Agentes Autônomos e Super Mario World

Fabrício Olivetti de França

Universidade Federal do ABC

Tópicos

1. Agentes Autônomos
2. Exemplo: Super Mario World
3. Super Mario World Reinforcement Learning
4. Instalação do Sistema
5. Projeto Final

UFABC

Agentes Autônomos

Agentes Autônomos

Um *Agente Autônomo* é um agente com inteligência artificial que executa uma tarefa envolvendo uma ou mais decisões sem interferência externa.

Estudado principalmente em Inteligência Artificial e utiliza Aprendizado de Máquina para automatizar a tarefa de *ensinar* o agente.

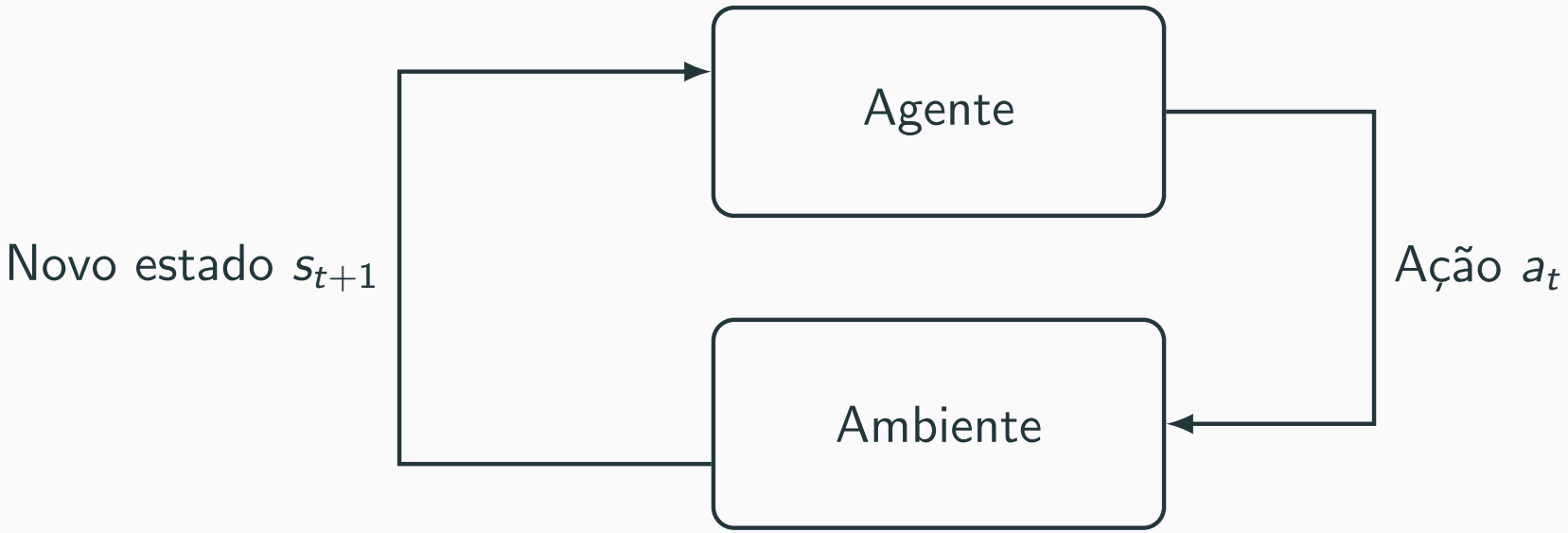
Agentes Autônomos

Características de um Agente Autônomo:

- Autonomia: não requer interferência externa para a tomada de decisão.
- Localização: é capaz de se situar no ambiente em que age.
- Interação: interage com o ambiente para adquirir novas informações.
- Inteligência: consegue agir de acordo com o estado do ambiente para atingir um determinado objetivo.

Agentes Autônomos

Basicamente, queremos um sistema em que o agente recebe o estado atual do ambiente e execute uma ação nesse ambiente para então receber um novo estado.



UFABC

Agentes Autônomos

Ou seja, queremos encontrar uma função $f : S \rightarrow A$, sendo S o conjunto de estados possíveis do sistema e A o conjunto de possíveis ações que podem ser executadas pelo agente.

