Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Tiago Negrisoli de Oliveira

MONOGRAFIA DE PROJETO ORIENTADO EM COMPUTAÇÃO II

Coordenação em sistemas multiagente em tempo real

Belo Horizonte
2020/ 1º semestre

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Coordenação em sistemas multiagente em tempo real

por

Tiago Negrisoli de Oliveira

Monografia Projeto Orientado em Computação II

Apresentado como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFMG

Prof. Dr. *Luiz Chaimowicz*
Orientador
Prof. Dr. *Anderson Rocha Tavares*
Co-orientador

Belo Horizonte
2020/ 1º semestre

Aos familiares,
aos amigos,
aos professores,
aos colegas de trabalho,
dedico este trabalho.

AGRADECIMENTOS

Inicialmente quero agradeço aos meus familiares, pelo amor e suporte.
Agradeço aos amigos, pelo companheirismo.
E finalmente aos professores, pelos conhecimentos e oportunidades.

1 RESUMO

Sistemas multiagentes (SMA) são ambientes em que vários agentes estão interagindo. Os objetivos desses podem ser os mesmos, assim tendo uma relação de cooperação, ou divergentes em que eles competem. Jogos estratégia em tempo real (RTS) podem ser vistos como SMAs e possuem um ambiente dinâmico com informações parciais do oponente e do mapa. Além disso, requerem dos jogadores gerenciamento da economia e recursos, criação e movimentação de unidades, construção de edifícios, aprimoramento de tecnologias e controle de embates contra oponentes. Essas características tornam esse tipo de ambiente complexo, em que o agente tem que lidar com um grande espaço de estados e de ações, tempo de raciocínio limitado e recompensas em longo prazo. O trabalho foca na parte de combate desse estilo de jogo, em que dois exércitos compostos por múltiplas unidades lutam. Dessa forma, foram propostas quatro formas de se lidar com esse cenário. (I) Aprendizado independente em que agentes não consideram outras. (II) Friend-or-Foe, método de coordenação que aprende com a pressuposição de que certas unidades são aliadas e outras são inimigas. (III) Correlated-Q, método de coordenação que aprende a partir do equilíbrio correlacionado dos agentes. (IV) Deep Q-Network (DQN) que é um método que utiliza uma rede neural para aproximar valores Q para as ações das unidades.

Palavras-chave: *Jogos de estratégia em tempo real, RTS, Inteligência artificial, aprendizado por reforço, coordenação, Sistemas multiagentes, Deep Q-Learning.*

2 ABSTRACT

Multi Agent systems (MAS) are ambients that multiple agents are interacting with. They can have either similar goals, needing to cooperate with each other, or opposite goals, where they compete. Real-Time Strategy games (RTS) are dynamic environment with partial information of opponents and the map and can be seen as MAS. Also, requires from the player resource and economic management, unit movement and creation, building, technological upgrades and controls in combat scenarios. All of these characteristics create difficulties in applying artificial intelligence techniques, such as large number of states, limited thinking time, long-delayed rewards and combination joint-action spaces. We focus on the combat of these games, where armies composed by many units fight. We propose four models that deal with those scenarios. (I) Independent learners that doesn't consider other agents. (II) Friend-or-Foe, a coordination method that assumes some units to be allies, and other foes. (III) Correlated-Q, another coordination method that learns from the correlated equilibrium of the agents. (IV) Deep Q-Network (DQN), a method that uses a neural network to approximate the Q values of the state-action pair.

Keywords*:* *Real-time Strategy games, Artificial Intelligence, reinforcement learning, Multi agent systems, Deep reinforcement learning.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

RTS	Real-Time strategy game
RL	Reinforcement Learning
POE	Portfolio Online Evolution
PGS	Portfolio Greedy Search
SSS	Stratified Policy Search
GAB	Greedy Alpha-Beta Search
SAB	Stratified Alpha-Beta Search
DQN	Deep Reinforcement Network
SMA	Sistemas multiagente
MAS	Multi agent Systems
PDM	Processo de decisão de Markov
MDP	Markov Decision Process
FFQ	Friend-or-Foe
CREQ	Correlated Q-Learning

SUMÁRIO

1. INTRODUÇÃO

Jogos de estratégia em tempo real são ambientes dinâmicos que envolvem raciocínio em longo prazo sobre aspectos estratégicos, por exemplo controle de economia, e em curto prazo, como em situações de combates entre exércitos. Essas batalhas são pontos vitais para o sucesso nesse tipo de jogo e se tornou um tópico amplo de estudo na área de inteligência artificial. Varias formas de se lidar com esse tipo de ambiente foram propostas. A mais simples delas são algoritmos de comportamento fixo, chamados também de scripts. Esses atribuem a cada unidade do exército ações seguindo uma regra pré-definida. Dessa forma, a estratégia desse tipo de comportamento não terá alteração. Um grande problema dessa abordagem é que oponentes podem tirar proveito da invariância do comportamento e podendo explorar uma fraqueza. Assim, duas outras abordagens vêm se destacando para essa aplicação, a busca e aprendizado.

Métodos baseado em busca verificam vários estados do jogo através de uma simulação, então, usando uma métrica qualquer, avalia o valor desses estados, e assim, escolhem a sequência de ações que o leve para o melhor avaliado. Algoritmo desse estilo vem apresentando resultados muito bons, entretanto, existem algumas limitações à essa abordagem. A primeira é que é necessário não apenas uma modelagem do oponente, para conseguir simular as ações tomadas por ele como uma modelagem do ambiente, para determinar os próximos estados gerados pelas ações tomadas. Outro grande problema para a aplicação desses métodos é o tempo de processamento, visto que eles devem verificar um grande número de estados, e possivelmente estados bem distantes do tempo atual do jogo.

Aprendizado é uma área da computação que consegue descobrir, a partir de dados, um padrão que maximiza uma função chamada objetivo. Aprendizado por reforço é um sub campo dessa área que dado um ambiente definido por estados, é um modelo no qual um agente observa esse ambiente na forma de percepções e interage por meio de ações. Dessa forma, o agente em um determinado estado s recebe informações e escolhe uma ação a possível em s. Assim, essa ação muda o estado s para s’ diferente e o agente recebe uma recompensa r correspondente a

essa mudança. Esse método vem ganhando grande destaque por conseguir lidar com ambientes complexos.

Assim, o trabalho vai apresentar quatro métodos que se baseiam em aprendizado por reforço. Além disso, mostrará como esses métodos lidam com um ambiente de combate de jogos RTS, contra oponentes de comportamento fixo e outros mais robustos baseado em busca.

2. CONTEXTUALIZAÇÃO E TRABALHOS RELACIONADOS

Um problema de decisão pode ser modelado utilizando o Processo de Decisão de Markov (MDP). Um MDP é um tupla $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{R}, \boldsymbol{\gamma})$, em que $\boldsymbol{S}$ é o espaço de estados, $\boldsymbol{A}$ é o conjunto finito de ações, $\boldsymbol{T}$ é uma função de transição, $\boldsymbol{R}$ é a função de recompensa e $\boldsymbol{\gamma}$ é um escalar em que $\boldsymbol{\gamma} \boldsymbol{\epsilon} [\boldsymbol{0}, \boldsymbol{1}]$.

Estados

$\boldsymbol{S}$ é um conjunto de todos os possíveis estados do problema. Ele também é chamado de espaço de estados. É necessário que para qualquer $\boldsymbol{s} \boldsymbol{\epsilon} \boldsymbol{S}$, ele tenha a propriedade de Markov.

A propriedade de Markov diz que um $\boldsymbol{s_t}$ contêm todas as informações necessárias para modelar futuros estados sem precisar considerar outros do passado. Formalmente, temos que:

$$\boldsymbol{T}[\boldsymbol{s_{t+1}} | \, \boldsymbol{s_t}] \; = \; \boldsymbol{T}[\boldsymbol{s_{t+1}} | \, \boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_t}]$$

$\boldsymbol{T}$ é a função de transição que mapeia probabilidades de se alcançar o estado $\boldsymbol{s_{t+1}}$ a partir de $\boldsymbol{s_t}$. Assim, a probabilidade de alcançar um estado é exclusivamente relacionado apenas ao estado anterior.

Recompensa

$\boldsymbol{R}$ é a função de recompensa imediata ao executar uma ação $\boldsymbol{a}$ no estado $\boldsymbol{s}$. Dessa forma, é possível definir a função chamada de *retorno*. Ela nos diz o total de recompensa descontada que se pode obter a partir do tempo $\boldsymbol{t}$. O *retorno* é definido como:

$$\boldsymbol{Gt} \; = \sum_{\boldsymbol{k=0}}^{\infty} \boldsymbol{y^k R_{t+k+1}}$$

O escalar $\boldsymbol{\gamma}$ dita até que ponto no futuro deve-se considerar as recompensas. Com o $\boldsymbol{\gamma}$ é próximo de zero, apenas recompensas imediatas terão influência no total

do retorno. Quando $\boldsymbol{\gamma = 1}$, recompensas futuras terão o mesmo peso que recompensas imediatas.

