UNIVERSIDADE FEDERAL DE SANTA CATARINA

CAMPUS FLORIANÓPOLIS

CENTRO TECNOLÓGICO – CTC

MESTRADO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

DISCIPLINA: APRENDIZADO DE MÁQUINA

DOCENTE: JOMI FRED HUBNER

THAYS DA CRUZ FRANCO

WILLIAN DO NASCIMENTO FINATO BENOSKI

DESENVOLVIMENTO DE AGENTE POR MEIO DE APRENDIZADO POR REFORÇO UTILIZANDO Q-LEARNING

FLORIANÓPOLIS

2022

1 INTRODUÇÃO

Atualmente há uma crescente na automatização de processos e aperfeiçoamento de aplicações de métodos computacionais em resoluções de problemas. Há uma área específica da engenharia e computação conhecida como aprendizagem de máquina, onde objetiva o aprendizado da máquina sem necessariamente estar programada para desempenhar tal função. Ainda assim, há subáreas como aprendizagem supervisionada, não-supervisionada e por reforço.

Para este trabalho foi contemplada a abordagem de aprendizado por reforço, sendo escolhida a aplicação para o jogo Pacman, onde o jogador controla uma cabeça com uma boca que repete os movimentos de abre e fecha preso em um labirinto com prêmios e fantasmas, o objetivo é recolher os prêmios sem ser alcançado pelos fantasmas. Neste contexto foi criado um agente, um ambiente, as ações, funções de recompensa e função de transição para que através de tentativa e erro, possa ser criada uma tabela Q e após um período de treino, o agente possa escolher qual decisão tomar.

São apresentadas também as alterações decorrentes das variações de parâmetros, a taxa de sucesso para diferentes tamanhos do problema e observações relevantes para o desempenho do agente quanto à técnica utilizada.

2 APRENDIZADO POR REFORÇO

De acordo com Mitchell (2017) na aprendizagem por reforço o agente escolhe as ações para alterar o ambiente em que está inserido, através do treinamento é possível reconhecer quais ações fornecem recompensas positivas ou negativas de acordo com o objetivo principal, ou seja, através de treinamento de modelos, é gerada uma sequência de decisões para um objetivo em um ambiente complexo e incerto

Como principais características do método, temos a recompensa atrasada, ou seja, o método só provê qual caminho é o melhor após já ter sido testado; exploração, onde o agente explora todas as possíveis ações, experimentando para verificar qual produz um aprendizado mais efetivo. observação parcial dos estados, verificando o histórico produzido; e aprendizado a longo prazo, quanto maior o tempo de treinamento, maior a probabilidade de um resultado efetivo.

A aprendizagem por reforço é fortemente aplicada em jogos e robótica, e tem como desafio a preparação do ambiente de simulação, transferência do modelo do ambiente para o mundo real e também atingir um ótimo local, ou seja, resultados satisfatórios.

Portanto, este método é aplicável para situações em que o programador não consegue prever ou mensurar todas os possíveis estados futuros resultantes, ou seja, um sistema incerto e complexo.

3 PROBLEMA PROPOSTO

O problema consiste na implementação de um agente por meio de aprendizado por reforço implementado por meio da técnica de *q-learning*. Esse agente será implementado por meio da linguagem python utilizando a IDE pycharm.

3.1 Pac-man

Pac-man é um jogo eletrônico criado em 1980 para arcades, desenvolvido pela Namco e distribuído pela Midway Games. O jogo se tornou um clássico quando se referenciam a vídeo games e arcades.

O jogo consiste de um mapa fechado de forma similar a um labirinto com caminhos fechados, dentro deste mapa existe um personagem jogável chamado pac-man, que é uma cabeça redonda amarelada com boca e olhos, e personagens inimigos que são fantasmas coloridos que transitam pelo mapa de forma a perseguir o jogador.

O jogo tem por objetivo consumir as bolinhas todas a bolinhas brancas presentes no mapa para assim avançar para o próximo nível. Além disso, para defesa do jogador existem bolas brancas maiores que se consumidas permitem ao jogador consumir os fantasmas e se alimentar deles aumentando a pontuação, por fim ainda existem as frutas que aparecem durante o progresso do jogo que se coletadas aumentam a pontuação, elas surgem de forma aleatória durante o progresso.