Note que esse problema é similar ao problema de classificação, porém temos alguns desafios extras.

Agentes Autônomos

Durante as aulas sobre Aprendizado Supervisionado, trabalhamos com conjuntos de exemplos $\{(\mathbf{x}, y)\}$ que descreviam exatamente a saída esperada para determinados vetores de entrada.

Nos Agentes Autônomos, nem sempre isso é possível ou economicamente viável.

Agentes Autônomos

Considerem os seguintes exemplos de agentes autônomos:

- Automóvel sem motorista.
- Casa Inteligente.
- Visita de rotina a pacientes.
- Sistema anti-invasão de computadores.

Agentes Autônomos

Considerem os seguintes exemplos de agentes autônomos:

- Automóvel sem motorista.
- Casa Inteligente.
- Visita de rotina a pacientes.
- Sistema anti-invasão de computadores.

Como você coletaria um conjunto de exemplos para o aprendizado?

Agentes Autônomos

Em alguns desses casos existem múltiplas ações possíveis.

- Automóvel sem motorista.
- Casa Inteligente.
- Visita de rotina a pacientes.
- Sistema anti-invasão de computadores.

Agentes Autônomos

Em outros não temos como gerar contra-exemplos de ações que **não** deveriam ser realizadas.

- Automóvel sem motorista.
- Casa Inteligente.
- Visita de rotina a pacientes.
- Sistema anti-invasão de computadores.

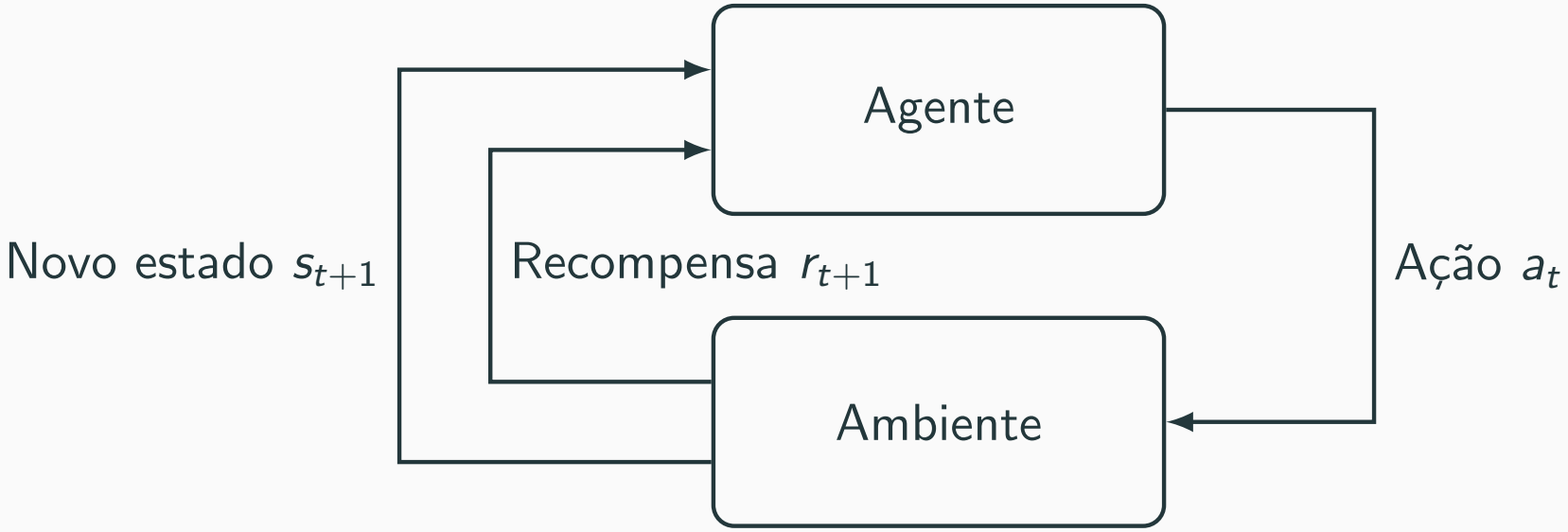
Agentes Autônomos

Também temos casos em que só saberemos a ação correta após uma série de ações serem aplicadas.