Política

Uma política é um conjunto de regras que controla o comportamento de um agente. Formalmente, é uma distribuição de probabilidades sobre cada uma das possíveis ações do agente. Ela é dada por:

$$\boldsymbol{\pi}(\boldsymbol{a} \mid \boldsymbol{s}) = \mathrm{P}[\boldsymbol{A_t} = \boldsymbol{a} | \boldsymbol{S_t} = \boldsymbol{s}]$$

Por ser uma distribuição de probabilidades, a ação do agente é aleatória e segue a distribuição $\boldsymbol{\pi}$. Uma política determinista é um caso especial em que a probabilidade de uma das ações é um, ou seja, o agente irá sempre escolhe-la.

A política ótima é àquela que maximiza a soma das recompensas ao longo do tempo t no ambiente. Essa função pode ser aprendida com a repetida interação do agente com o ambiente baseando-se nas recompensas obtidas.

A equação de Bellman é uma formulação que calcula o valor total de um estado em um MDP. Ela é definida por:

$$V(s) = \max_a R(s,a) + \boldsymbol{\gamma} \sum_{s\prime} T(s,a,s')V(s')$$

$V(s)$ é o valor de estar no estado s. $V(s')$ é o valor de estar no próximo estado alcançado ao executar a ação a. O que ela nos diz é que o valor de um estado pode ser calculado pela recompensa imediata ao executar uma ação nele e pelos valores dos próximos estados que podem ser alcançados. Como estamos tratando de um ambiente que tem probabilidades associadas às transições, o valor é ponderado por essa probabilidade. Assim, calcula-se a ação que maximiza o valor V.

Com isso, é possível definir uma métrica para calcular o valor de uma ação $Q(s,a)$ a partir de $V(s)$. Q é uma função que calcula a recompensa total que pode ser obtida ao tomar uma ação em um determinado estado. Ela pode ser definida a partir da equação de Bellman da seguinte forma:

$$Q(s,a) = R(s,a) + \boldsymbol{\gamma} \sum_{s' \in S} T(s,a,s')V(s')$$

O valor de Q é igual à recompensa imediata esperada e pela recompensa em longo prazo descontada. É possível também definir V a partir de Q da seguinte forma:

$$V(s) = \max_{a \in A} Q(s',a)$$

Por fim, a função Q pode ser escrita de forma recursiva:

$$Q(s,a) = R(s,a) + \boldsymbol{\gamma} \max_{a' \in A} Q(s',a)$$

Q-Learning

Q-Learning é um método de aprendizado por reforço que procura uma política ótima. Ele tenta aprender os valores da função Q de forma iterativa conforme o agente interage com o ambiente. Essa atualização é dada pela equação:

$$Q(s,a) \leftarrow (1 - \alpha)Q(s,a) + \alpha[\, R(s,a) + \boldsymbol{\gamma} \max_{a} Q(s',a')]$$

Uma grande limitação para esse algoritmo é que, em sua versão tabular, ambientes com um grande espaço de estado e conjunto de ações pode se tornar incomputável de se calcular e armazenar esses valores. Uma forma para contornar esse problema é utilizar uma técnica que em vez de armazenar os valores, é criado uma função para aproxima-los. Essa variação do Q-Learning é chamada de Approximate Q-Learning. A forma de se fazer isso é extrair do ambiente um conjunto de características F que representam uma simplificação do estado s. Assim, utilizando pesos W, é aproximado o valor $Q(s,a)$ pela equação a seguir:

$$Q(s,a) = \sum_{i}^{n} W_i * F_i\,(s,a)$$

Assim, é necessário ajustar os pesos de forma a aprender uma boa aproximação de Q. A atualização dos pesos é feita da seguinte forma:

$$diff \leftarrow [R(s,a) + \boldsymbol{\gamma} \max_{a'} Q(s',a')] - Q(s,a)$$

$$W_i \leftarrow W_i + \alpha * diff * F_i(s,a)$$

Algoritmos de Coordenação

Existem duas formas de se aplicar o algoritmo Q-Learning para ambientes com múltiplos agentes. A primeira forma é a direta aplicação do método tradicional, em que outros agentes são considerados parte do ambiente, dessa forma a função Q leva apenas em consideração as próprias ações.

A outra forma de se aplicar nesse tipo de ambiente é quando se leva em consideração as ações de outros. Assim, considerando um conjunto de n agentes a função Q é expandida em $Q(s, a_0, \dots, a_n)$. Vários métodos foram propostos para lidar com tais situações.

Mini-Max Q-Learning

Dado um conjunto de agentes e as recompensas, o método mini-max assume que os outros agentes têm como objetivo minimizar o ganho que ele pode receber. Dessa forma, ele tentará maximizar a recompensa que ele pode receber no pior caso.

Littman [14] propôs uma forma de aplicar o mini-max no algoritmo Q-Learning. Supondo dois agentes, seus respectivos conjuntos de ações A e O. Ele define o valor do estado V(s) da seguinte forma:

$$V(s) = \max_{\pi \epsilon PD(A)} \min_{o \epsilon O} \sum_{a \epsilon A} \pi_a Q(s,a,o),$$

Em que π_a é uma probabilidade associada à ação a. Assim, para calcular a distribuição π ótima, é necessário encontrar o maior V que segue essas restrições. Para isso, utiliza-se um algoritmo de programação linear que resolve o sistema.

Friend-or-Foe Q-Learning

Friend-or-Foe (FFQ) é um método proposto em [7] em que o calculo da função V(s) é determinado pela crença que o agente tem sobre outros. Ele pode considerar outros como aliados, dessa forma, ele acredita que eles irão trabalhar juntos para maximizar as suas recompensas. Em contrapartida, ele pode considerar

outros como inimigos, de forma que ele assume que eles irão agir de forma a minimizar a recompensa.

Considerando o conjunto de ações de agentes considerados aliados A, e outro conjunto de ações dos inimigos Y, temos que, no caso geral:

$$Nash_i(s, Q_1, \dots, Q_n) = \max_{\pi \epsilon \Pi(A_1 x \dots x A_n)} \min_{y_1 \in Y_1, \dots, y_n \in Y_n} \sum_{x_1, \dots, x_n \in X_1 x \dots x X_n} \pi(a_1) * \dots * \pi(a_n)\, Q_i(s, a_1, \dots, a_n)$$

Quando consideramos um agente inimigo, a equação nos da o algoritmo mini-max, e pode ser realizado um método de programação linear para encontrar as probabilidades π e o valor máximo da função. A atualização é feita da seguinte forma:

$$Q_i(s, a_1, \dots, a_n) = (1 - \alpha)\, Q_i(s, a_1, \dots, a_n) + \alpha[\, R(s, a_i) + \boldsymbol{\gamma}\, Nash_i(s, Q_1, \dots, Q_n)]$$

Em geral, a ideia é que os aliados trabalham juntos pra maximizar os valores, enquanto os inimigos tentam minimiza-lo. O algoritmo trata um jogo de n-jogador como um jogo de 2 jogadores com um conjunto de ações estendido.

Correlated Q-Learning

Um equilíbrio de Nash é um vetor independente de probabilidades distribuídas sobre as ações. Ele representa uma situação que nenhum jogador tem motivação para mudar sua estratégia. O equilíbrio correlacionado é um equilíbrio mais amplo que o Nash, pois ele é uma distribuição de probabilidade conjunta sobre as ações dos jogadores. Diferentemente do Nash, o correlacionado possui um meio de ser calculado eficientemente utilizando um algoritmo de programação linear.

O algoritmo de equilíbrio correlacionado (CREQ) proposto em [6] toma uma função CE_i que calcula o valor V a partir do vetor $Q^{\rightarrow} = (Q_1, \dots, Q_n)$. Assim, temos que:

$$CE_i(Q^{\rightarrow}(s)) = \{ \sum_{a^{\rightarrow} \in A} \sigma(a^{\rightarrow}) Q_i(s, a^{\rightarrow}) \}$$

E ele tem que satisfazer a restrição:

$$\sigma \in \arg\max_{\sigma \in CE} \sum_{i \in I} \sum_{a^{\rightarrow} \in A} \sigma(a^{\rightarrow}) Q_i(s, a^{\rightarrow})$$

Deep Q-Learning

Quando se leva em conta a limitação do Q-Learning, algoritmos de aproximação do valor Q começam a ganhar destaque. O Deep Q-Learning é uma forma de se estimar os valores de $Q(s,a)$. Ele utiliza uma rede neural que recebe alguma representação do estado e retorna para cada ação possível do agente um valor Q. O artigo da DeepMind [14] apresentou a aplicação desse método em jogos de Atari, e mostrou que ele consegue vencer a nível humano os jogos. A função de perda é dada por:

$$\mathcal{L} = (Q(s,a) - (r + \gamma \max_{a' \epsilon A} Q(s', a'))^2$$

E caso o episódio tenha terminado:

$$\mathcal{L} = (Q(s,a) - r)^2$$

Uma dificuldade para a aplicação desse método é a forma de treinamento. Utiliza-se o algoritmo de backpropagation para o treinamento do modelo, porém o método assume que os dados de treino são independentes e igualmente distribuídos. Em grandes partes de ambientes MDPs essa propriedade não é válida. Para resolver esse problema é utilizado um buffer de experiências. Uma experiência é uma tupla (s, a, r, s'). Assim, a cada interação do agente com o ambiente, é armazenada no buffer a experiência obtida. Quando ocorrer o treinamento, é realizada uma amostra de experiências e possivelmente serão mais ou menos independentes entre si. Para isso é necessário que o tamanho desse buffer seja grande.

