completamente vetorizada, a sua rede deve treinar praticamente instantaneamente. Uma vetorização "parcial" também é possível, caso ache mais fácil.
- Observação: quando fala-se *array* em NumPy, este *array* pode ser N-dimensional.
- Perceba que as matrizes `inputs` e `expected_outputs` possuem dimensões `(num_inputs, num_samples)` e `(num_outputs, num_samples)`, respectivamente. Isso é diferente do que vem sendo praticado nas *frameworks* de redes neurais, mas mantive assim para ficar coerente com a notação dos *slides*.

- Em `utils.py`, há uma função `sigmoid_derivative()`, que calcula a derivada da função sigmóide (funciona para *array*).
- As implementações parciais nos métodos `neural_network.py` são apenas para guiar a implementação do aluno. Fique à vontade para mudá-las, desde que a interface de código não seja quebrada.
- Para um *array* do NumPy, o operador de multiplicação de matriz elemento a elemento (*element-wise*) é `*`, enquanto a multiplicação usual de matriz usa o operador `@`.
- Diferentemente de como é feito nos *slides*, você não precisa repetir o vetor de *biases* para calcular o vetor $z^{[l]}$ de forma vetorizada. Basta somar o vetor de *biases* com a matriz resultante do produto dos pesos com o vetor de ativações da camada anterior que o NumPy se encarrega de fazer a operação corretamente. Para entender melhor, veja: https://numpy.org/doc/stable/user/basics.broadcasting.html
- A função `np.mean(x)` calcula a média de todos os elementos do *array* x. Caso queira fazer a média em apenas uma direção, usar `np.mean(x, axis=dim)`, em que `dim` é o índice da dimensão.
- Para aplicar o logaritmo neperiano em toda um *array* `x`, use `np.log(x)`.
- Caso você tenha algum problema em *runtime* durante o cálculo do *log* na função de custo, o motivo provavelmente é algum bug no seu código. Evite ficar tentando fazer "gambiarras" na função de custo para desaparecer com esse problema, provavelmente só vão mascarar algum bug.
- No cálculo dos gradientes, **não** esqueça de dividir pelo número de exemplos!
- Os tamanhos das dimensões de um *array* N-dimensional podem ser obtidos com:
  `array.shape`
- Em uma situação, eu necessitei transformar um *array* unidimensional de n elementos em uma matriz bidimensional com $n \times 1$ elementos (i.e. vetor da MAT). Um jeito fácil de fazer isso é usar:
  `x = x.reshape((x.shape[0], 1))`