- Automóvel sem motorista.
- Casa Inteligente.
- Visita de rotina a pacientes.
- Sistema anti-invasão de computadores.

Agentes Autônomos

Uma outra forma de entender um ambiente de um agente autônomo é quando temos pelo menos uma recompensa instantânea associada a ação realizada:



A recompensa pode ser positiva ou negativa.

Agentes Autônomos

Quais recompensas podemos atribuir para possíveis ações dos sistemas de exemplo?

- Automóvel sem motorista.
- Casa Inteligente.
- Visita de rotina a pacientes.
- Sistema anti-invasão de computadores.

Agentes Autônomos

Quais recompensas podemos atribuir para possíveis ações dos sistemas de exemplo?

- Automóvel sem motorista: $r_t = d(x, x_{pista})^{-1} - d(x, x_{obstáculo})^{-1}$.
- Casa Inteligente: $r_t = s(x) - c(x)$ (sensores corporais indicam relaxamento, número de chutes nos aparelhos inteligentes).
- Visita de rotina a pacientes $r_t = diag(x) + conforto(x) - alter(x)$.
- Sistema anti-invasão de computadores $r_t = -falhas(x) - recurso(x)$.

Nesse caso queremos encontrar uma função $f : S \times A \rightarrow \mathbb{R}$, ou seja, uma função que recebe um estado e uma ação e retorna a recompensa instantânea.

Com isso, nosso problema se torna um problema de regressão.

O aprendizado dessa função caracteriza o Aprendizado por Reforço, pois desejamos aprender apenas a recompensa instantânea que reforça ou desestimula uma determinada ação executada.

Formalização Matemática

A formalização desse problema é definida através de um Processo de Decisão de Markov (*Markov Decision Process - MDP*) como:

$(S, A, R, \mathbb{P}, \gamma)$

com:

- S: conjunto de possíveis estados.
- A: conjunto de ações.
- $R : S \times A \to \mathbb{R}$: mapa de recompensa para cada par de estado e ação.
- $\mathbb{P}$: probabilidade de transição de um estado para outro dada uma ação.
- γ: fator de desconto.

Markov Decision Process

1. No instante $t = 0$, inicia em um estado inicial $s_0 \sim p(s_0)$.
2. O agente seleciona uma ação a_t.
3. Calcula recompensa r_t.
4. Amostra o próximo estado $s_{t+1} \sim p(s_t, a_t)$.
5. Agente recebe a recompensa r_t e o novo estado s_{t+1}.
6. Se s_{t+1} não for um estado final, retorna ao passo 2.

Markov Decision Process

Uma política $\pi : S \rightarrow A$ é uma função que dado um estado retorna a próxima ação a ser executada pelo agente.

O objetivo do nosso processo de decisão é encontrar uma política ótima π^* que maximiza a recompensa cumulativa descontada: $\sum_{t \geq 0} \gamma^t r_t$.

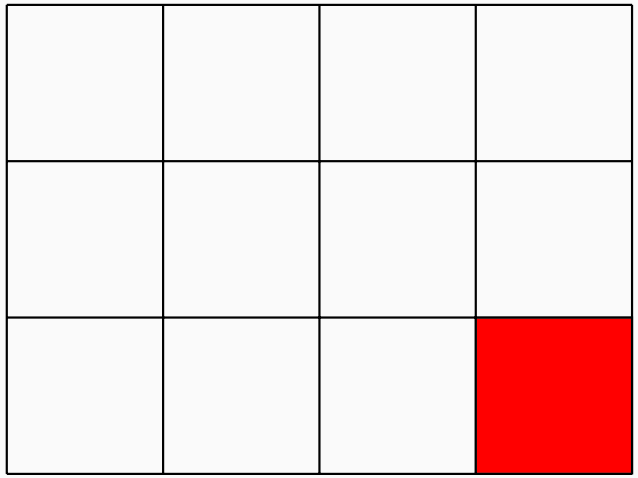
UFABC

Exemplo simples

$A = \{\rightarrow, \leftarrow, \downarrow, \uparrow\}$

$R = -1$, para qualquer entrada.