Existe outro problema na aplicação prática do método. A equação de Bellman calcula o valor $Q(s,a)$ através do valor $Q(s', a')$. Uma questão é que provavelmente o estado s e s' são bem similares, e a rede pode não conseguir distinguir muito bem entre eles. Assim, ao atualizar os parâmetros da rede para aproximar $Q(s,a)$ para o

real valor Q, pode ocorrer de alterar o valor $Q(s', a')$ ou de outros com estados próximos. Isso faz com que o treino seja muito instável. Para solucionar esse problema pode-se utilizar outra rede que é uma cópia da rede original. Assim, utilizamos a nova rede para fazer o calculo $Q(s', a')$ e elas são sincronizadas a cada N atualizações.

3. DESENVOLVIMENTO DO TRABALHO

O trabalho foi propor quatro modelos de aprendizado por reforço que consegue lidar com ambientes multiagentes. Para isso, foram utilizadas algumas ideias chaves. Primeiramente discutiremos a estrutura dos algoritmos de aprendizado independente, FFQ e CREQ que são baseados no Approximate Q-Learning e em seguida serão apresentados a implementação do Deep Q-Learning.

3.1. Approximate Q-Learning

Os métodos que se baseiam nessa ideia possuem algumas ideias chaves para a sua modelagem. Essas tentam tratar a questão da complexidade do ambiente, assim como acelerar o aprendizado dos métodos. Uma visão geral do agente é dada pela figura a seguir:

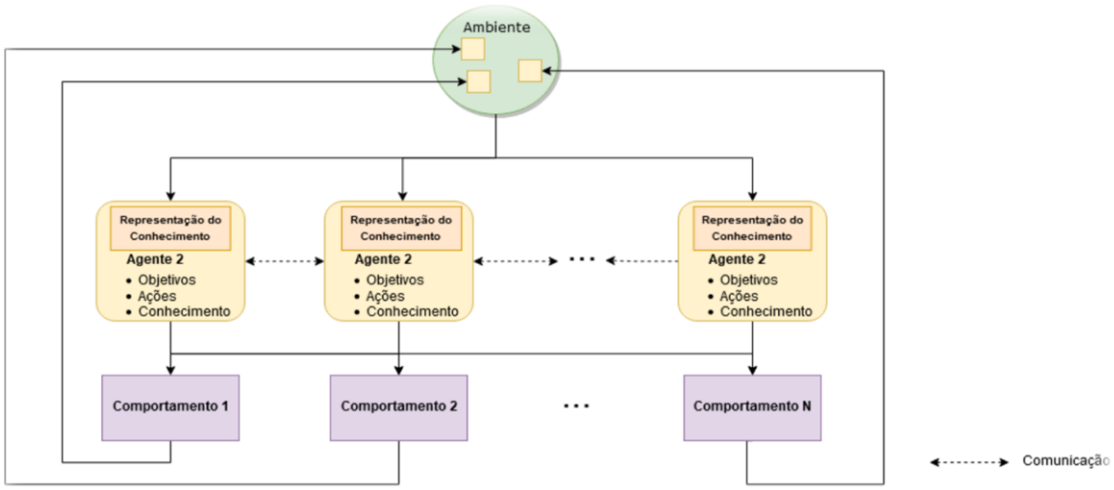

Figura 1: Esquema do modelo de aproximação de estados

3.1.1. Simplificação do espaço de estados

Como foi discutido anteriormente, uma das grandes limitações para o Q-Learning é que em sua versão tabular, armazenar os valores $Q(s,a)$ para cada estado e ação pode ser incomputável. Jogos RTS geralmente possuem um enorme espaço de estados e de ações. StarCraft, um jogo criado pela Blizzard, é estimado possuir 10^{1685} estados possíveis e 10^{50} ações por estado. Dessa forma, para lidar com tal complexidade foi utilizado à função de aproximação. Com isso, é possível estimar os valores Q sem precisar armazena-los diretamente. São extraídas

características que descrevem o estado de forma mais simplificada. Para cada uma das ações disponíveis, é mantido pesos, e, para estimar o valor da ação, é realizada uma multiplicação dos pesos pelas características. Outra vantagem da aproximação é que a atualização dos pesos generaliza para outro conjunto estado-ação, já que alguns estados podem compartilhar características semelhantes.

As características escolhidas na implementação foram:

- Bias, um valor constante em 1;
- Posição x da unidade;
- Posição y da unidade;
- Porcentagem de vida restante;
- Tempo até poder realizar o próximo ataque;
- Número de inimigos no alcance;
- Número de inimigos que podem atacar;
- Número de aliados no alcance;
- Distância para o inimigo mais próximo;
- Alcance máximo de ataque da unidade;
- Porcentagem de vida restante da unidade inimiga mais próxima;
- Tempo de partida;

Cada uma das características é calculada a partir do estado atual do jogo. Elas são normalizadas para ficar no intervalo entre $[0, 1]$.

3.1.2. Simplificação do espaço de ações

Descrever o estado por um vetor de características o simplifica muito, já que geralmente $n < |S|$. Entretanto, o número de ações pode ser muito grande. No método de aproximação, é armazenado um vetor de pesos com o mesmo tamanho de número de características para cara ação possível do agente. Isso pode ser um problema, pois o espaço de ações pode ser muito grande. Por exemplo, em um jogo, pode-se escolher mover a unidade em qualquer posição do mapa, ou escolher atacar qualquer unidade ou estrutura, criando uma grande combinação de possíveis ações. Pensando nisso, em vez de considerar ações de baixo nível, as ações disponíveis é um conjunto de scripts as quais o agente pode seguir. Scripts é um

conjunto de regras de comportamento fixo de mais alto nível, assim como proposto em [1].

Os scripts escolhidos são:

- **AlphaBeta:** realiza uma busca AlphaBeta com profundidade baixa e retorna a melhor ação a ser realizada
- **AttackClosest:** ataca a unidade mais próxima;
- AttackValue: ataca a unidade com maior valor (medido por dano por segundo)
- **AttackWeakest:** ataca a unidade com menor porcentagem de vida
- **No-Overkill AttackValue (NOKAV):** mesmo que o AttackValue, mas tenta não causar dano extra;
- **Kiter:** ataca a unidade mais próxima, recuando das unidades inimigas entre ataques;
- **Kiter AttackValue:** mesmo que o Kiter, mas ataca a unidade de maior valor (dano por segundo);
- **Kiter NOKAV:** mesmo que o Kiter, mas ataca de acordo com o NOKAV;
- **MoveForward:** move para frente;
- **MoveBackward:** move para trás
- **Cluster:** faz a unidade se aproximar das aliadas a partir de um centro;

3.1.3. Acelerando o aprendizado

Quando estamos lidando com o aprendizado independente, os agentes consideram outros como parte do ambiente, ou seja, ele não modela o comportamento desses agentes de forma explicita. Quando se está em um ambiente cooperativo, os agentes devem encontrar uma combinação de ações conjunta apenas observando recompensas, sem considerar a existência de outros. Embora seja demonstrado que com uma função de recompensa igual à coordenação nesse cenário ocorre, não existe garantias sobre o tempo de convergência. Para isso, foi utilizado uma técnica para agregar o conhecimento dos agentes. A ideia é que cada unidade está explorando diferentes partes do conjunto de estado-ações possíveis. Com esse método é possível que um agente se beneficie da experiência de outros,

fazendo com que o algoritmo convirja mais rápido. No nosso modelo, ao final de cada episódio (partida), os pesos agregados (globais) $W^{g}{}_{i}^{a}$ da ação a associado à característica i do estado é atualizado da seguinte forma:

$$W^{g}{}_{i}^{a} \leftarrow \frac{\sum_{u \in U} u_{W_i^{a}{}_*} C_u(a)}{\sum_{u \in U} C_u(a)},$$

Em que $C_u(a)$ é uma função que contabiliza quantas vezes a unidade u escolher o script a, e U é o conjunto de todos os agentes. Assim, W^{ga} resultante é a soma dos pesos das unidades, ponderado pelo quanto à unidade escolheu a ação, normalizada pelo quanto que a ação foi escolhida no total. Ao final da partida, o pelo global é distribuído entre os agentes.

