Relatório P4 – Train a Smartcab to Drive

13 de outubro de 2016

1 Sobre a implementação

1.1 Q-Value

O objeto *QValue* encontrado em *agent.py* guarda 1 estado s e o mapeamento de ações a para $Q(s, a)$. Preferi essa representação ao invés de gravar um objeto para cada para (s, a) pois isso tornou mais fácil a obtenção de $\arg\max_a Q(s, a)$.

2 Parâmetros do programa

Criei algumas parâmetros para a linha de comando do programa. Abaixo seguem as descrições de cada um e como utilizá-los.

2.1 Parâmetro α

A taxa de aprendizado $\alpha \in \mathbb{R}, \alpha \in [0, 1]$ determina o quanto que um novo dado adquirido irá sobrescrever o antigo. Se 0, o agente não aprende coisa alguma; se 1, apenas a informação mais recente é considerada pelo agente.

Foi utilizado um valor constante durante todo o procedimento e ele pode ser configurado via parâmetro `alpha`:

```
$ python smartcab/agent.py -alpha <valor>
```

2.2 Parâmetro γ

O fator de desconto $\gamma \in \mathbb{R}, \gamma \in [0, 1]$ determina a importância dos futuros *rewards*. Se 0, o algoritmo leva em consideração apenas o *reward* atual, enquanto um valor próximo de 1 otimiza o aprendizado levando em consideração melhores valores no longo prazo.

O valor γ pode ser configurado via parâmetro `gamma`:

```
$ python smartcab/agent.py -gamma <valor>
```

2.3 Parâmetro ϵ

A estratégia utilizada no aprendizado foi ϵ-*greedy*, onde, com probabilidade ϵ, é escolhida uma ação aleatória e, com probabilidade $1 - \epsilon$, é escolhida a ação que maximiza $Q(s, a)$, onde s é um estado e a é uma ação. O valor sofre um decaimento por $\epsilon = \epsilon * c, 0 < c < 1$, independente entre os diversos *Q-values*.

O valor $\epsilon \in [0, 1]$ pode ser configurado via parâmetro `epsilon`:

```
$ python smartcab/agent.py -epsilon <valor>
```

2.4 Número de agentes *dummy*

O parâmetro `num-dummies` indica o número de veículos adicionais que estão percorrendo o ambiente:

```
$ python smartcab/agent.py -num-dummies <valor>
```

2.5 *Grid Search*

Para verificar a melhor parametrização, foi criada a flag `grid-search`. Os parâmetros pesquisados são:

- $\alpha \in (0.1, 0.2, 0.3 \ldots 1)$
- $\gamma \in (0, 0.1, 0.2 \ldots 1)$
- $\epsilon \in (0, 0.1, 0.2 \ldots 1)$

O ambiente será configurado com 3 agentes *dummy*.Para cada combinação dos 3 parâmetros, são executadas 10 rodadas de aprendizagem independentes contendo 100 viagens cada, em que as 10 últimas viagens de cada rodada são analisadas, gerando dois valores:

- Valor médio do *reward* por ação efetuada.
- Proporção do número de viagens realizadas com sucesso.

Para utilizar o *grid-search*, deve ser usando a seguinte linha de comando:

```
$ python smartcab/agent.py --grid-search > /dev/null
```

A saída do programa será uma listagem dos seguintes atributos em ordem:

- Média de *reward* por ação.
- Proporção do número de viagens realizadas com sucesso.
- α – *learning rate*
- γ – *discouting factor*
- ϵ – ϵ-*greedy factor*

Se `grid-search` for utilizado, os parâmetros `alpha`, `gamma`, `epsilon` e `num-dummies` serão ignorados.

2.6 *compare-best*

Foi criado um método chamado *compare_ best*, que implementa a lógica do veículo ficar parado ao observar o semáforo no estado vermelho e seguir a direção definida por *next_ waypoint*, a menos dos seguintes casos:

- Se *light* for *green*, a ação *left* é permitida se o valor observado de *oncoming* for diferente de *right* e *forward*.
- Se *light* for *red*, é permitida a ação *right* se o valor observado de *left* for diferente de *forward*.

