Relatório do Laboratório 11: Programação Dinâmica

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 algoritmos de programação dinâmica no contexto de solução de um Processo Decisório de Markov (Markov Decision Process - MDP). Os algoritmos implementados serão avaliação de política (policy evaluation), iteração de política (policy iteration) e iteração de valor (value iteration), com o objetivo de avaliar políticas e determinar políticas ótimas para um grid world.**

***Index Terms*—MDP, policy evaluation, policy iteration, value iteration**

I. Implementação

A. Implementação de Avaliação de Política

Essa parte tratava-se da implementação da função *policy_evaluation()*, presente no arquivo *dynamic_programming.py*, fornecido pelo código base do professor. De maneira simples, essa função consiste na equação: $v_{k+1}(s) = \sum_{a \in A} \pi(a|s) r(s,a) + \gamma \sum_{a \in A} \sum_{s' \in S} \pi(a|s) p(s'|s,a) v_k(s')$, apresentada de forma bem semelhante no roteiro do laboratório [1].

Para implementá-la, os estados s (como também s') se tornaram tuplas (i, j), que foram iteradas por todo o grid world. Foram feitos loops também para iterar pelas ações a, e os resultados da equação acima foram somados ao valor associado ao estado em que se estava nessa iteração. O valor de $\pi(a|s)$ era encontrado no parâmetro *policy[current_state][action]*, assim como $r(s,a)$ estava em *grid_world.reward(current_state, action)*, $p(s'|s,a)$ estava em *grid_world.transition_probability(current_state, action, next_state)*, e $v_k(s')$ era a *policy* para um outro estado s'.

tratava-se principalmente de três ações, a se saber: primeiro, era pré-processada a imagem de entrada (através da função *preprocess_image()*; segundo, essa entrada pré-processada servia de entrada para rede, levando-a a um output; por fim, esse output era processado pelo algoritmo YOLO (através da função *process_yolo_output()*), retornando, então, informações de onde estariam a bola e os dois postes da trave. A predição da rede do segundo passo foi implementado com a função *predict()*, do Keras.

Para a implementação da rede neural conforme os parâmetros requisitados pelo roteiro do laboratório [1], utilizou-se um código fortemente inspirado das linhas de código apresentadas na seção Dicas do roteiro do laboratório [1], passando uma camada como entrada da outra camada.

A rede foi avaliada executando-se o script *make_detector_network.py*, gerando um resumo da rede, como o apresentado na Tabela 1 do roteiro do laboratório [1].

B. Implementação de Iteração de Valor

C. Implementação de Iteração de Política

D. Comparação entre Grid Worlds diferentes

1) Função detect(): Essa função tratava-se principalmente de três ações, a se saber: primeiro, era pré-processada a imagem de entrada (através da função *preprocess_image()*; segundo, essa entrada pré-processada servia de entrada para rede, levando-a a um output; por fim, esse output era processado pelo algoritmo YOLO (através da função *process_yolo_output()*), retornando, então, informações de onde estariam a bola e os dois postes da trave. A predição da rede do segundo passo foi implementado com a função *predict()*, do Keras.

2) Função preprocess_image(): Essa função foi implementada com forte inspiração nas últimas três dicas da seção Dicas fornecidas pelo roteiro [1], com o detalhe adicional de dividir os elementos do vetor da imagem por 255, a fim de transformar os valores RGB em valores de 0 a 1.

3) Função process_yolo_output(): Para processar o output através do algoritmo YOLO, percorreu-se a matriz output, guardando quais índices detinham a maior probabilidade de haver bola, a maior e a segunda maior probabilidade de haver postes de trave. Tendo posse desses índices, o restante das informações eram processados de acordo com as equações apresentadas na página 3 do roteiro do laboratório [1]. Por fim, era retornada essas informações processadas da bola e das duas traves.

II. Resultados e Conclusões

A. Implementação da rede em Keras

Ao se executar o script *make_detector_network.py*, o resultado foi igual ao apresentado na Tabela 1 do roteiro [1], comprovando a correta implementação da rede. Esse resultado pode ser visto nas Figuras 1 e 2.

