Universidade do Minho
Mestrado Integrado em Engenharia Informática
Classificadores e Sistemas Conexionistas

Deep Learning e Tensor Flow - Criação de Autonomia

Joel Morais, A70841
Tiago Fraga, A74092

31 de Janeiro de 2019

Conteúdo

Capítulo 1

Introdução

Com a realização deste trabalho prático pretende-se conceber e implementar modelos de Deep Learning nos mais variados domínios com particular ênfase no que diz respeito à construção de modelos capazes de incutir inteligência a agentes de forma a que estes possam tomar decisões e adotar comportamentos que lhes permitam otimizar um determinado processo num determinado ambiente.

Este projeto procura criar os referidos modelos com o objetivo de resolver os ambientes **Acrobot v1** e **CartPole v1**, pertencentes ao *toolkit Gym* da *OpenAI*, usando técnicas usadas nas aulas.

Os modelos baseados em Deep Learning serão desenvolvidos utilizando a ferramenta **TensorFlow** e a *API* de alto nível **Keras**, de modo a que seja criado um agente que desempenhe, com sucesso, a tarefa de jogar estes ambientes. Esse sucesso vai ser traduzido de acordo com a eficácia que o agente terá em resolver os objetivos propostos em cada jogo. O *toolkit* da *OpenAI* fornece informações detalhadas sobre cada ambiente, para que seja possivel trabalhar com eles em cada instante ou em cada ação tomada (Figura 1).

Pretendemos como objetivo deste trabalho, implementar um modelo único que seja capaz de conter os dois casos de estudo que iremos abordar, deste modo, conseguimos dados suficientes para fazer uma análise critica do mesmo, mais aprofundada. Por fim, é necessária uma otimização dos modelos de Deep Learning de modo a que o agente possa jogar o jogo autonomamente e com o melhor resultado possível. Para isto, é necessária uma coleção do conjunto de dados até á conceção e treino do modelo em si. Em suma, o agente terá de fazer uso do modelo desenvolvido para obter a informação necessária para executar as suas tarefas.

Capítulo 2

Casos de Estudo

2.1 CartPole v1

O *Cartpole*, também conhecido por Pêndulo Invertido, é constituído por uma **linha** um **carro** e um **pêndulo vertical**. O objetivo do jogo é manter o pêndulo o máximo de tempo possivel em cima do carro ao longo da linha. O carro apenas se pode mover para a esquerda e para a direita pela linha.

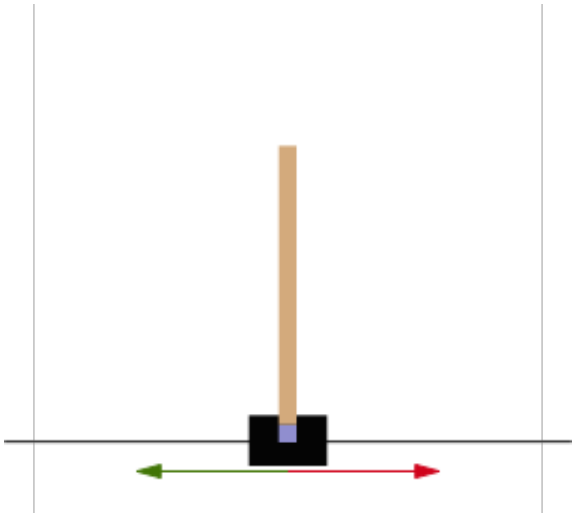

Figura 2.1: CartPole v1

A cada paço executado é possível observar a sua posição (x), a sua velocidade (x_dot), o ângulo (theta) e a velocidade angular (theta_dot).

Este ambiente é constituído por duas ações, a ação 0 para o carro se deslocar para a esquerda, e

a ação 1 para deslocar o carro para a direita

Num	Action
0	Push cart to the left
1	Push cart to the right

Figura 2.2: Documentação do CartPole

Existe um prémio com o valor de 1, por cada paço tomado e o pêndulo permaneça em cima do carro.