3.1.4. Agrupamento de Unidades

Para os métodos de coordenação, a função Q é expandida para considerar ações de outros agentes. Dessa forma, cada uma das unidades mantém um modelo de outros, e suas crenças sobre os valores das ações para as outras unidades. Na expansão do método Q-Learning com n agentes, a tabela Q manteria um valor $Q_i(s, a_1, \ldots, a_n)$, para cada estado e cada combinação de ações. Mesmo utilizado a aproximação do estado e scripts como ações, o tamanho da tabela seria:

$$|Q| = |F| * n^{|A|},$$

Em que $|A|$ é o número de scripts, $|F|$ é o número de características extraídas do estado e n é o número de agentes. Quando estamos tratando de ambientes como Jogos RTS, mesmo que o número de scripts possa ser pequeno, existe um grande número de unidades. Em um jogo real de StarCraft, o limite é de 200 unidades, tornando essa tabela inviável de ser armazenada e de ser realizado operações nela. Para lidar com isso, foi realizado um agrupamento das unidades. No inicio do jogo é separado unidades de tipos iguais, e então é feito a divisão de grupos. Unidades de um mesmo grupo compartilham o conhecimento, e para a análise da tabela Q é levado em consideração apenas o grupo como um todo. Com isso, é reduzido muito o tamanho de valores Q armazenados.

3.2. Deep Q-Learning

O algoritmo DQN é um método que utiliza uma rede neural para tentar estimar valores $Q(s, a)$ para cada possível ação do agente. Para isso, a rede recebe como entrada alguma representação do estado atual do ambiente. Em muitas aplicações para jogos, a utilização da imagem original do jogo é comum, assim como em [4]. Para a nossa aplicação, utilizamos uma simplificação da imagem do jogo. Como em [8], a entrada do modelo é um conjunto de matrizes binárias que representam o estado atual do jogo. Cada um dos planos corresponde a uma representação do mapa, e terá valor 1 nas posições das unidades que tiverem uma característica que àquele plano representa e 0 caso contrário.

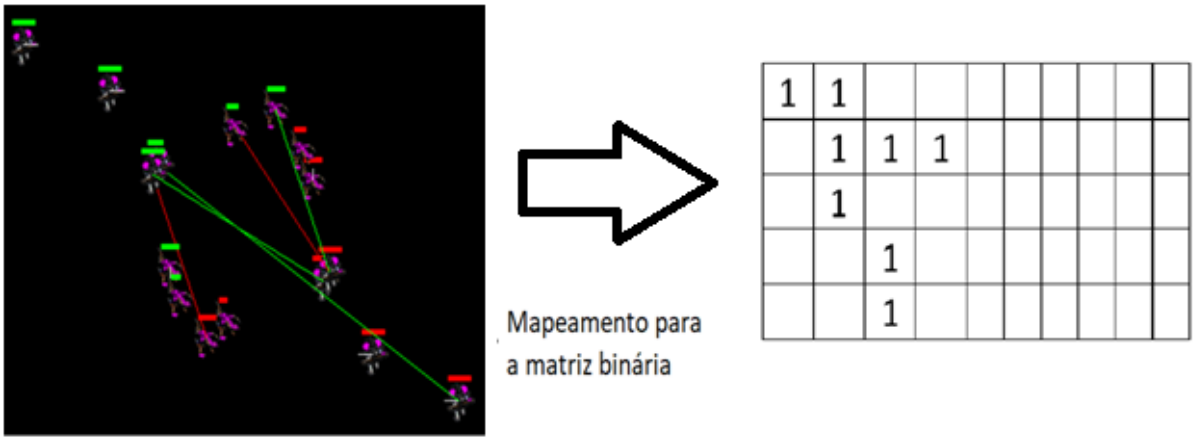

Figura 2: Transformação para a representação binária da posição das unidades do jogador verde

As características utilizadas para formar a entrada da imagem são:

- A unidade escolhida;
- Unidades com vida restante menor que 25%;
- Unidades com vida restante entre 25-50%;
- Unidades com vida restante entre 50-75%;
- Unidades com vida restante acima de 75%;
- Tipos de unidades;
- Unidades aliadas;
- Unidades inimigas;
- Unidades que podem atacar;
- Unidades que podem se mover;

A arquitetura escolhida teve como inspiração na DeepMind para Atari, um modelo que conseguiu derrotar jogos para Atari 2600 com performance parecida com a de seres humanos.

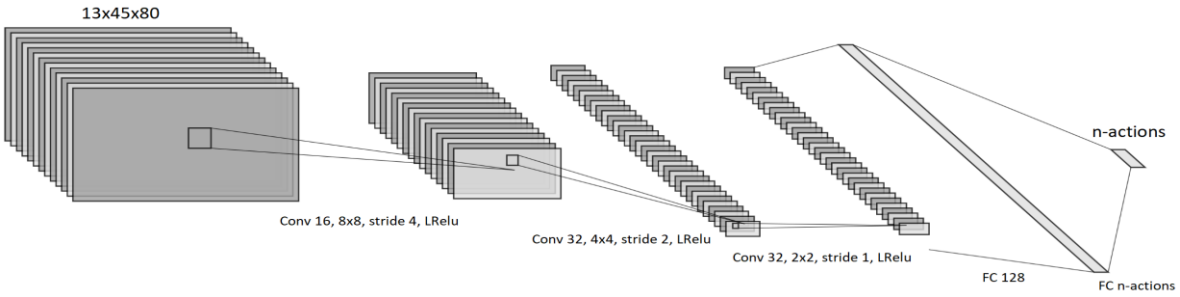

Figura 3: Arquitetura da rede DQN

Assim como nos outros modelos apresentados, em vez da rede calcular valores Q para ações de baixo nível, ela considerará suas ações como scripts.

3.3. Recompensa

A recompensa é uma parte de extrema importância no aprendizado por reforço. É a partir dela que o agente tem uma medida se a ação tomada é boa ou não. A melhor forma de se aplicar a recompensa em jogos é atribuí-la ao final, sendo positiva se a partida foi ganha e negativa caso contrário. Um problema dessa abordagem é que dependendo do ambiente, uma partida pode levar muitas interações com o ambiente, fazendo com que a recompensa se torne muito esparsa. Com isso, o agente pode levar muito tempo ate aprender um comportamento ótimo.

LTD2 é uma métrica apresentada em [2], que tenta calcular o valor do estado em uma situação de combate em jogos de StarCraft. Ele considera a vida atual das unidades e o dano por segundo que elas podem causar comparado às do oponente. O LTD2 é calculado da seguinte forma:

$$LTD2(s) = \sum_{u \in U_1} \sqrt{hp(u)} * dpf(u) - \sum_{o \in U_2} \sqrt{hp(o)} * dpf(o)$$

Para os nossos modelos, nós atribuímos uma recompensa a cada interação do agente com o ambiente. Para tentar avaliar as ações intermediárias, nós utilizamos a métrica LTD2. O objetivo é que o agente consiga aprender ações mais

rapidamente durante uma partida. Ao final de um episódio, é realizada a atribuição de uma recompensa maior e de acordo com o resultado final.

RESULTADOS E DISCUSSÃO

Na primeira parte do trabalho, os experimentos tiveram como foco o algoritmo de aproximação com aprendizado independente com o objetivo de entender o comportamento no cenário descrito. Para isso, foi usado um simulador de combate em StarCraft, chamado SparCraft. Desenvolvido por Dave Churchill, ele permite a composição de diferentes tipos de exércitos. Além disso, existem diversos comportamentos já implementados o que permite o teste contra diferentes tipos de adversários.

O primeiro teste para validação da abordagem foi contra um oponente de comportamento fixo. Por não ter variância em sua estratégia, o nosso modelo deveria conseguir aprender ações para ganhar. Esse experimento foi executado com diferentes composições de exércitos e número de unidades. Com um número de unidades $n \in \{8, 32, 96\}$, foram utilizados as seguintes composições de unidades:

- $\frac{n}{2}$ *Protoss Dragoon e* $\frac{n}{2}$ *Protoss Zealot: Ambas são unidades fortes. Dragoons são unidades de ataque à distância, e os Zealots são de ataques próximos;*

- $\frac{n}{2}$ *Protoss Dragoon e* $\frac{n}{2}$ *Terran Marine: Ambas são unidades de ataques à distância. Dragoons são unidades fortes, e Marines são fracas;*

- $\frac{n}{2}$ *Zerg Zergling e* $\frac{n}{2}$ *Terran Marine: Ambas são unidades fracas. Marines são unidades de ataque à distância, e os Zerglings são de ataques próximos;*

O oponente de comportamento fixo escolhido para o teste foi o script AttackClosest. Ele irá atribuir para todas as unidades à ação de atacar o oponente mais próximo.