3.2 Modelagem do problema

Para que seja possível avaliar o problema de forma a implementar o agente é necessário formaliza-lo de acordo com o processo decisório de Markov (MDP). É uma abordagem possível que utiliza estatística e métodos de programação dinâmica, ou seja, com atualização constante de parâmetros e com memória de dados antigos, assim estimando o melhor resultado dado parâmetros do ambiente atual.

Para a formalização do problema precisa-se definir 4 conjuntos de dados que são fundamentais para a composição da MDP e do algoritmo de decisão, sendo eles: o conjunto de estados do ambiente, o conjunto de ações, a função que determina a recompensa e a função que representa as transições, esses conjuntos tem as seguintes funções:

- Conjunto de estados do ambiente (S): Esse conjunto de estados são todos os espaços possíveis que o personagem pode ou não acessar. Pode ser definido como todos o mapa do problema.
- Conjunto de ações do ambiente (A): Esse conjunto é composto das ações possíveis em cada estado do ambiente. Se naquele espaço a ação não é possível de ser realizada o agente é recompensado de forma extremamente negativa e a ação não é completada.
- Funções de recompensa: é uma função que determina de forma imediata a recompensa que será utilizada posteriormente para calcular a ação a ser tomada.
- Funções de transição: essa função é composta pela probabilidade de o ambiente mudar de um estado para o outro caso a ação seja executada.

Para o nosso problema do pac-man pode-se definir seus dados iniciais, contudo deve-se levar em conta o mapa que será utilizado, já que são definidos uma série de layouts distintos para o funcionamento do jogo. Para o *q-learning* vamos utilizar o menor dos mapas para realizar o treinamento, o que será justificado nas sessões posteriores.

Esse layout, nomeado de *smallgrid* consiste de uma área 5x5 envolto por uma parede e com apenas dois pontos a serem consumidos e uma estrutura em formato de C interno ao mapa. Cada posição na grade é definida por um vetor de duas posições iniciados em [0,0] e indo até [5,5], todas as posições são mapeadas até aquelas que contém paredes, o que pode ser observado na figura 1. Esses dados são importantes de serem avaliados pois definem nosso espaço na grade e compõem nosso estado do ambiente.

Em cada posição pode-se realizar 5 ações distintas de acordo com a documentação do programa disponibilizada pela universidade de Berkley que é a desenvolvedora do programa. As ações definidas são: movimento para a esquerda (L), movimento para a direita (R), movimento para cima (U), movimento para baixo (D) e por fim permanecer estático no local.

Para as recompensas definidas para o agente são definidos os seguintes valores, a cada ciclo de iteração do jogo, se o agente realizar qualquer uma das ações descritas acima ele recebe uma recompensa negativa de -1, se ele tentar ir para uma parede conta como se a ação fosse de ficar estático e por consequência também recebe uma recompensa negativa de -1. Para as ações de consumir as bolinhas pequenas e grandes, o consumo delas indica uma recompensa de 10, e consumir os fantasmas dá uma recompensa de 250, para a completude do mapa quando todas as bolinhas são consumidas o jogo recompensa o agente com 500 pontos.

Por fim para as funções de transição de estados a definição é dada por duas probabilidades dentro do programa. A primeira delas é nomeada como girar da moeda, no programa se essa probabilidade definida for menor que o valor de épsilon do programa então será utilizada uma ação aleatória a ser selecionada, caso contrário serão definidas as ações de acordo com a tabela formada pelos valores de Q e selecionada de acordo com o melhor valor da tabela Q para as ações disponíveis de realizar a ação da tabela e qual realizar, caso os valores da tabela sejam iguais para várias ações ele seleciona por base em probabilidade randômica.

Figura 1 – Layout de small grid

3.3 Programação do agente

O processo de programação do agente consiste de uma série de funções que foram pré-definidas pela própria universidade de Berkley, e sua solução foi baseada no programa desenvolvido pelo usuário srinadhu que postou seu código no github para ser avaliado. Seguindo o programa disponível pela universidade de Berkley existem 5 funções pré-definidas que devem ser preenchidas pelo usuário com seu código próprio, sendo elas: computeValueFromQValues, computeActionFromQValues, getQValue, getAction e update.