2.2 Acrobot v1

O segundo ambiente estudado foi o *Acrobot v1*. A descrição do OpenAI Gym referente a este jogo diz o seguinte: "O sistema *Acrobot* inclui duas articulações e dois links, onde uma das articulações está a atuar entre os dois links. Inicialmente, os links estão a pender verticalmente para baixo, e o objetivo é balouçar o da ponta do link mais inferior até uma certa altura ". Podemos fazer a analogia do *Acrobot* em relação a um ginástico a balouçar numa barra. Inicialmente, o jogo começa da seguinte maneira (Figura 4):

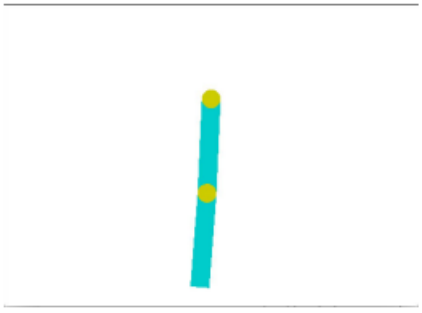

Figura 2.3: Inicio do jogo.

Seja aplicando no sentido dos ponteiros do relógio, no sentido contrário, ou em nenhum dos dois para a articulação entre os dois links, o objetivo é balançar o link inferior acima do *threshold*, como mostrado na figura 3:

O objetivo principal, tal como dito, é treinar um agente de modo a que o link inferior atinja um determinado threshold, no menor número de passos possível.

Figura 2.4: Objetivo do jogo.

No *Acrobot v1*, existe um prémio de com o valor de -1 a cada instante, até que o agente consiga atingir o seu objetivo. Tendo isto em consideração, o nosso objetivo é maximizar a pontuação final.

Capítulo 3

Modelo de Deep Learning

Em primeiro lugar, de forma a dar inicio ao projeto foi necessário ter o seguinte software instalado nas máquinas dos elementos do grupo.

1. Python 3.7
2. NumPy
3. OpenAIGym
4. TensorFlow
5. Keras

Neste capitulo iremos explicar como foi feito o povoamento do modelo com os dados necessários para posteriormente ser feito o treino do agente.

Definimos um modelo segundo *Reeinforcement Learning*, ou seja, através da aleatoriedade de alguns jogos efetuados, apenas os resultados com melhor pontuação deviam ser guardados, desta maneira o agente apenas irá aprender segundo ações que foram bem tomadas.

Começamos por inicializar dois *arrays* de forma a guardar o *training data* e os resultados aceitáveis. Este último array é construido através de um parâmetro a que chamamos de *score* minimo. Este parâmetro pode ser alterado conforme o objetivo em mente, se pretendermos otimizar os resultados finais, deve-se ao aumentar este valor, se nao for esse o objetivo então pode manter um valor nao muito alto, e desta forma o esforço computacional não é tão grande, visto que mais jogos correspondem ao pretendido. Ao logo do projeto, este valor foi sendo alterado de forma a termos vários dados para analisar.

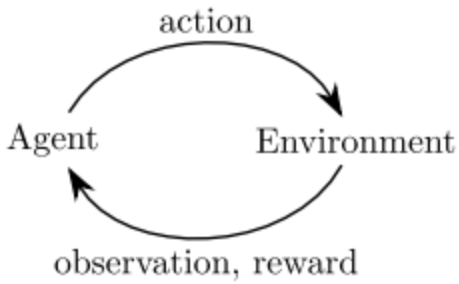

Figura 3.1: Modelo de desenvolvimento do projecto.

Para recolher os dados necessários é necessário jogar o ambiente bastantes vezes, como este modelo é baseado em aprendizagem por erro, quantos mais forem os jogos executados nesta fase mais precisos irão ser os resultados e mais observações serão feitas.

Para podermos jogar o ambiente é necessário recorrermos a ações aleatórias, onde essa ação pode levar-nos completar o jogo, bem como a perde-lo.

No caso do **CartPole** o jogo apenas tem duas ações possíveis (empurrar o carrinho para a direita ou para a esquerda), apenas precisamos que a aleatoriedade varie entre 0 e 1. No entanto, no **Acrobot** o jogo tem três ações, portanto a aleatoriedade das mesmas varia entre 0 e 2.

A cada ação executada no ambiente, este é capaz de retornar um conjunto de dados que nos fornecem informação sobre como essa ação afetou o jogo.

```
observation, reward, done, info = env.step(action)
```

Figura 3.2: Dados fornecidos pelo ambiente após uma ação.

Por fim, de forma no *training data* são guardados as ações tomadas e as observações dessas mesmas ações se esse episódio tiver um resultado final superior ao minimo pretendido.

Capítulo 4

Treino dos Agentes

Após concluir a construção do modelo Deep Learning, passamos ao treino do agente encarregue de jogar ambiente. O principal objetivo do grupo foi de conseguir criar um agente capaz de jogar o ambiente autonomamente com os melhores resultados possíveis, no entanto nao fizemos dos resultados a nossa ambição numero 1.