Partidas de Treino	Unidades	Número de Unidades	Taxa de vitória no Teste
1,00E+06	Zergling Marine	8	95,40%
1,00E+06	Zergling Marine	32	75,05%
1,00E+06	Zergling Marine	96	69,70%
1,00E+06	Dragoon Marine	8	99,40%
1,00E+06	Dragoon Marine	32	97,70%
1,00E+06	Dragoon Marine	96	94,85%
1,00E+06	Dragoon Zealot	8	99,80%
1,00E+06	Dragoon Zealot	32	99,20%
1,00E+06	Dragoon Zealot	96	99,35%

Tabela 1: Experimentos contra oponente de comportamento fixo AttackClosest

Dessa forma, os experimentos mostraram que nossa abordagem consegue ganhar desse tipo de oponente de forma concisa e em todos os cenários apresentados mesmo sem a coordenação explícita. Uma observação sobre o cenário Zergling e Marine, que não apenas nesse experimento como em vários outros se mostrou o pior caso. Em que o nosso modelo apresenta dificuldade em aprender.

O próximo experimento foi em relação à taxa de aprendizado. Em muitas outras aplicações que utilizam a função de aproximação, taxas de aprendizado menores melhoram o aprendizado. Assim, testamos contra o mesmo oponente de comportamento fixo, no nosso pior caso, e variando a taxa de aprendizado.

Partidas de Treino	Unidades	Número de Unidades	Taxa de Aprendizado	Taxa de Vitória no Teste
1,00E+06	Zergling Marine	32	0,1	32,30%
1,00E+06	Zergling Marine	32	0,01	62,30%
1,00E+06	Zergling Marine	32	0,001	58,10%
1,00E+06	Zergling Marine	32	0,0001	69,60%
1,00E+06	Zergling Marine	32	0,00001	19,80%

Tabela 2: Experimentos contra oponente de comportamento fixo variando taxa de aprendizado

Dessa forma, os experimentos mostraram que de fato taxas de aprendizado menores têm certo benefício. Em todos os casos, a curva de aprendizado foi crescente, porém, para taxas de aprendizado grandes, a convergência foi muito rápida, mas não para o ótimo. Conforme a taxa foi diminuindo, a convergência demorou mais, porém teve um ganho grande em desempenho. Até certo limite, em que embora tivesse convergindo, contra um oponente de comportamento fixo, um

milhão de partidas não foram suficientes para a convergência. Logo, as taxas foram fixadas entre 0,01 e 0,0001 para todos os outros experimentos. Esses resultados também apontaram na direção para a escolha dos hiperparâmetros dos algoritmos de coordenação.

Por último, foram feitos testes contra oponentes de busca. Eles são oponentes mais robustos, em que pensam sobre suas ações, e as possíveis ações do oponente. Alguns deles, como o GAB e SAB são estado da arte para essa aplicação. Por se basearem em busca, alguns testes demoraram muito tempo para serem realizados, assim, um número menor de unidades foi testado, e por menos partidas. Os cenários apresentados são Dragoon Zealot, que é o nosso melhor caso, e o Zergling Marine, o pior caso.

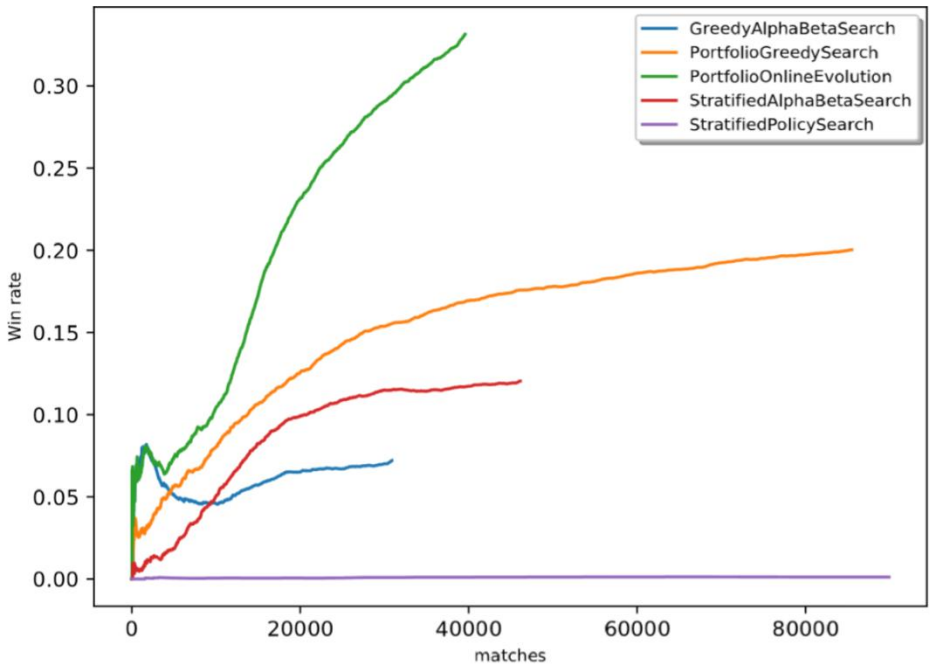

Figura 4: Gráfico de taxa de vitória contra oponentes de busca com o exército composto de Dragoons e Zealots

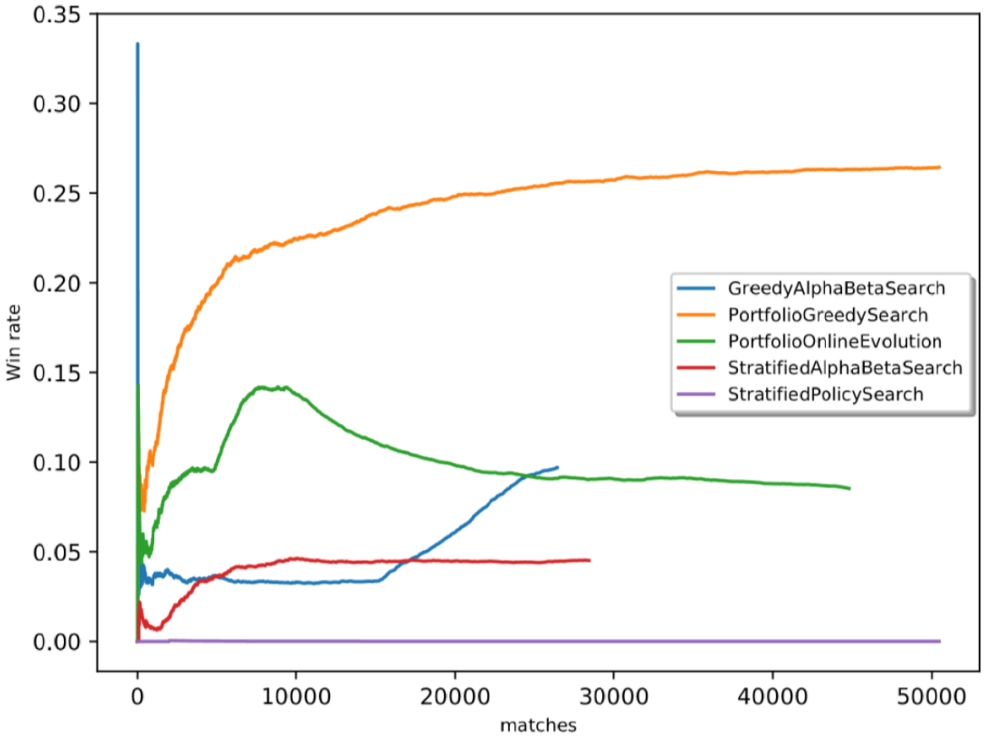

Figura 5: Gráfico de taxa de vitória contra oponentes de busca com o exército composto de Zerglings e Marines

Pelos resultados, pode-se perceber que de fato o Zergling Marine é o pior caso do nosso modelo. Além disso, o Stratified Policy Search (SSS) em ambos consegue ganhar de forma categórica, embora não seja o oponente mais forte. Além disso, podemos ver que o ganho comparado a esses algoritmos não é ainda satisfatório, mesmo as curvas mostrando um aprendizado.

Para a segunda parte do trabalho, foram testados os algoritmos de coordenação, assim como o método DQN. Em relação aos de coordenação, os testes foram realizados contra o oponente de comportamento fixo, e, após isso, os pesos foram usados para o treino contra os oponentes mais sofisticados.