Para o programa do agente existem outras funções que são importantes de serem avaliadas para o entendimento do programa, então visualizando o código iniciamos pela função ___init___, que tem o seguinte código:

```
    def __init__(self, **args):

        ReinforcementAgent.__init__(self, **args)

        self.qValues = util.Counter()
```

Essa função inicializa os valores da tabela Q baseando-se no mapa escolhido e forma nossos valores iniciais da tabela que são definidos como 0. Para agora poder recolher os valores dessa tabela e utiliza-las desenvolvemos a função getQValue por meio do seguinte código:

```
def getQValue(self, state, action):

    return self.qValues[(state, action)]
```

Para essa função temos que ele acessa a tabela Q e recupera os dados daquela célula em especifico, levando em conta o estado em que se localiza e a ação a ser realizada. Em seguida definimos a função computeValueFromQValues,q eu se dá da seguinte forma:

```
    def computeValueFromQValues(self, state):

        legalActions = self.getLegalActions(state)
        bestQVal = -float('inf')

        if len(legalActions)==0:
            return 0

        for currentAction in legalActions:
            qValue = self.getQValue(state, currentAction)
            bestQVal = max(bestQVal, qValue)

        return bestQVal
```

Essa função tem por característica definir e computar qual são os valores da tabela Q para as ações possíveis de serem realizadas naquele local. Nesse caso, a primeira parte da função avalia quais são as possíveis ações a serem tomadas no local em que o agente está posicionado, se não houverem ações disponíveis no local então retorna o valor de 0 e, portanto, significa que o agente não pode mais se mexer e o jogo acaba. Se houverem ações legais ele avalia as possibilidades e compara o valor de cada valor da tabela Q disponível com um valor inicial de -infinito, assim selecionando qual o valor de Q para cada movimento.

A próxima função é a computeActionFromQValues:

```
def computeActionFromQValues(self, state):

    legalActions = self.getLegalActions(state)
    bestQVal = -float('inf')

    if len(legalActions)==0:
        return None


    for currentAction in legalActions:
        qValue = self.getQValue(state, currentAction)
        if qValue>bestQVal:
            bestQVal = qValue
            action = currentAction

    return action
```

Essa função consiste no cálculo de decisão para a melhor ação a ser tomada em determinado estado atual. Inicialmente ela procura todas as ações legais disponíveis naquela posição similar a função anterior, em seguida escolhe o melhor valor de Q atualizado pela função anterior, assim deixando já pré-calculado todos os valores de saída possíveis.

Para a próxima função definimos o getAction:

```
def getAction(self, state):


    legalActions = self.getLegalActions(state)
    action = None

    if len(legalActions)==0:
        return action

    if util.flipCoin(self.epsilon):
        action = random.choice(legalActions)
    else:
        action = self.computeActionFromQValues(state)


    return action
```

Para essa função definimos quais ações podem ser tomadas de acordo com a possibilidade, e fazemos a comparação por meio da função aleatória flipcoin, se esse valor for menor que o épsilon definido então a ação tomada será randômica, caso contrário serão computados os valores das ações e suas recompensas e assim decidida qual será a próxima ação a ser tomada.

Por fim, define-se a função update

```
def update(self, state, action, nextState, reward):

        self.qValues[(state, action)] = self.getQValue(state, action) \
                                    + self.alpha \
                                    * (reward + self.discount
                                              * self.getValue(nextState)\
                                              - self.getQValue(state,
action))
```

Essa função realiza a atualização dos valores da tabela Q de acordo com a equação definida para esse problema que consiste em:

$$Q^*(s,a) <- Q(s,a) + \alpha[r + \gamma * \max(Q(s',t') - Q(s,a))] \quad (1)$$

Essa função tem por definição que a atribuição do novo valor de Q depende do valor de Q atual somado a taxa de aprendizado do algoritmo, que influencia direto a recompensa e os valores do desconto, comparando o valor posterior e atual da tabela Q.

4 RESULTADOS OBTIDOS

4.1 Variação dos parâmetros (alpha, gama e epsilon)

Inicialmente deve-se avaliar as variações dos parâmetros do *Q-learning* para o processo de decisão do nosso agente. Mas antes de detalhar a escolha dos valores é importante entender a influência de cada um deles para esse processo decisório.

Para os parâmetros alpha, ou a chamada taxa de aprendizado é um valor que indica o quanto o seu agente está aprendendo do ambiente em que ele é submetido, esse valor é relativamente complexo de ser avaliado de forma decisiva tendo em vista que se o valor for zero o agente não tem a capacidade de aprender, então não importa se o agente receberá recompensas positivas ou negativas, todos esses valores são desconsiderados e o agente se move de forma totalmente randômica. Isso impacta negativamente o processo do jogo do pacman a aleatoriedade do movimento é incoerente devido a necessidade de desviar dos fantasmas existentes no mapa, então um movimento aleatório não deve ser utilizado.