B. Algoritmo YOLO

Uma amostra das imagens geradas pelo algorito de detecção de objetos YOLO foram reproduzidos nas Figuras de 3 a 7. Por

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 120, 160, 3)	0	
conv_1 (Conv2D)	(None, 120, 160, 8)	216	input_1[0][0]
norm_1 (BatchNormalization)	(None, 120, 160, 8)	32	conv_1[0][0]
leaky_relu_1 (LeakyReLU)	(None, 120, 160, 8)	0	norm_1[0][0]
conv_2 (Conv2D)	(None, 120, 160, 8)	576	leaky_relu_1[0][0]
norm_2 (BatchNormalization)	(None, 120, 160, 8)	32	conv_2[0][0]
leaky_relu_2 (LeakyReLU)	(None, 120, 160, 8)	0	norm_2[0][0]
conv_3 (Conv2D)	(None, 120, 160, 16)	1152	leaky_relu_2[0][0]
norm_3 (BatchNormalization)	(None, 120, 160, 16)	64	conv_3[0][0]
leaky_relu_3 (LeakyReLU)	(None, 120, 160, 16)	0	norm_3[0][0]
max_pool_3 (MaxPooling2D)	(None, 60, 80, 16)	0	leaky_relu_3[0][0]
conv_4 (Conv2D)	(None, 60, 80, 32)	4608	max_pool_3[0][0]
norm_4 (BatchNormalization)	(None, 60, 80, 32)	128	conv_4[0][0]
leaky_relu_4 (LeakyReLU)	(None, 60, 80, 32)	0	norm_4[0][0]
max_pool_4 (MaxPooling2D)	(None, 30, 40, 32)	0	leaky_relu_4[0][0]
conv_5 (Conv2D)	(None, 30, 40, 64)	18432	max_pool_4[0][0]
norm_5 (BatchNormalization)	(None, 30, 40, 64)	256	conv_5[0][0]
leaky_relu_5 (LeakyReLU)	(None, 30, 40, 64)	0	norm_5[0][0]
max_pool_5 (MaxPooling2D)	(None, 15, 20, 64)	0	leaky_relu_5[0][0]

Figura 1. Primeira parte do summary da rede implementada em Keras.

.

max_pool_5 (MaxPooling2D)	(None, 15, 20, 64)	0	leaky_relu_5[0][0]
conv_6 (Conv2D)	(None, 15, 20, 64)	36864	max_pool_5[0][0]
norm_6 (BatchNormalization)	(None, 15, 20, 64)	256	conv_6[0][0]
leaky_relu_6 (LeakyReLU)	(None, 15, 20, 64)	0	norm_6[0][0]
max_pool_6 (MaxPooling2D)	(None, 15, 20, 64)	0	leaky_relu_6[0][0]
conv_7 (Conv2D)	(None, 15, 20, 128)	73728	max_pool_6[0][0]
norm_7 (BatchNormalization)	(None, 15, 20, 128)	512	conv_7[0][0]
leaky_relu_7 (LeakyReLU)	(None, 15, 20, 128)	0	norm_7[0][0]
conv_skip (Conv2D)	(None, 15, 20, 128)	8192	max_pool_6[0][0]
conv_8 (Conv2D)	(None, 15, 20, 256)	294912	leaky_relu_7[0][0]
norm_skip (BatchNormalization)	(None, 15, 20, 128)	512	conv_skip[0][0]
norm_8 (BatchNormalization)	(None, 15, 20, 256)	1024	conv_8[0][0]
leaky_relu_skip (LeakyReLU)	(None, 15, 20, 128)	0	norm_skip[0][0]
leaky_relu_8 (LeakyReLU)	(None, 15, 20, 256)	0	norm_8[0][0]
concat (Concatenate)	(None, 15, 20, 384)	0	leaky_relu_skip[0][0] leaky_relu_8[0][0]
conv_9 (Conv2D)	(None, 15, 20, 10)	3850	concat[0][0]

Total params: 445,346
Trainable params: 443,938
Non-trainable params: 1,408

Figura 2. Segunda parte do summary da rede implementada em Keras, com a última linha da Figura 1 como primeira linha dessa figura.

.

essas imagens consegue-se observar o correto funcionamento do algoritmo, indicando que a implementação foi feita corretamente. Assim como esperado, então, conseguiu-se detectar a bola e os postes das traves nas imagens fornecidas pelo professor.

Tendo em vista o que foi apresentado, pode-se notar, por fim, que a rede e o algoritmo YOLO realmente se demonstrou eficaz em realizar essa detecção de objetos em imagens.