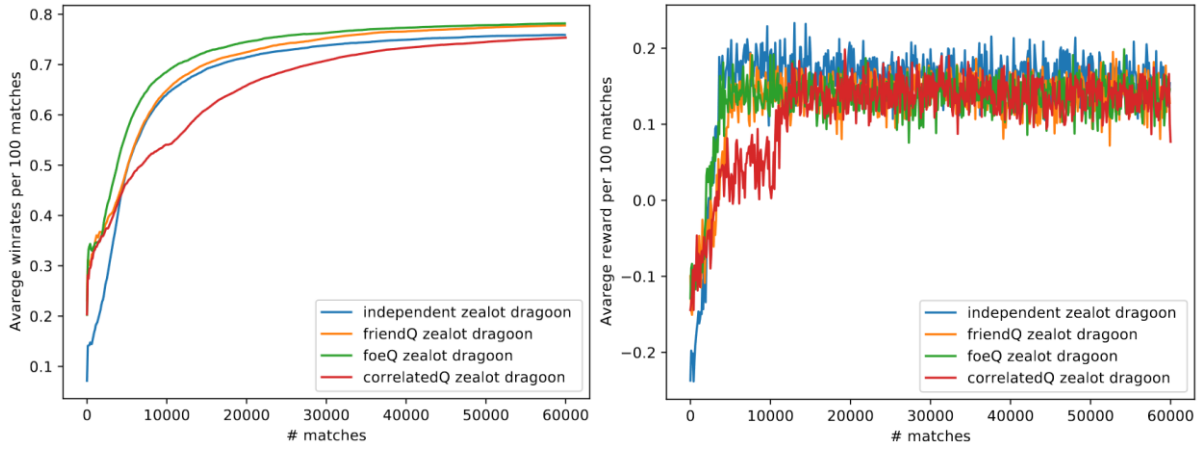

Figura 6: Gráfico de taxa de vitória e recompensa com a composição Dragoon e Zealot contra oponente comportamento fixo

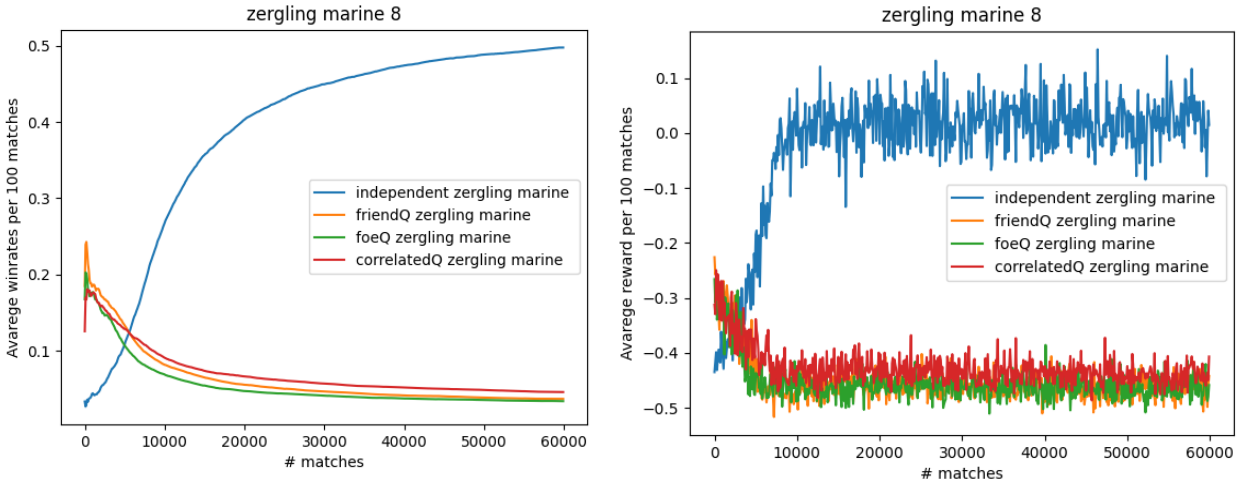

Figura 7: Gráfico de taxa de vitória e recompensa com a composição Zergling e Marine contra oponente de comportamento fixo

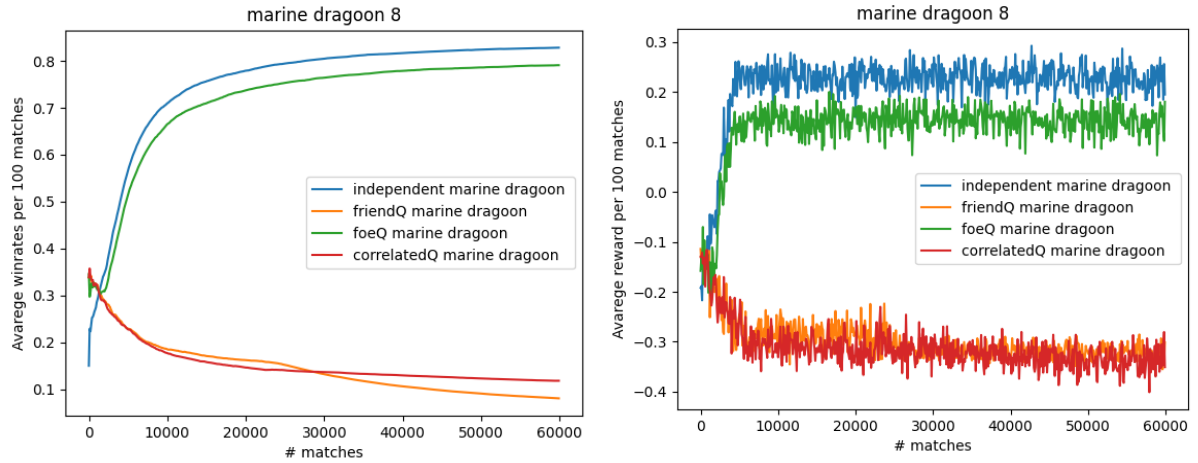

Figura 8: Gráfico de taxa de vitória e recompensa com a composição Marine e Dragoon contra oponente de comportamento fixo

Os gráficos mostram que apenas no cenário Zealot Dragoon, o melhor caso do algoritmo, os métodos de coordenação conseguem aprender de forma concisa. Os outros dois casos, os métodos de coordenação tiveram um problema para descobrir estratégias para ganhar. Além disso, o independente apresentou uma maior taxa de vitória nesses dois cenários.

Quando verificamos as escolhas dos métodos na composição do Zergling e Marine, é possível entender melhor o porquê da taxa de vitória não ter sido tão boa.

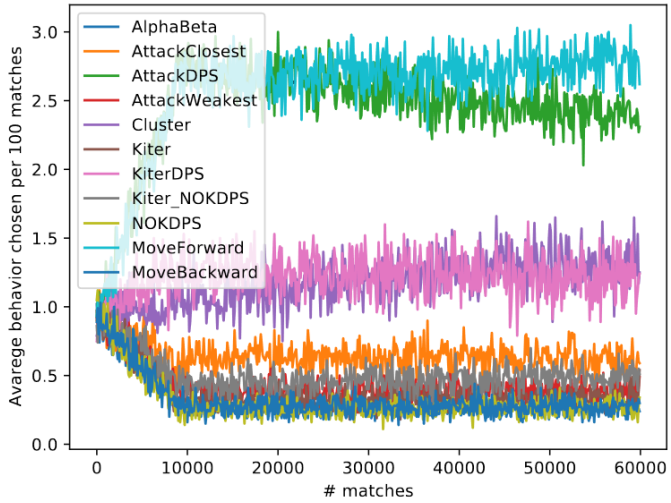

Figura 9: Gráfico de escolha de scripts por partida na composição Zergling Marine do CorrelatedQ

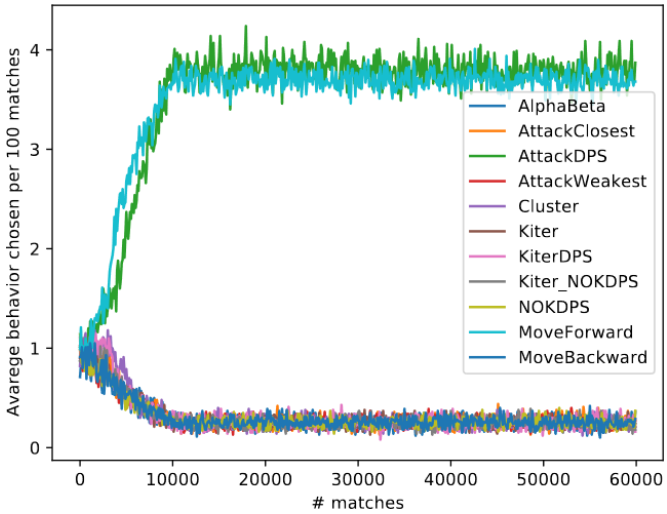

Figura 10: Gráfico de escolha de scripts por partida na composição Zergling Marine do FoeQ

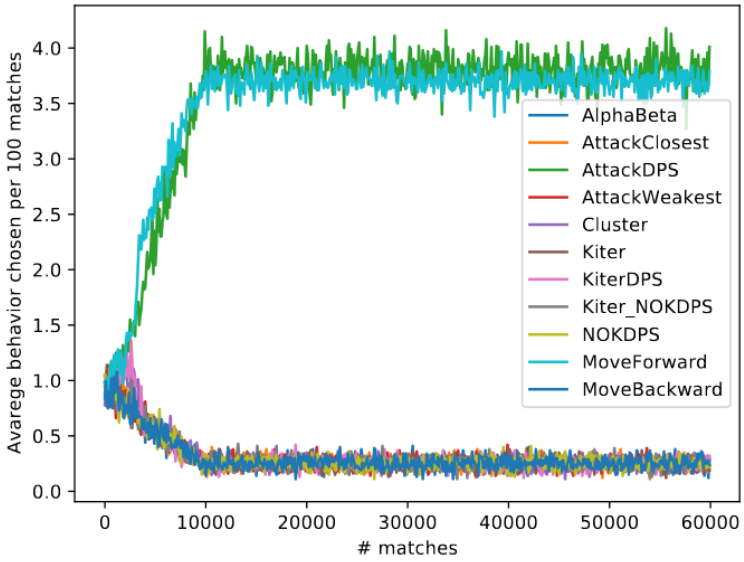

Figura 11: Gráfico de escolha de scripts por partida na composição Zergling Marine do FriendQ

Observando esses dados, é possível perceber que em todos o comportamento MoveForward foi escolhido muitas vezes. Esse comportamento faz com que a unidade move para frente, mas ele não ataca. Embora em certos cenários ele possa ser útil, como principal comportamento ele não é bom.