Contudo, se esse valor for 1 o agente aprenderá o problema de forma especifica incapaz assim de generalizar e apenas podendo ser aplicado ao ambiente determinístico ao qual foi submetido. Essa situação também é indesejável, se o pacman fosse um jogo sem inimigos apenas coletando as bolinhas esse modelo até poderia ser aplicado para cada mapa individualmente e assim ele coletaria todas as bolinhas. Contudo, com a existência de inimigos que se movimentam de acordo com seu próprio programa não é possível aprender uma movimentação especifica para o mapa precisando assim se adaptar.

Para o menor mapa citado nesse trabalho o valor ideal para alpha se aproxima de 0.2, esse valor após uma série de testes se mostrou o mais satisfatório devido a velocidade com que auxilia o nosso agente a tomar as decisões corretas com menor tempo de treinamento. O treinamento do nosso agente consiste de 2000 vezes do jogo sendo jogado, dessa forma, em 1400 jogos o agente já consegue aprender uma política ótima de decisão evitando o fantasma com esse parâmetro. A alteração desse valor para mais próximo de 0 torna o agente mais randômico e o treino se tornou ineficiente, o agente não conseguiu manter uma consistência no

seu comportamento, apesar de vitorioso em muitos momentos o agente ficava parado ao invés de ir atrás das bolinhas de forma direta e o treino demorava mais, contudo, mesmo com maior necessidade de treino o agente ainda conseguiu aprender uma política ótima.

Para os valores maiores de alpha o programa se tornou especifico demais aprendeu o comportamento do mapa, mas desconsiderava a presença do movimento do fantasma, o agente começou a realizar as mesmas ações independendo do movimento do inimigo então pela quantidade de testes realizado com o agente ele não era consistente nas vitórias. Em suma, valores pequenos de gama fizeram com que nosso agente se tornasse aleatório e demorasse mais para encontrar a politica ótima, valores acima de 0.2 fazem com que nosso agente comece a se tornar especifico demais, se tornando inconsistente na quantidade de vitórias. Vale ressaltar que por via de regras a faixa entre 0.1 à 0.5 tiveram comportamento de encontrar a politica ótima, contudo, com tempos de treinamento diferentes. Uma comparação entre as diferentes taxas de aprendizagem pode ser observada na tabela 1.

Tabela 1 – Comparação entre taxas de aprendizagem

Taxa de aprendizagem	Tempo de treino p/ 1ª vitória	Vitória/Derrota
0	1000	5/10
0.2	1400	10/10
0.5	1800	10/10
1	1200	6/10

Fonte: Desenvolvida pelos autores (2022)

Já para o parâmetro Gamma ou chamado de taxa de desconto é uma taxa que indica a importância das recompensas atuais e futuras para o agente. Isso pode ser explicado da seguinte forma, para jogos com a presença de inimigos as recompensas futuras podem estar próximas ao inimigo aumentando a chance do agente encontra-lo, isso faz com que em muitos casos a recompensa futura não seja tão importante quanto a recompensa atual. A taxa de desconto indica o percentual de quanto a recompensa futura vai ser menos valiosa que a recompensa atual.

Para o nosso caso do pacman esse valor deve ser mantido alto, devido a necessidade do pacman coletar todas as bolinhas do jogo para poder vencer. Contudo, esse valor não pode ser um devido a movimentação dos fantasmas que deve ser levada em conta. Se o valor for um o agente estará tão focado apenas nas recompensas, devido a não haver diferença para ele entre a

valia da recompensa futura em relação a atual que na maior parte dos casos ignorará a existência dos inimigos e será imprudente. Já caso o valor for zero o agente não vai considerar as recompensas futuras e apenas vai focar nas atuais, isso para o jogo indica que nosso agente não tenderá a ir atrás das recompensas.

Para os valores avaliados a taxa de desconto mais eficiente se mostrou um valor de 0.8 tornando o treino do nosso agente eficiente e equilibrado em relação as recompensas atuais e futuras. O agente leva em consideração a presença dos fantasmas e também leva em consideração que ainda existem recompensas a serem coletadas.