Referências

[1] M. Maximo, "Roteiro: Laboratório 11 - Programação Dinâmica". Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.

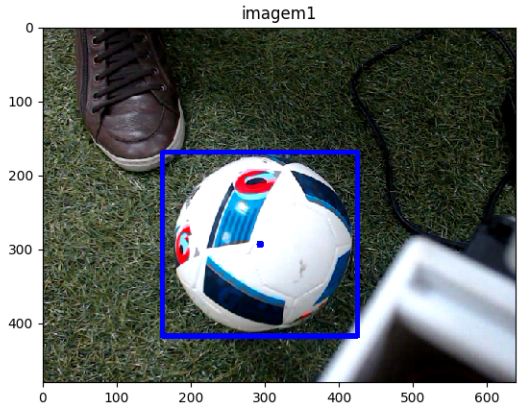

Figura 3. Detecção de uma bola através do algoritmo de YOLO.

.

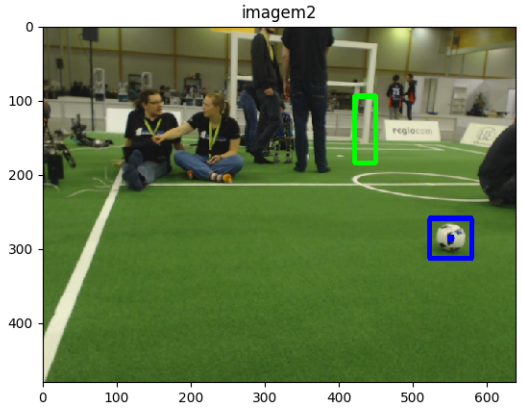

Figura 4. Detecção de uma bola e de um dos postes da trave através do algoritmo de YOLO.

.

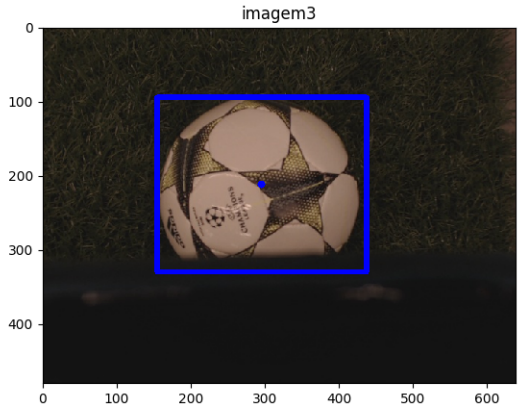

Figura 5. Detecção de uma bola (mesmo que não completamente na imagem) através do algoritmo de YOLO.

.

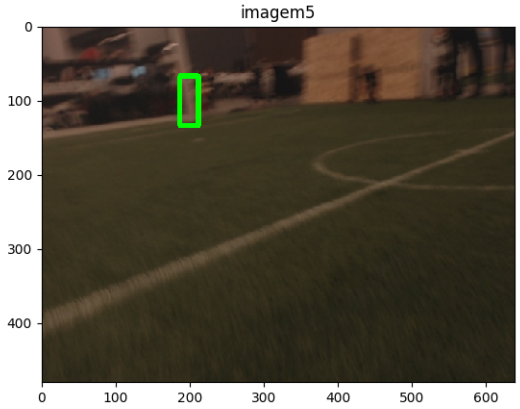

Figura 6. Detecção de um poste de trave através do algoritmo de YOLO.

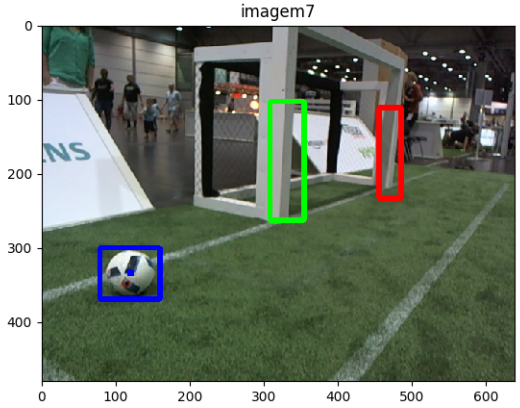

Figura 7. Detecção de uma bola e dois postes da trave através do algoritmo de YOLO.