Além disso, o programa aceita o parâmetro `compare-best`, da seguinte maneira:

```
$ python smartcab/agent.py --compare-best
```

Opcionalmente, podem ser passados os parâmetros `alpha`, `gamma`, `epsilon` e `num-dummies`. O agente executa 100 viagens e, ao final, para todo estado possível, é feita uma comparação entre a ação escolhida pelo método *compare_ best* e a escolhida pelo agente. São listados todos os estados observados pelo menos uma vez pelo agente, as duas ações escolhidas e o total de vezes que o estado foi observado. A última linha contem 2 proporções:

- Proporção entre o total de estados em que as ações escolhidas foram iguais, dividido pelo total de estados.
- Proporção entre o total de estados em que as ações escolhidas foram iguais, dividido pelo total de estados, ponderada pelo número de vezes que o estado foi observado.

3 Perguntas e respostas

1. **Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?**

Utilizando ações escolhidas aleatoriamente, o carro chega ao destino aproximadamente 20% das vezes.

É possível notar que o mapa se compara como se as extremidades fossem conectadas. Por exemplo, se um veículo encontra-se na última coluna do mapa e recebe um comando para seguir no sentido leste, na próxima iteração ele aparecerá na primeira coluna. No entanto, o parâmetro *next_ waypoint* parece não levar isso em consideração, gerando caminhos com comprimento maior do que o necessário.

Verifica-se também que ações que desobedeceriam leis de trânsito ou provocariam colisões não são efetuadas. O veículo permanece parado em sua posição do estado anterior, com um $reward < 0$.

Atributo	Possíveis valores	Número de valores
light	*red, green*	2
left	*None, left, forward, right*	4
forward	*None, left, forward, right*	4
right	*None, left, forward, right*	4
next_ waypoint	*left, forward, right*	3
Total de valores	384	

Tabela 1: Componentes de um estado e número de valores possíveis.

2. **What states have you identified that are appropriate for modeling the smartcab and environment? Why do you believe each of these states to be appropriate for this problem?**

O estado utilizado contem os seguintes atributos:

- *light* – assume um dentre os seguintes valores {*red*, *green*}. Indica basicamente se o veículo pode seguir ou se deve permanecer parado.
- *next_ waypoint* – assume um dentre os valores {*left*, *forward*, *right*}. Indica qual a direção e sentido que o veículo deve seguir para se aproximar do objetivo. Além disso, existem regras específicas relacionadas ao par (*light*, *next_ waypoint*), nos casos (*green*, *left*) e (*red*, *right*), que alteram o valor do *reward* da ação.
- *oncoming*, *left*, *right* – assume um dentre os valores {*None*, *left*, *forward*, *right*}. É relevante pois definem se existem carros vindo de outras direções e para onde eles se dirigem.

3. **How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?**

Para obter o total de estados, devemos fazer um produtório do número de estados possíveis para cada atributo. Conforme visto na tabela 1, o valor é 384. O parâmetro *deadline* foi considerado, mas sua inclusão iria aumentar muito o conjunto e, por isso, foi descartado.

Para a configuração definida (100 execuções com 3 carros *dummy* no mesmo ambiente), me parece pouco provável que todos os estados sejam visitados pelo menos uma vez. Para tal análise, é possível rodar o programa com a opção *–compare_ best* (será melhor explicada mais adiante).

```
$ python smartcab/agent.py --compare-best
```

Ao final da execução, será mostrada uma lista com todos os estados e a quantidade de vezes que cada um foi utilizado. Com a parametrização padrão

Recompensa média	Proporção de viagens com sucesso	α	γ	ϵ
1.90755	1.00	0.6	0.3	1.0
1.93843	1.00	0.6	0.4	0.8
1.98036	1.00	0.2	0.5	1.0
2.06157	1.00	0.6	0.4	1.0

Tabela 2: Componentes de um estado e número de valores possíveis.