Dessa forma, foi escolhido o melhor cenário (Zealot Dragoon) para testar contra oponentes mais sofisticados, para ver se os métodos trazem um ganho contra eles em comparação ao independente.

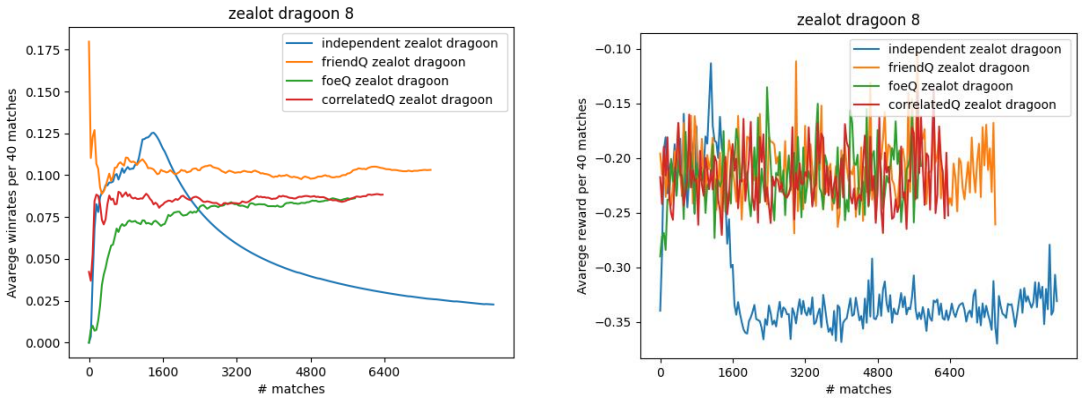

Figura 12: Gráfico de taxa de vitória e recompensa com a composição Marine e Dragoon contra POE

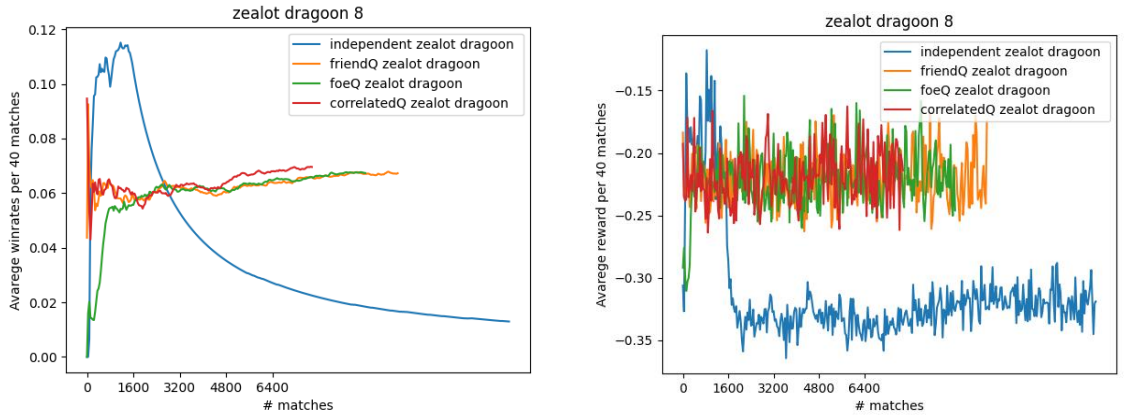

Figura 13: Gráfico de taxa de vitória e recompensa com a composição Marine e Dragoon contra PGS

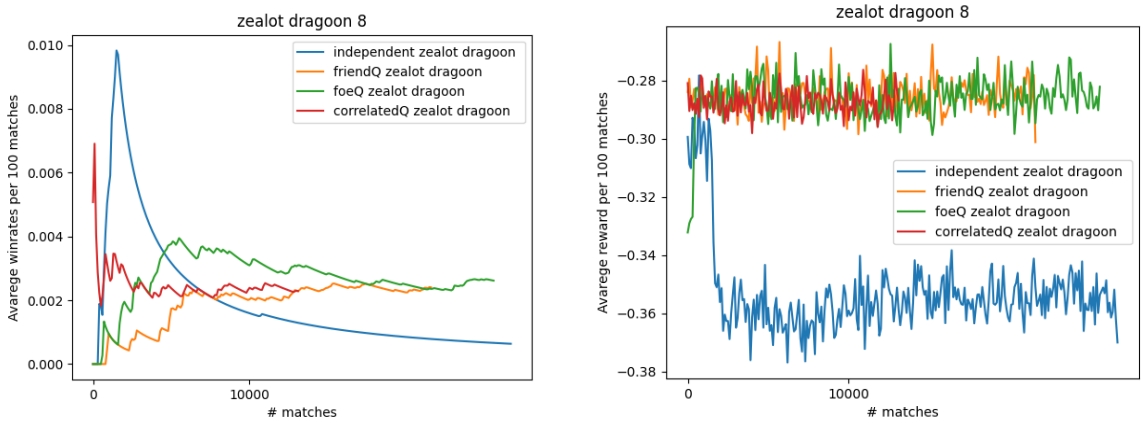

Figura 14: Gráfico de taxa de vitória e recompensa com a composição Marine e Dragoon contra SSS

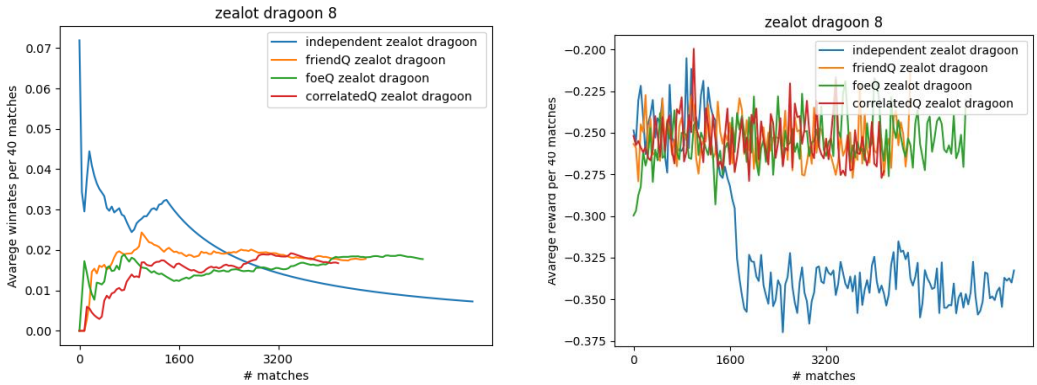

Figura 15: Gráfico de taxa de vitória e recompensa com a composição Marine e Dragoon contra SSS

É possível ver que contra esses oponentes a taxa de vitória ainda é baixa, mesmo com os métodos de coordenação. Mas um aspecto interessante de se notar é que o método de aprendizado independente apresenta uma queda grande nos primeiros episódios. Como vistos nas figuras 2 e 3, ele irá demorar um longo tempo para começar a aprender, já que precisa coordenar as unidades de forma não

explicita contra um oponente bem sofisticado. Já os métodos de coordenação, todos tem um aumento na taxa de vitória, mesmo que pequeno, e logo convergem, porém em uma estratégia não muito boa contra esses oponentes.

Para o DQN, foram feitos alguns experimentos iniciais para determinar os melhores parâmetros e arquitetura da rede. A escolha final foi uma variante da arquitetura apresentada em [4]. Alguns parâmetros testados foram o tamanho do buffer, o tempo para sincronizar as redes, o batch que determina o número de experiências que serão amostradas entre outros. Alguns dos experimentos feitos são:

opp treino	partidas treino	alpha	gamma	epsilon	batch size	buffer size	sync	# units	units types	win rate
AttackClosest	30000	1,00E-05	9,00E-01	1,0 - 0,05	100	100k	1k	4	Dragoon Zealot	57,80%
AttackClosest	30000	1,00E-05	9,00E-01	1,0 - 0,05	100	100k	5k	4	Dragoon Zealot	70,60%
AttackClosest	30000	1,00E-05	9,00E-01	1,0 - 0,05	100	100k	5k	16	Dragoon Zealot	81,30%
AttackClosest	30000	1,00E-05	9,00E-01	1,0 - 0,05	100	100k	5k	4	Zerg Marine	47,60%
AttackClosest	30000	1,00E-05	9,00E-01	1,0 - 0,05	100	100k	5k	16	Zerg Marine	68,50%
AttackClosest	40000	1,00E-05	9,00E-01	1,0 - 0,1	64	100k	1k	4	Zerg Marine	61,20%

Tabela 3: Testes do DQN contra oponentes de comportamento fixo

Um aspecto interessante desses resultados é perceber que, como as unidades utilizam apenas uma rede para escolher as ações, o maior número de unidades faz com que a rede convirja mais rapidamente.