Para efetuar o treino do agente é necessário construir uma rede neuronal. É esta a rede que consoante o modelo treinado que lhe fornecemos, é capaz de tomar as melhores decisões para ser capaz de concluir os objetivos dos jogos.

```
def build_model(input_size, output_size):
    model = Sequential()
    model.add(Dense(128, input_dim=input_size, activation='relu'))
    model.add(Dense(52, activation='relu'))
    model.add(Dense(output_size, activation='softmax'))
    model.compile(loss='mse', optimizer=Adam())
    return model
```

Figura 4.1: Rede neuronal do agente.

Inicialmente, começamos por construir uma rede **MLP** - *Multi Layer Percepton* com a definição **Sequential**. Desta forma criamos uma rede neuronal, cujas camadas sao ligadas sequencialmente. Segundo pequenos estudos que efetuamos antes de iniciar o trabalho, este método além de ser o mais comum para este tipo de casos, é também o mais simples de implementar, portanto decidimos avançar com esta rede.

De seguida adicionamos três camadas à rede, sendo que as duas primeiras sao com ativação **relu**, os que a torna densamente conectadas. A primeira tem 128 neurónios e a segunda tem 52.

A última camada, tem ativação **softmax** pois é a opção que mais se adequa ao projeto, pois

efetua uma média ponderada de qual a melhor ação a ser executada. Além desta usamos a ativação **linear**, no entanto a primeira revelou-se como melhor opção.

De forma a otimizar a rede, ao longo do trabalho fomos alterando estes valores, tanto em numero de camadas, como o numero de neurónios em cada camada, bem como a ativação que era feita em cada camada. Como foram as camadas descritas que nos forneceram melhores resultados, optamos pelas mesmas.

```
Epoch 1/10
12236/12236 [==============================] - 1s 94us/step - loss:
0.2483
Epoch 2/10
12236/12236 [==============================] - 1s 71us/step - loss:
0.2348
Epoch 3/10
12236/12236 [==============================] - 1s 67us/step - loss:
0.2333
Epoch 4/10
12236/12236 [==============================] - 1s 68us/step - loss:
0.2334
Epoch 5/10
12236/12236 [==============================] - 1s 64us/step - loss:
0.2325
Epoch 6/10
12236/12236 [==============================] - 1s 63us/step - loss:
0.2324
Epoch 7/10
12236/12236 [==============================] - 1s 66us/step - loss:
0.2315
Epoch 8/10
12236/12236 [==============================] - 1s 65us/step - loss:
0.2318
Epoch 9/10
12236/12236 [==============================] - 1s 65us/step - loss:
0.2317
Epoch 10/10
12236/12236 [==============================] - 1s 65us/step - loss:
0.2318
```

Figura 4.2: Épocas de treino do agente.

Por fim, de forma a obter uma rede solida e capaz de responder aos objetivos propostos, efetuamos o seu treino durante 10 épocas. Quanto maior for este valor, mais prepara fica a rede para situações inesperadas durante o jogo.

Capítulo 5

Resultados Obtidos

5.1 CartPole v1

Os objetivos propostos para este ambiente foram cumpridos, uma vez que fomos capazes de criar um agente capaz de jogar autonomamente e com resultados médios a rondar os 500.

Os resultados foram obtidos através da ultima parte de desenvolvimento do projeto, após a criação do modelo e o treino do agente: o jogo.

Para dar inicio ao jogo, é necessário começar com uma ação aleatória. Após a sua execução, é gerado as observações e o prémio dessa ação no ambiente jogável. Por fim, com a rede treinada e com a primeira ação executada, o agente toma decisões para continuar a jogar rumo ao objetivo final.

Para auxiliar o agente na tomada de decisão da ação correta, é utilizado o método **predict** da biblioteca *Keras*.

No fim dos jogos pretendidos, é feita uma média dos resultados obtidos, bem como a percentagem das escolhas que o agente toma(entre 0 e 1).

Para obter a pontuação média referida no inicio do capitulo, foram usados os parâmetros descritos nos capítulos anteriores, mas o que mais ressalvamos foi o facto de que no inicio do projeto, de forma a construir o modelo de Deep Learning, apenas guardamos os jogos em que o agente teve um score superior a 100, portanto, nao é de estranhar que este seja capaz de obter estes bons resultados.

```
[500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
247.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 259.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 264.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 241.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 255.0, 500.0, 500.0,
500.0, 245.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0]
Average Score: 485.11
choice 1:0.5007936344334275  choice 0:0.49920636556657255
```

Figura 5.1: Resultados médios obtidos.