(*num-dummies*=3, $\alpha = 0.1$, $\gamma = 0.1$, $\epsilon = 1.0$), apenas 31 estados tiveram contagem >= 1.

Se for analisado todos os estados, o Q-Learning não conseguirá obter dados suficientes para o aprendizado de 100% do conjunto. Na prática, isso não é tão problemático pois, nesse caso específico, 10 estados correspondem à 95.97% das entradas que o veículo recebe durante as 100 viagens (1216 de um total de 1267 chamadas ao método *LearningAgent.update*).

4. **What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?**

É possível perceber claramente que, aos poucos, o agente começa a seguir a ação indicada por *next_ waypoint*, sem deixar de considerar que deve ter ação = *None* nos casos em que deve ficar parado. Também é possível verificar que o número de ações escolhidas que recebem *reward* < 0 tornam-se mais raros.

Tal comportamento é explicado pelo equilíbro entre *exploration* e *exploitation* implementada via estratégia ϵ-*greedy*; ela define que, com probabilidade ϵ, é escolhida uma ação aleatória e, com probabilidade $(1 - \epsilon)$, é escolhida a ação que maximiza o *Q-value* do estado atual. Como o valor de ϵ vai decaindo com o tempo, as primeiras ações são majoritamente aleatórias e de contexto exploratóro, ou seja, para cada estado possível, diferentes ações são executadas e suas respectivas recompensas são obtidas. Tal comportamento permite que os *Q-values* convirjam para um modelo em que seja possível ordenar as ações de acordo com a recompensa futura esperada. Após ocorrer a convergência, entra a fase de *exploitation*, em que agente seleciona, para cada estado, a melhor ação possível.

5. **Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?**

Para obter uma melhor parametrização, utilizei a opção `grid-search`. Os parâmetros *alpha*, *gamma* e *epsilon* foram analisados. A tabela 2 mostra o resultado de tal procedimento.

A melhor parametrização foi $(\alpha = 0.6, \gamma = 0.4, \epsilon = 1.0)$. O agente parece se comportar bem, usualmente consegue chegar ao destino e comete poucos erros

de escolha de ação, embora não com 100% de acerto nas escolhas das ações.

6. **Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties? How would you describe an optimal policy for this problem?**

A implementação que considero ótima foi descrita na seção sobre o parâmetro `compare-best`. Em geral, a proporção ponderada de escolhas iguais entre o algoritmo ótimo e a ação que maximiza o *Q-value* fica em torno de 97%. Para verificar tal afirmação, basta rodar:

```
$ python smartcab/agent.py -alpha 0.6 -gamma 0.4 \
  -epsilon 1 --compare-best
```

É interessante notar que, quando a política de escolha da ação seleciona aquela que maximiza o *Q-value* (não sendo, portanto, a escolha aleatória), é muito raro encontrar ações em que $reward < 0$, ou seja, o agente basicamente não tenta cometer infrações ou entrar em colisão com outros veículos.

Garantir que o agente minimize o tempo até o destino é mais difícil. Se, para um dado estado s, a melhor ação a é do veículo andar em alguma direção, o aprendizado depende muito de, na etapa de *exploration*, o sorteio aleatório indicar a pelo menos uma vez; o *reward* positivo teoricamente fará com que a seja a ação que maximize o *Q-value* e, consequentemente, seja escolhido quando a etapa de *exploitation* for iniciada. Ocorre que é perfeitamente possível que as únicas ações escolhidas aleatoriamente sejam *None* e as outras duas direções que irão gerar *reward* negativo; assim, podem ocorrer casos do agente escolher ficar parado em situações em que deveria se movimentar.

Uma má sequência de escolhas aleatória pode gerar casos mais extremos: se, para o estado em que o semáforo está verde, não há carros em *left*, *oncoming* e *right*, e *next_ direction=forward*, não houver um sorteio da ação *forward*, é muito provável que a ação que maximizará o *Q-value* desse estado será *None*. Nesse cenário, praticamente nenhuma viagem é concluída com sucesso.