Tabela 2 – Comparação entre taxas de desconto

Taxa de aprendizagem	Tempo de treino p/ 1ª vitória	Vitória/Derrota
0	2000	2/10
0.4	1300	8/10
0.8	1000	10/10
1	1500	9/10

Fonte: Desenvolvida pelos autores (2022)

Já para o épsilon ou taxa de exploração, ele é um parâmetro necessário para o agente poder se manter explorando dados desconhecidos do ambiente. Como o agente precisa passar por um treinamento, ele não pode apenas explorar locais já conhecidos devido a necessidade de absorver mais informação. Como nossa tabela do Q-learning é completada dinamicamente uma taxa de exploração baixa ou nula fará com que nosso agente apenas ir para locais conhecidos e ficará focado em um máximo local e não global. No caso a taxa de exploração adiciona um fator randômico ao nosso agente que em alguns casos mesmo com o valor da tabela do Q alto, ele pode selecionar ir pra outro lugar conhecer o ambiente a sua volta. A ideia é que quanto mais familiarizado com o ambiente o agente estiver essa taxa deve decair com o tempo, se tornando zero a longo prazo.

Para o nosso problema do Pac-man no mapa pequeno esse valor surtiu pouca diferença no comportamento do agente, como o caminho é pequeno as taxas mais baixas dão conta de resolver o problema e até mesmo com o valor zero o agente conseguiu descobrir a política ótima, contudo, com o valor aumentando o agente começou a ter certa dificuldade devido a sempre querer explorar mais locais desconhecidos e tomando ações randômicas, assim

dificultando na descoberta da politica ótima de ação. Eventualmente o agente consegue chegar na política, mas com maior dificuldade.

Tabela 3 – Comparação entre taxas de desconto

Taxa de aprendizagem	Tempo de treino p/ 1ª vitória	Vitória/Derrota
0	1100	10/10
0.05	1000	10/10
0.4	1200	10/10
1	1800	9/10

Fonte: Desenvolvida pelos autores (2022)

Contudo, quando utilizados estes valores para um mapa maior do pac-man independente dos valores tomados e das combinações avaliadas, em geral é muito difícil se não impossível encontrar uma política ótima por meio do Q-learning simples, em geral o agente não consegue generalizar mapas maiores, pois, não consegue identificar que o fantasma é uma recompensa ruim em todas as posições avaliadas. Isso se dá, pois, ele teria que ser capaz de encontrar o fantasma em praticamente todas os estados possíveis do ambiente o que levaria muito tempo e devido ao seu início randômico muitas vezes não consegue nem alcançar todos os estados.

Devido a alta quantidade de mortes quando avaliamos pelo teste o agente, é possível observar que para a combinação ideal dos valores das taxas alpha = 0.2, Gamma = 0.8 e Epsilon 0.05 o agente tende a ficar em apenas um lado do mapa circulando apenas aquela área e inevitavelmente acertando um fantasma a longo prazo. Esse comportamento é esperado devido a incapacidade de explorar todo o ambiente. Em geral, levaria um tempo de treino muito alto para que fosse possível que randomicamente o agente observasse todos os estados e que em todos eles um fantasma poderia existir.

Esse problema já se torna presente apenas na versão um pouco maior do mapa que mesmo com a variação não foi possível chegar a valores satisfatórios. O mapa chamado de smallClassic pode ser observado na figura 2. Nesse mapa temos uma grid de 12x5 o que resulta em 60 estados existentes e em que podem ser tomadas 5 ações em cada estado, o que resultaria numa tabela de Q-learning com 600 posições possíveis. Para que isso funcionasse o agente deveria ser capaz de durante o treino acertar todas as posições em que o fantasma existiria e identificar que aquele local tem recompensa negativa, tentando se afastar dele.