Após a escolha da arquitetura e dos hiperparâmetros, outros testes foram realizados. Foram feitos dois experimentos, o primeiro foi o treino da rede diretamente contra o oponente de comportamento fixo, no caso o AttackClosest, e o segundo o treino foi feito com selfplay, em que o modelo treina contra ele mesmo. Em seguida foi testado o resultado do treino contra o oponente.

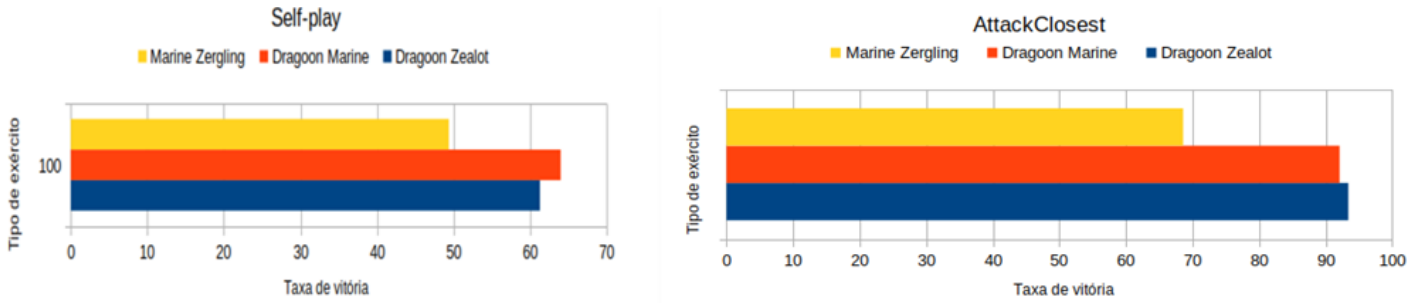

Figura 16: Taxas de vitória contra o oponente AttackClosest

Os gráficos mostram que o DQN consegue ganhar de forma concisa nesse cenário. Em seguida, foi feito o experimento contra os mesmos oponentes de busca. Como a rede demora a aprender e a execução dos oponentes de busca são lentas, foi utilizado os pesos do treinamento contra o AttackClosest para a execução, assim, esperando que o processo seja agilizado. Os resultados foram:

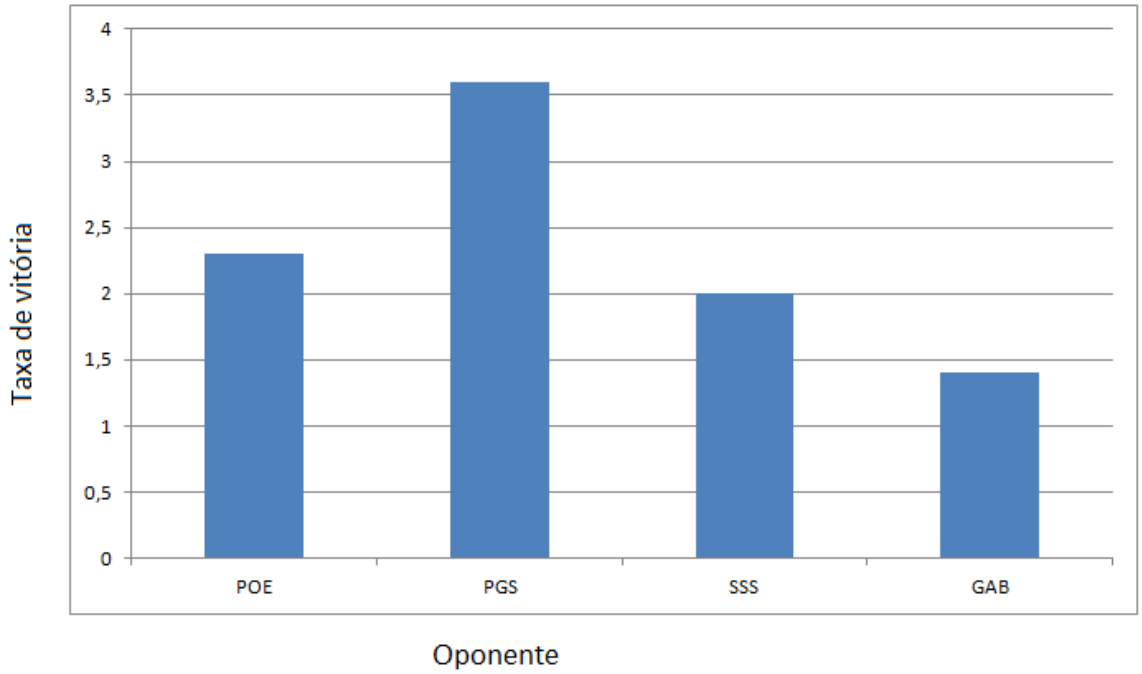

Figura 17: Gráfico de taxa de vitória contra oponentes de busca

O método de Deep Q-Learning também não conseguiu um bom desempenho contra esses métodos, embora seja um algoritmo mais robusto.

CONCLUSÕES

O trabalho tinha como foco apresentar modelos para se lidar com ambientes multi agentes, e apresentar o comportamento do método de aprendizado independente comparado ao de coordenação. Com a realização dos experimentos, foi possível ver que os métodos apresentados não obtiveram grande sucesso contra oponentes sofisticados.

Em relação aos modelos de coordenação, as adaptações feitas e simplificações para tornar viável à aplicação podem ter influenciado no desempenho dos métodos.

Quanto ao DQN, alguns fatores podem ter influenciado. Primeiro é que para determinar os hiperparâmetros e arquitetura, foi utilizado o script AttackClosest, que comparado aos outros oponente é bem simples. Outro é o tempo de treinamento, já que os algoritmos de busca levam um longo tempo para escolher as ações, o que limitou o número de partidas de treino.

Uma última possível razão é a forma de recompensa adotada. A recompensa durante a partida pode influenciar a escolha de ações não ótimas. Se por exemplo, um agente está ganhando, e decide ficar parado, ele receberá recompensas boas durante a partida e mesmo que ele perca, a soma pode ser positiva.

REFERÊNCIAS

[1] Churchill, David & Buro, Michael. (2013). Portfolio greedy search and simulation for large-scale combat in starcraft. 1-8. 10.1109/CIG.2013.6633643.

[2] C. Wang, P. Chen, Y. Li, C. Holmgård and J. Togelius, "Portfolio online evolution in StarCraft", *Proc. 12th Artif. Intell. Interact. Digit. Entertainment Conf.*, pp. 114-121, 2016.

[3] Lelis, L. H. S. 2017. Stratified strategy selection for unitcontrol in real-time strategy games. InInternational JointConference on Artificial Intelligence, 3735–3741.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, & Martin Riedmiller. (2013). Playing Atari with Deep Reinforcement Learning.

[5] Stone, Peter & Veloso, Manuela. (2000). Multiagent Systems: A Survey from a Machine Learning Perspective. Autonomous Robots. 8. 10.1023/A:1008942012299.

[6] Greenwald, Amy & Hall, Keith. (2004). Correlated-Q Learning.

[7] Littman, M. L. (2001). Friend-or-Foe Q-learning in General-Sum Games. In Proceedings of the Eighteenth International Conference on Machine Learning (ICML01) (pp. 322–328). San Francisco, CA, USA: Morgan Kaufmann.

[8] Barriga, Nicolas A. & Stanescu, Marius & Besoain, Felipe & Buro, Michael. (2019). Improving RTS Game AI by Supervised Policy Learning, Tactical Search, and Deep Reinforcement Learning. IEEE Computational Intelligence Magazine. 14. 11. 10.1109/MCI.2019.2919363.

[9] Lowe, R. & Wu, Y. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. In Aviv Tamar et al. Advances in Neural Information Processing Systems

[10] Claus, C., & Boutilier, C. (1998). The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (pp. 746–752).

[11] Peng, P., Yuan, Q., Wen, Y., Yang, Y., Tang, Z., Long, H., ... Group, A. (2017). Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games

[12] Panait, L. & Luke, S. (2005). Cooperative Multi-Agent Learning: The State of the Art. In Journal Autonomous Agents and Multi-Agent Systems (pp. 387-434).

[13] DEEP Reinforcement Learning Hands-O. 2. ed. rev. e aum. [*S. l.*]: Packt, 2020.

[14] Littman,M.L.(1994).Markov games as a framework for multiagent reinforcement learning. Proceedings of the Eleventh International Conference on Machine Learning (pp.157{163).