S =

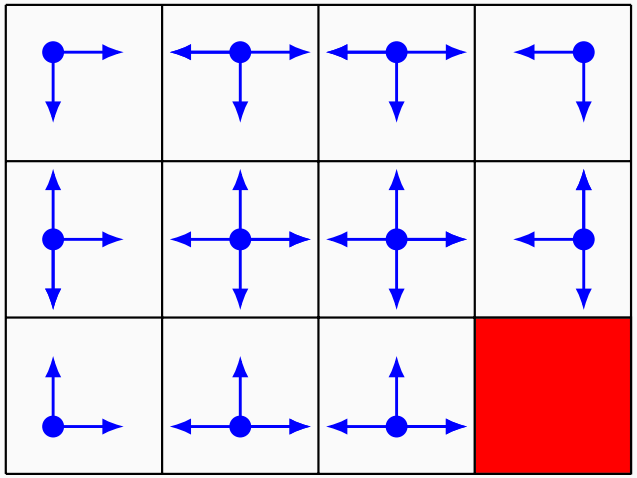


O estado final é a célula marcada em vermelho.

UFABC

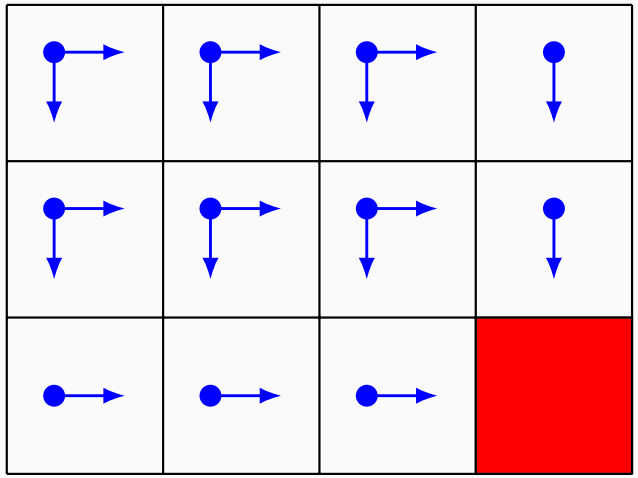
Exemplo simples

Uma política aleatória é ilustrada abaixo:

$S =$ 

Exemplo simples

E a política ótima:

$S =$ 

Desafios

- Em muitas situações o estado é uma visão incompleta, levando a mais de uma possível consequência para uma determinada ação.
- A transição para o novo estado pode ser estocástico, com incertezas.
- A ação pode não ser executada com a precisão esperada.