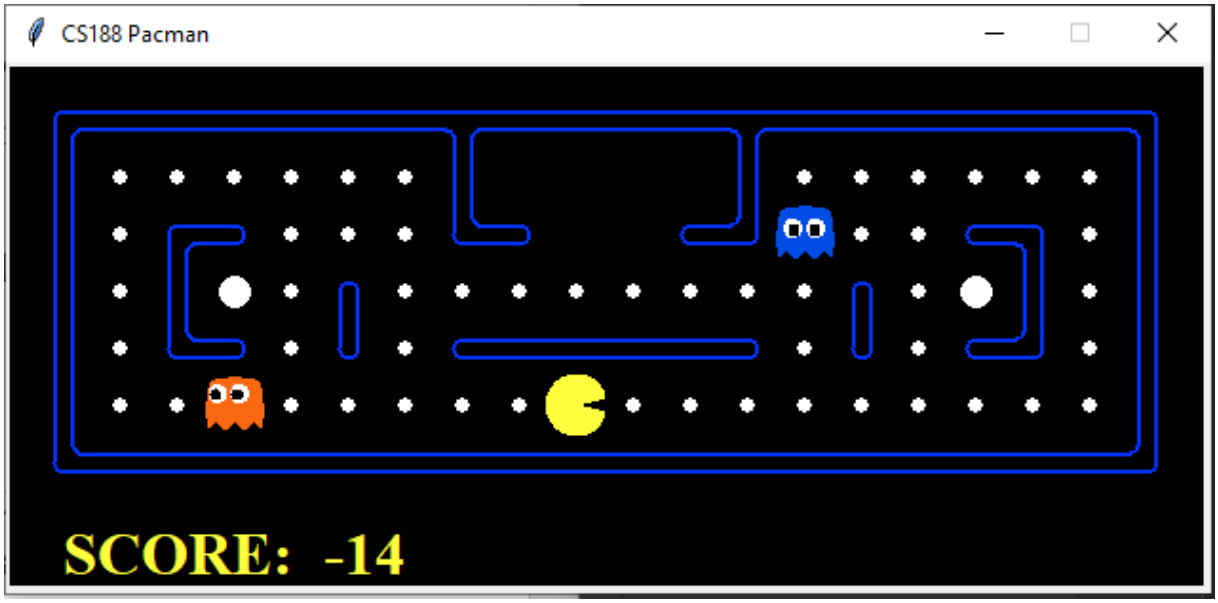

Figura 2 – Layout de smallClassic

4.2 Q-learning aproximado

Um problema em mapas grandes ou em espaços de estados com muitas possibilidades o Qlearning pode não ser suficiente como exemplificado pelos mapas anteriores. Isso pode ocorrer por uma série de fatores distintos, mas em geral, o principal problema como citado anteriormente é a dificuldade de encontrar estados com alta recompensa o que dificulta as decisões a serem tomadas pelo agente, além disso, a tabela dos valores de Q se torna muito grande e difícil de ser explorada e totalmente preenchida pelas decisões do agente, e por fim atualização dos campos dos valores de Q demoram a serem preenchidos e atualizados de forma a propagar as recompensas por novas ações.

No caso do pacman isso se torna visivelmente problemático devido a presença dos fantasmas com movimentos randômicos, a atualização da tabela Q deve levar em consideração todas as posições do mapa, todos os movimentos do agente e também se os fantasmas podem alcançar aquela posição em algum momento para preencher a tabela, o que exige uma quantidade absurda de tempo para explorar todos os pontos do espaço.

Considerando agora o mapa médio clássico do jogo que pode ser observado na figura 3, o espaço de estado desse mapa inclui 107 possibilidades de posição, a localização dos fantasmas pode ter 107^2 possibilidades, ignorando a existência das frutas que podem aparecer o jogo e todos os movimentos que o pacman pode realizar que nesse caso consideramos 4, retirando a

possibilidade de ficar para isso faz com que o espaço de estado total do problema tenha 19600688 posições na tabela. Esse número escala de forma exponencial e é inviável resolver esse problema pelo método de Qlearning convencional, justamente pelos problemas já citados.

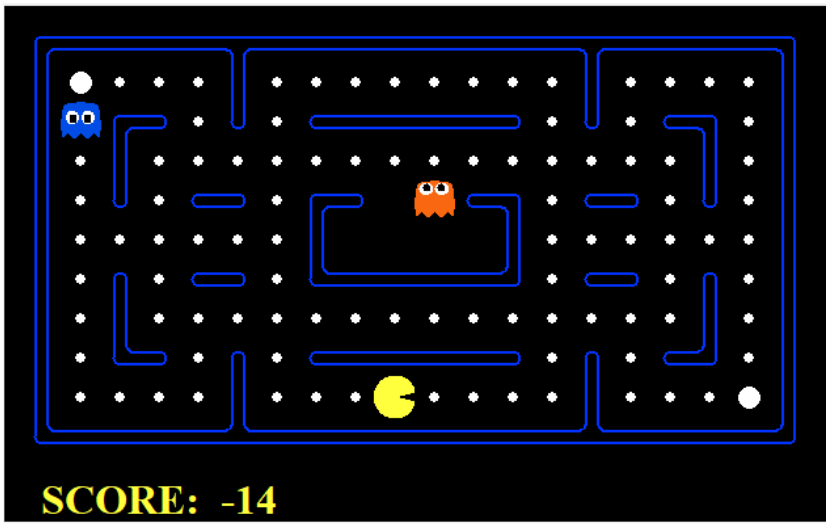

Figura 3 – Layout de mediumClassic

Para solucionar esse problema recorre-se ao Q-learning aproximado que é um método que permite que a tabela Q seja reduzida e que o processo considere mais componentes do problema original. Contudo, para chegar nessa ideia pode-se pensar em alguns métodos para a solução desse problema.