5.2 Acrobot v1

Neste ambiente, não conseguimos atingir os resultados pretendidos, ou seja, não conseguimos obter um agente capaz jogar autonomamente.

Esta situação deve-se ao facto de que no momento da construção do modelo, quando eram jogados jogos aleatórios, o agente nunca conseguiu termina-los atingindo o *threshold*. Desta forma, nunca houve jogos que correspondessem ao minimo exigido(terminar o jogo), portanto não conseguimos obter dados de treino.

Em todos os casos estudados, obtivemos sempre o valor minimo possivel, ou seja, como o ambiente limita a 500 passos por jogo, as pontuações foram sempre de -500, pois o prémio após cada ação é com o valor de -1.

Apesar de várias otimizações e alterações que foram feitas ao modelo, em nada alterou os resultados. Fomos guardando as observações em que o agente esteve mais perto do objetivo, no entanto, desta forma também não foi possivel.

Capítulo 6

Conclusões e Trabalho Futuro

Após a análise dos resultados podemos afirmar que o nosso modelo apenas é capaz de jogar um dos ambientes estudados.

No primeiro ambiente, o **CartPole**, foi possivel atingir o objetivo final e com bons resultados. Apenas será necessário fazer otimizações ao modelo atual, caso os resultados já obtidos nao sejam suficientes para o estudioso em causa. Caso contrário, apresenta-se como um modelo bastante robusto e confiável.

No segundo ambiente, o **Acrobot**, não foi possivel atingir o resultado final, portanto foi possivel constatar as debilidades do modelo no momento em que o ambiente jogável foi alterado. Para resolver este problema no futuro será necessário aplicar técnicas de *Reeiforcement Learning* mais avanças e esforços computacionais superiores aos que os elementos do grupo têm neste momento. Deste modo, é possivel procurar com exaustão casos em que o agente consiga terminar o jogo com sucesso e assim criar uma rede neuronal capaz de jogar autonomamente.

Apêndice A

Anexo

Apresentamos o código das funções implementadas.

- **CartPole v1**

```
import gym
import random
import numpy as np
from tensorflow import keras
from keras.models      import Sequential
from keras.layers      import Dense
from keras.optimizers import Adam


# MODEL PREPARATION


def play_a_random_game_first():
    for step_index in range(goal_steps):
        #env.render()
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        print("Step {}:".format(step_index))
        print("action: {}".format(action))
        print("observation: {}".format(observation))
        print("reward: {}".format(reward))
        print("done: {}".format(done))
        print("info: {}".format(info))
        if done:
            break
    env.reset()
```

```
27
28
29  def model_data_preparation():
30      training_data = []
31      accepted_scores = []
32      for game_index in range(intial_games):
33          score = 0
34          game_memory = []
35          previous_observation = []
36          for step_index in range(goal_steps):
37              action = random.randrange(0, 2)
38              observation, reward, done, info = env.step(action)
39
40              if len(previous_observation) > 0:
41                  game_memory.append([previous_observation, action])
42
43              previous_observation = observation
44              score += reward
45              if done:
46                  break
47
48          if score >= score_requirement:
49              accepted_scores.append(score)
50              for data in game_memory:
51                  if data[1] == 1:
52                      output = [0, 1]
53                  elif data[1] == 0:
54                      output = [1, 0]
55                  training_data.append([data[0], output])
56
57          env.reset()
58
59      print(accepted_scores)
60
61      return training_data
62
63
64  def build_model(input_size, output_size):
65      model = Sequential()
66      model.add(Dense(128, input_dim=input_size, activation='relu'))
67      model.add(Dense(52, activation='relu'))
68      model.add(Dense(output_size, activation='linear'))
69      model.compile(loss='mse', optimizer=Adam())
```

```
70      return model
71
72
73  def train_model(training_data):
74      X = np.array([i[0] for i in training_data]).reshape(-1, len(
            training_data[0][0]))
75      y = np.array([i[1] for i in training_data]).reshape(-1, len(
            training_data[0][1]))
76      model = build_model(input_size=len(X[0]), output_size=len(y[0]))
77
78      model.fit(X, y, epochs=10)
79      return model
80
81
82
83  # MAIN
84
85  env = gym.make('CartPole-v1')
86  env.reset()
87  goal_steps = 500
88  score_requirement = 60
89  intial_games = 10000
90
91
92  play_a_random_game_first()
93  training_data = model_data_preparation()
94  trained_model = train_model(training_data)
95
96
97
98  scores = []
99  choices = []
100 for each_game in range(1000):
101     score = 0
102     prev_obs = []
103     for step_index in range(goal_steps):
104         env.render()
105         if len(prev_obs)==0:
106             action = random.randrange(0,2)
107         else:
108             action = np.argmax(trained_model.predict(prev_obs.reshape(-1,
                    len(prev_obs)))[0])
109
```

```
            choices.append(action)
            new_observation, reward, done, info = env.step(action)
            prev_obs = new_observation
            score+=reward
            if done:
                break

        env.reset()
        scores.append(score)


print(scores)
print('Average Score:',sum(scores)/len(scores))
print('choice 1:{}  choice 0:{}'.format(choices.count(1)/len(choices),
    choices.count(0)/len(choices)))
```