Exemplo: Super Mario World

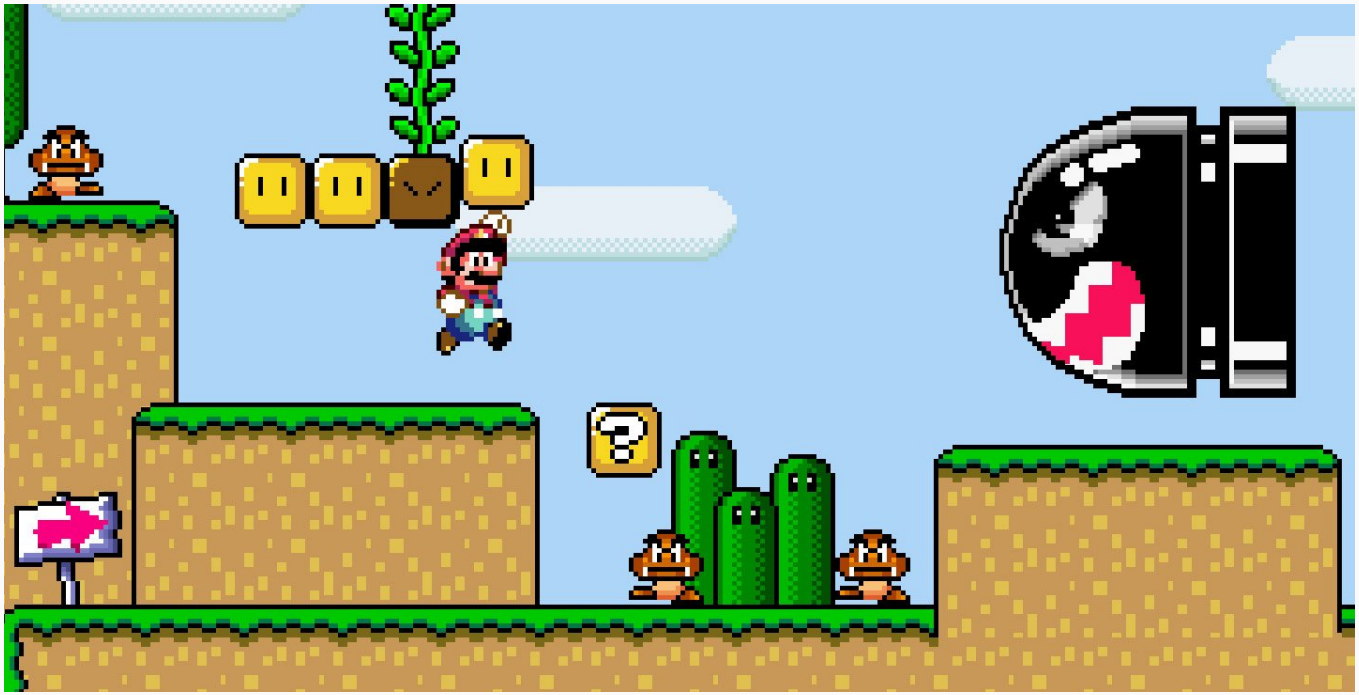
Super Mario World

O encanador Mario está na terra dos dinossauros para resgatar a princesa do reino dos cogumelos que foi sequestrada por uma tartaruga gigante.

UFABC

Super Mario World

O jogo é um side-scroller de plataformas em que o jogador controla o Mario e tem como objetivo atingir o final da fase contendo diversos inimigos e obstáculos.



UFABC

Super Mario World

Para isso ele conta com as seguintes ações que podem ser combinadas:

- Andar para Direita ou Esquerda.
- Correr.
- Pegar item carregável.
- Pular.
- Pular e Girar.
- Soltar fogo (em certas condições).
- Girar a capa (em certas condições).

Super Mario World

O jogo foi lançado em 1990 pela Nintendo para o video-game Super Nintendo (SNES).

Faz sucesso até hoje com suas diversas variações criadas por fãs e através do relançamento do jogo original em outras plataformas.

Mapa de Memória

Durante a execução do jogo, o sistema mantém em memória todas as informações relevantes do estado atual do jogo:

- Cores de cada pixel da tela.
- Pontuação atual.
- Sprites atualmente utilizados.
- Situação atual do Mario.
- Posição atual dentro da fase.
- Situação dos inimigos visíveis.
- dentre outras.

A cada ação realizada pelo jogador, essa informação é atualizada de acordo.

UFABC

Mapa de Memória

As informações de memória podem ser encontradas nesse site: https://www.smwcentral.net/?p=maptype=ram.

Foram informações coletadas por pessoas envolvidas com emuladores, ROM hacks e TAS.

A coleta foi feita através de ferramentas de mapeamento de memória e testes manuais.

Super Mario World Reinforcement Learning

Emulando Super Mario World

Existem diversos emuladores de Super Nintendo capazes de reproduzir esse jogo e tantos outros.

Poucos deles possuem uma interface para criar agentes autônomos.

Retro Learning Environment foi criado para solucionar esse problema.

Retro Learning Environment

Interface programática capaz de fazer uma interface com diversos emuladores.

Atualmente suporta alguns jogos de Atari e SNES utilizando como base os emuladores Stella e Snes9x, respectivamente.

API disponível em C++ e Python.