O primeiro poderia ser dar pequenas recompensas intermediárias ao problema para auxiliar no aprendizado do agente, recompensando assim se o agente tender a explorar mais o seu ambiente. Contudo, esse método exige que o designer do problema tenha um bom conhecimento do mapa previamente e do seu comportamento, e também não diminui a tabela Q, o que não auxilia tanto na procura.

Outra ideia possível é aprender as recompensas do ambiente como uma combinação linear de todos os componentes do jogo, para cada estado encontrado pelo agente sua representação leva em conta os componentes do local e atualiza o Q-learning de acordo com esse conhecimento adquirido.

Quando combinada essa técnica com o Q-learning obtemos o Q-learning aproximado, onde a tabela Q é atualizada com a junção das componentes ativas para aquele estado. Esse comportamento é definido pela seguinte equação:

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + w_n f_n(s,a) \qquad (2)$$

Assim, é adicionado um peso w a cada uma das possíveis componentes do sistema e como ela influencia o comportamento do agente e seu resultado final. Para isso os pesos são atualizados baseados nas componentes da seguinte forma:

$$w_i = w_i + \alpha\left[r + \gamma * \max\left(Q(s',t') - Q(s,a)\right)\right] f_i(s,a) \qquad (2)$$

Isso faz com que se alguma ação negativa aconteça o agente correlacione as componentes que influenciaram e seus pesos e as culpe, assim evitando que o comportamento seja repetido ao longo do treinamento. Isso pode ser considerado como uma forma de aproximação por meio do método de regressão linear dos valores. De forma positiva conseguimos assimilar mais comportamentos e correlaciona-los, enquanto de forma negativa muitas componentes iguais podem ter resultado diferentes quando combinadas, exigindo um treino maior.

A implementação esse algoritmo foi baseado no algoritmo compartilhado pelo autor srinadhu na plataforma Github. Nesse método as recompensas de cada um dos componentes são definidas de forma similar ao anterior, onde o consumo de bolinhas conta 10 pontos de recompensa, consumir fantasmas retorna 250 pontos, vencer o jogo retorna 500 pontos e ficar parado ou se movimentar retorna -1 ponto.

São definidas três funções para o cálculo dos pesos e atualização dos dados, sendo elas getWeights, getQValue e Update. A função getWeight apenas retorna o valor dos pesos associados a cada um desses componentes definidos previamente e seu comportamento. A função getQValue retorna qual o valor de Q para aquele estado e ação, baseado no peso e na componente que está sendo solicitada.

```
def getQValue(self, state, action):

    feat = self.featExtractor
    features = feat.getFeatures(state, action)
    qv = 0
    for feature in features.keys():
        qv += self.weights[feature] * features[feature]
    return qv
```

Na primeira parte desse código recupera-se a componente f que será utilizada, associando-a ao estado e ação. O valor de q é zerado naquele local e o novo valor adicionado ao local é o peso em relação a componentes utilizada multiplicado pela componente. Isso é feito até que todas as componentes associadas ao estado e ação sejam levadas em conta e somadas para atualizar o peso considerando tudo.

Para a função update a ideia então é atualizar os valores do peso baseado nas transições, para isso esse código verifica quais são as ações legais para o local, identifica o valor inicial do q que seria menos infinito para que o agente possa procurar um novo estado, ele então para cada ação disponível ele atualiza os valores do maior valor de Q e calcula o valor de cada componente associada. Por fim, com o resultado dessa componente e tendo a diferença dos valores, ele atualiza os pesos para o estado e toma sua ação. Isso é feito por meio da função abaixo:

```
def update(self, state, action, nextState, reward):

    actionsFromNextState = self.getLegalActions(nextState)
    maxqnext = -999999
    for act in actionsFromNextState:
        if self.getQValue(nextState, act) > maxqnext:
            maxqnext = self.getQValue(nextState, act)
    if maxqnext == -999999:
        maxqnext = 0
    diff = (reward + (self.discount * maxqnext)) - self.getQValue(state,
action)
    features = self.featExtractor.getFeatures(state, action)
    self.qvalue[(state, action)] += self.alpha * diff
    for feature in features.keys():
        self.weights[feature] += self.alpha * diff * features[feature]
```

Com esse código de atualização dos pesos, pode-se agora testar o comportamento do Pac-man de acordo com essa nova proposta. Para isso a universidade de Berkley indicou o uso de sua função de featuraExtractor, que é responsável por recuperar informações importantes para o agente usar, como, se a comida será ou não consumida, o quão longe a próxima bolinha se encontra, se a colisão com o fantasma é iminente e se o fantasma está um espaço próximo do agente.