- **Acrobot v1**

```

import gym
import random
import numpy as np
import keras
from statistics import mean, median
from collections import Counter


from keras.models import Model, Sequential, load_model
from keras.layers import Input, Dense
from keras.optimizers import Adam
LR = 1e-3



# MODEL PREPARATION


def showGame(nr = 10):

    for _ in range(nr):
        env.reset()

        s2s, s3s = [], []
```

```
26          while True:
27              #env.render()
28
29              #action = 2
30              action = random.randrange(-1,2)
31              observation, reward, done, info = env.step(action)
32              #print (observation, reward)
33              #print (env.state)
34              _,_,_,_, s2,s3 = observation
35              s2s.append(s2)
36              s3s.append(s3)
37              if done:  break
38          #print(np.mean(np.array(s2s)))
39          #print(np.mean(np.array(s3s)))
40
41
42
43
44  def saveGoodGames(nr=10000):
45      observations = []
46      actions = []
47      minReward = -100
48
49      for i in range(nr):
50          env.reset()
51          action = env.action_space.sample()
52
53          obserVationList = []
54          actionList = []
55          score = 0
56
57          while True:
58
59
60              env.render()
61
62              observation, reward, done, info = env.step(action)
63              print(observation)
64              action = env.action_space.sample()
65              obserVationList.append(observation)
66              if action == 1:
67                  actionList.append([0,1,0] )
68              elif action == 0:
```

```
69                  actionList.append([1,0,0])
70              else:
71                  actionList.append([0,0,1])
72
73              score += reward
74              if done:  break
75
76
77          if score > minReward:
78              print(score)
79              observations.extend(obserVationList)
80              actions.extend(actionList)
81      observations = np.array(observations)
82      actions = np.array(actions)
83      return observations, actions
84
85
86  def trainModell(observations, actions):
87
88      model = Sequential()
89      model.add(Dense(128,  activation='relu'))
90      model.add(Dense(256,  activation='relu'))
91      model.add(Dense(256,  activation='relu'))
92      model.add(Dense(2,  activation='softmax'))
93
94      model.compile(optimizer=Adam())
95
96      model.fit(observations, actions, epochs=10)
97      return model
98
99
100 def playGames(nr, ai):
101
102     observations = []
103     actions = []
104     minReward = 70
105     scores=0
106     scores = []
107
108     for i in range(nr):
109         env.reset()
110         action = env.action_space.sample()
111
```

```
112         obserVationList = []
113         actionList = []
114         score=0
115         while True:
116             #env.render()
117 
118             observation, reward, done, info = env.step(action)
119             action = np.argmax(ai.predict(observation.reshape(1,4)))
120             obserVationList.append(observation)
121             if action == 1:
122                 actionList.append([0,1] )
123             elif action == 0:
124                 actionList.append([1,0])
125             score += 1
126             #score += reward
127             if done:  break
128 
129 
130         print(score)
131         scores.append(score)
132         if score > minReward:
133             observations.extend(obserVationList)
134             actions.extend(actionList)
135     observations = np.array(observations)
136     actions = np.array(actions)
137     print (np.mean(scores))
138     return observations, actions
139 
140 
141 
142 # MAIN
143 
144 
145 gym.envs.register(
146         id='Acrobot-v2',entry_point='gym.envs.classic_control:AcrobotEnv',
                max_episode_steps=1000
147         )
148 
149 env = gym.make('Acrobot-v2')
150 env.reset()
151 
152 showGame(1)
153 print("Loading ...")
```

```
154 obs, acts = saveGoodGames()
155 print ('training 1st modell')
156 #firstModel = trainModell(obs, acts)
157 print("Playing the games ...")
158 #obs, acts = playGames(1000, firstModel)
```