Retro Learning Environment

Métodos que utilizaremos para os projetos:

```
loadROM(rom, core)   # Carrega a rom de um jogo,
                     # core deve ser a string 'snes'
act(action)          # A ação a ser executada
game_over()          # Retorna True se perdeu o jogo
getLegalActionSet()  # Retorna a lista das ações válidas
getScreenRGB()       # Retorna uma array de dimensões (a, l, 4)
                     # com informações de cada pixel da tela
getRAM()             # Retorna o conteúdo da memória RAM
                     # como uma array de bytes
```

UFABC

Retro Learning Environment

As ações úteis do Super Mario World são definidas como:

```
actions_map = {'noop':0, 'down':32, 'up':16, 'jump':1, 'spin':3,
               'left':64, 'jumpleft':65, 'runleft':66, 'runjumpl
               'right':128, 'jumpright':129, 'runright':130, 'ru
               'spin':256, 'spinright':384, 'runspinright':386,
               }
```

Verifiquem que cada botão é um bit de um número binário de 8 bits.

UFABC

Estado

Como opções de estado temos as informações de pixel da tela ou extrairmos nossa própria matriz representando as informações importantes da tela atual.

A primeira opção já está pronta, mas é mais difícil do computador aprender; na segunda podemos simplificar bastante a tarefa de aprendizado, mas teremos um trabalhinho para descobrir como extrair tais informações.

Raw Pixels vs Extração de Atributos

Raw Pixels:

- Informação completa.
- Não necessita de pré-processamento.
- Um pixel sozinho não contém informação útil sobre o estado atual.
- A quantidade de estados possíveis se torna enorme.

Raw Pixels vs Extração de Atributos

Extração de atributos:

- Informação parcial.
- Necessita de um trabalho intenso para conseguir extrair os atributos corretamente.
- Cada entrada do mapa de estado tem um significado completo.
- A quantidade de estados possíveis é bem menor que a representação Raw Pixels.

Extração de Atributos

Para esse trabalho, vamos gerar uma matriz de estado em que cada posição representa um bloco de 16x16 pixels com valores $\{-1, 0, 1\}$:

- 0: espaço vazio, se for na última linha representa um buraco.
- $+1$: blocos que os personagens podem andar em cima ou colidir.
- -1: adversários ou qualquer objeto que pode ferir o personagem.

Extração de Atributos

Essa matriz está centralizada no personagem e sua dimensão representa a janela de observação do estado atual.

Tal janela pode ser definida por vocês:

Extração de Atributos

Dada uma janela de raio r, vamos gerar uma array de tamanho $(2r+1)^2$ contendo a informação de cada bloco de pixels.

Supondo as coordenadas x, y do Mario na fase do jogo, precisamos mapear os pixels de $\Delta x = x - 16r$ até $\Delta x = x + 16r + 1$ na posição horizontal e $\Delta y = y - 16r$ até $\Delta y = y + 16r + 1$ na posição vertical, com passo de 16 em 16.

Extração de Atributos

A posição do Mario dentro da fase atual está contida nos endereçoes $0x94$ para a posição x e $0x96$ para a posição y. Essas informações estão armazenadas em dois bytes no formato *little endian*, ou seja, o byte $0x94$ é a parte menos significativa e $0x95$ a mais significativa.

Para construir esses valores devemos fazer:

```
marioX = ram[0x95]*256 + ram[0x94]
marioY = ram[0x97]*256 + ram[0x96]
```

Multiplicar por 256 é equivalente a deslocar 8 bits para a esquerda.

Extração de Atributos

Para cada $x + \Delta x, y + \Delta y$, devemos determinar se contém um bloco / chão que o Mario pode andar ou colidir.

Essa informação está na posição $0x1C800$ da memória para todos os pixels, e para obter precisamos fazer:

```
x = np.floor(dx/16)
y = np.floor(dy/16)
ram[0x1C800 + np.int(np.floor(x/16)*432 + y*16 + x%16)]
```

Se o valor for 1 então existe um bloco nessa posição. Tal informação foi obtida em https://www.smwcentral.net/?p=viewthreadt=78887

Extração de Atributos

Para determinar se uma determinada posição contém um inimigo, devemos verificar os *sprites* ativos.

Um *sprite* é um bloco de desenho de um personagem, item ou qualquer objeto animado e com caixa de colisão.

No SMW é permitido até 22 sprites ao mesmo tempo na tela. Cada sprite possui informações de posição, tipo, tamanho espalhados na memória RAM.