Com esses dados e os pesos o agente pode finalmente identificar o comportamento e como ele deve reagir ao mapa maior e a presença de fantasmas. Vale salientar que os dados iniciais para esse problema foram mantidos, com as taxas alpha em 0.2, gamma em 0.8 e épsilon em 0.05 que são taxas importantes para o funcionamento do método. Agora considerando essas atualizações é possível que o nosso agente acesse todo o mapa em apenas 50 fases de treinamento, sendo bem sucedido em 10 testes consecutivos. Isso para qualquer um dos mapas,

contudo leva-se em conta agora que o treinamento é mais demorado levando minutos até sua conclusão.

Para o mapa médio por exemplo, o treinamento de 50 dados demorou cerca de 2 minutos de treinamento continuo para identificar o problema por completo e conseguir achar a politica ótima de ação. Quando avaliado o teste do agente foi observado que o agente tende a evitar os fantasmas a todo o custo, mesmo que ele possa consumi-los devido ao consumo da bola maior. Isso mostra que o agente entende que chegar próximo aos fantasmas é sempre negativo mesmo que os consumir aumente sua pontuação final. Dessa forma ele desvia dos fantasmas de forma satisfatória indo atrás do resultado final.

Para os mapas maiores o treinamento do sistema demorou cerca de 5 minutos e tem uma quantidade baixa de derrotas em relação as vitórias. O agente ainda consegue na maioria das vezes completar o problema sem perder com apenas 50 treinos. O aumento do número de treinamento fez com que o agente ficasse ainda melhor errando menos durante os testes e sendo mais vitorioso.

Tabela 4 – Comparação entre taxas de vitória por mapa

Tipo de mapa	Qte de treino	Vitória/Derrota
smallClassic	20	8/10
	40	10/10
	50	10/10
mediumClassic	20	3/10
	40	5/10
	50	8/10
originalClassic	20	1/10
	50	5/10
	70	8/10

Fonte: Desenvolvida pelos autores (2022)

5 CONCLUSÕES

Através do estudo realizado foi possível concluir que a taxa de aprendizado interfere significativamente no desempenho do agente, se igual a zero, gera ações aleatórias, se igual a um, torna o ambiente muito especifico. Para a fase de treino de 2000 partidas, com 1400 o agente já encontra a política ótima de decisão, quando próxima a 0 o agente não buscava o objetivo, quando próxima a 1 ignorava o inimigo portanto, o parâmetro alpha influencia significativamente no desempenho.

Quanto à taxa de desconto, quando muito baixa o agente ignora as recompensas futuras, e quando muito alta não diferencia recompensas futuras das atuais. Já a taxa de exploração deve ser decrescida com a experimentação, dado que o ambiente se torna conhecido a medida em que é explorado, para valores altos este processo é mais devagar devido a necessidade de explorar os ambientes desconhecidos Ainda assim, é possível concluir que este tipo de método pode ser utilizado para mapas pequenos pois para mapas grandes o tempo de processamento é muito elevado pelas inúmeras possibilidades de ações e estados, para estes casos é recomendado o uso de q-learning aproximado e redes neurais.

Desta forma, é possível concluir que o método de aprendizado por reforço obteve sucesso no desempenho do agente para o objetivo de jogar e ganhar partidas do jogo pacman. Podendo ser utilizado em diversas outras situações onde não é possível descrever ou enumerar todos os estados que possam ser gerados.

REFERÊNCIAS

MITCHEL, T.M. Machine Learning. McGraw Hill Education. 1 ed. India. 2017

Project 3: Reinforcement Learning. UC BERKELEY, 2014. Disponível em: < http://ai.berkeley.edu/reinforcement.html>

RL_Pacman, GITHUB, 2018. Disponível em: <https://github.com/srinadhu/RL_Pacman>