Extração de Atributos

O primeiro passo é extrair as informações de cada sprite, a posição de memória $0x14C8 + i$ indica se o i-ésimo sprite está sendo utilizado e o status dele. Sendo 0 a não utilização desse slot.

A posição x, y do sprite está dividida em regiões distintas da memória:

```
spriteX    = ram[0xE4+i] + ram[0x14E0+i]*256
spriteY    = ram[0xD8+i] + ram[0x14D4+i]*256
```

UFABC

Extração de Atributos

O id do sprite está na posição $0x15EA + i$, o valor 44 representa um cogumelo e 216 um bloco com item.

Caso queiram complementar essas informações, basta imprimir o valor dessa posição de memória no momento que o sprite de interesse aparecer na tela.

Extração de Atributos

Na posição $0x0420 + id$ podemos determinar o tamanho desse sprite.

Um valor 0 indica que o sprite ocupa um tamanho $4x4$ na nossa matriz.

Extração de Atributos

Após coletar as informações de cada sprite ativo, comparamos para cada posição $x + \Delta x, y + \Delta y$ se um dos sprites ativos está dentro dos pixels que representa nosso bloco de informação da matriz de tamanho 16×16.

UFABC

Instalação do Sistema

Para quem quer se aventurar

Procedimento testado no Ubuntu 16.04. Instale os seguintes pacotes:

libsdl1.2-dev libsdl-gfx1.2-dev libsdl-image1.2-dev cmake git

Para quem quer se aventurar

Faça o download dos arquivos em:

http://professor.ufabc.edu.br/ folivetti/AM/

e descompacte-os.

Para quem quer se aventurar

Na pasta *Retro-Learning-Environment-master* execute o comando:

pip install .

cruze os dedos e aguarde. Se o procedimento for bem sucedido, basta entrar na pasta *rle* e testar os algoritmos que já estão prontos.

Para quem não quer se aborrecer

Instale o VirtualBox 5.2 (para usuários do Ubuntu, tenha certeza que é o 5.2).

Faça o download do arquivo SuperMarioWorld.ova

Importe o arquivo no VirtualBox em *File* $\rightarrow$ *Import Appliance*.

Arquivos fornecidos

Na pasta *rle*, temos os seguintes arquivos:

- *rominfo.py*: funções para extrair atributos da memória RAM do jogo.
- *utils.py*: funções utilitárias para os agentes baseado em Regras e Q-Learning.
- *marioQLearning.py*: arquivo contendo a implementação do Q-Learning para esse jogo.
- *play.py*: programa para mostrar uma determinada política gerada pelo Q-Learning.
- *Q*.pkl*: políticas aprendidas em alguns milhares de iterações.

Projeto Final

Projeto

O objetivo do projeto final é a implementação de um algoritmo de aprendizado por reforço.

Pode ser um dos algoritmos dados em sala de aula ou qualquer outro da literatura.

NÃO pode ser baseado em regras ou derivações desse e nem utilizar o código de Q-Learning fornecido.

Projeto

- Grupos de até 4 integrantes.
- Avaliação se dará pela distância total percorrida pelo agente.
- Estilo competição, o melhor grupo receberá a maior nota.
- Os grupos seguintes receberão nota proporcional a distância do melhor.

Projeto

Baseline: 1890 (arquivo Qbom0.pkl).

Agentes abaixo do baseline receberão 0 ou 1 ponto.

Acima disso, os grupos receberão de 2 a 3 pontos definidos pela equação:

$$ptos(x) = 2 + \frac{x - 1890}{melhor - 1890} \tag{1}$$

Se todos obtiverem uma pontuação próxima a 1890, todos receberão 1,5 pts.

UFABC

Entrega

- Até 03/05/2018
- Arquivo .zip contendo os códigos desenvolvidos, instrução de como executar o agente gerado e nomes dos componentes do grupo.
- Estudem o código do Q-Learning para aprender como salvar o estado atual do seu agente e recuperá-lo.

Próxima Aula

Na próxima aula aprenderemos sobre os algoritmos:

- Q-Learning
- Sarsa
- Ant Colony Optimization

UFABC