
# Estudo de Aprendizagem por Reforço em Jogo Tower Defense

**Augusto Vicente Fernandes Dias**

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação. No âmbito da dupla diplomação com a Universidade Tecnológica Federal do Paraná.

Trabalho orientado por:

Prof. Dr. Rui Pedro Lopes

Prof. Dr. Juliano Henrique Foleiss

Bragança

2019-2020


# Estudo de Aprendizagem por Reforço em Jogo Tower Defense

**Augusto Vicente Fernandes Dias**

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação. No âmbito da dupla diplomação com a Universidade Tecnológica Federal do Paraná.

Trabalho orientado por:

Prof. Dr. Rui Pedro Lopes

Prof. Dr. Juliano Henrique Foleiss

Bragança

2019-2020

# Dedicatória

Dedico este trabalho aos meus pais e meus irmãos, que sempre estiveram me apoiando e sem eles certamente nada teria acontecido.

# Agradecimentos

Quero agradecer primeiramente a Deus por iluminar meu caminho até os dias de hoje.

Quero agradecer aos meus pais e meus irmãos que sempre me incentivaram, apoiaram e me ajudaram nas escolhas da vida desde muito cedo, amo-os.

Quero agradecer a minha namorada por estar sempre me apoiando, mesmo que à distância.

Quero agradecer aos meus amigos, todos aqueles que acompanharam de perto e de longe nessa minha jornada até o fim.

Quero agradecer ao meu orientador Rui Pedro Lopes que me acolheu como orientando, e me ajudou quando precisei.

Quero agradecer ao meu co-orientador Juliano Henrique Foleiss, que desde o início desde trabalho vem me ajudando.

Quero agradecer ao professor Marcos Silvano por ter feito a ponte que possibilitou minha vinda até Portugal.

Quero agradecer às duas universidades, UTFPR e IPB por terem sido minha segunda casa e possibilitado meu crescimento educacional.

# Resumo

A aprendizagem por reforço está revolucionando a inteligência artificial, isso representa que sistemas autônomos estão compreendendo cada vez mais o mundo visual. A aprendizagem por reforço é uma das abordagens de aprendizagem de máquina que funciona alterando o comportamento do agente por meio de *feedbacks*, como recompensas ou penalidades por suas ações. Trabalhos recentes utilizam aprendizagem por reforço para treinar agentes capazes de jogar jogos eletrônicos e obter pontuações até mais altas que jogadores humanos profissionais. As aplicações para agentes inteligentes em jogos incluem propiciar desafios mais complexos aos jogadores, melhorar a ambientação dos jogos proporcionando interações mais complexas e até servem como forma de prever o comportamento dos jogadores quando o jogo está em fase de desenvolvimento. A maioria dos trabalhos atuais, derivados de uma arquitetura conhecida como rede Q profunda, trabalham usando técnicas de aprendizagem profunda para processar a imagem do jogo, criando uma representação intermediária. Esta representação é, então, processada por camadas de rede neural capazes de mapear situações do jogo em ações que visam maximizar a recompensa ao longo do tempo. Entretanto, este método é inviável em jogos modernos, renderizados em alta resolução com taxa de quadros cada vez maior. Além disto, este método não funciona para treinar agentes que não estão mostrados na tela. Desta forma, neste trabalhos propomos um *pipeline* de aprendizagem por reforço baseado em redes neurais cuja entrada são metadados fornecidos diretamente pelo jogo e as ações são mapeadas diretamente em ações de alto-nível do agente. Propomos esta arquitetura para um agente jogador de defesa de torre, um jogo de estratégia em tempo real cujo agente não é representado na tela diretamente.

**Palavras-chave:** Aprendizagem por Reforço, Inteligência Artificial, Rede Neural, Tower Defense.

# Abstract

Reinforcement learning is revolutionizing artificial intelligence, this means that autonomous systems are increasingly understanding the visual world. Reinforcement learning is one of the machine learning approaches that works by changing the agent's behavior through feedbacks, such as rewards or penalties for his actions. Recent work using reinforcement learning to train agents capable of playing electronic games and obtain scores even higher than professional human players. As applications for intelligent agents in games can offer more complex challenges to players, improve the ambience of more complex interactive games and even visualize the behavior of players when the game is in development. Most current works, using an architecture known as the deep Q network, use deep learning techniques to process an image of the game, creating an intermediate representation. This representation is then processed by layers of neural network capable of mapping game situations into actions that aim to maximize a reward over time. However, this method is not feasible in modern games, rendered in high resolution with an increasing frame rate. In addition, this method does not work for training agents who are not shown on the screen. Thus, in this work we propose a reinforcement learning pipeline based on neural networks, whose input is metadata, selected directly by the game, and the actions are mapped directly into high-level actions by the agent. We propose this architecture for a tower defense player agent, a real time strategy game whose agent is not represented on the screen directly.

**Keywords:** Reinforcement learning, artificial intelligence, Neural network, Tower Defense.

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

Um dos principais objetivos do campo da Inteligência Artificial (IA) é produzir agentes totalmente autônomos que interajam com seus ambientes para aprender comportamentos ótimos, melhorando ao longo do tempo através de tentativa e erro. Criar sistemas de IA que sejam responsivos e possam efetivamente aprender tem sido um desafio de longa data, desde robôs, que podem perceber e reagir ao mundo ao seu redor, até agentes puramente baseados em software, que podem interagir com linguagem natural e multimídia [1].

Dentro do campo de estudo da IA existem inúmeras abordagens, modelos propostos e arquiteturas. A aprendizagem de máquina supervisionada é uma abordagem no qual o processo de treinamento se baseia em respostas já conhecidas. É necessário que o agente tenha que mapear a entrada com uma saída. No entanto, em uma situação onde há uma sequência muito grande de ações a serem rotuladas, esta abordagem se torna inviável.

Uma outra abordagem é a Reinforcement Learning (RL), que pode ser determinada como uma estrutura matemática baseada em princípios psicológicos e neurológicos para a aprendizagem autônoma orientada pela experiência. Embora a RL tenha tido alguns sucessos no passado, as abordagens anteriores não tinham escalabilidade e eram inerentemente limitadas a problemas de baixa dimensionalidade. Essas limitações existem pois os algoritmos de RL compartilham os mesmos problemas de complexidade que outros algoritmos: complexidade de memória, complexidade computacional e, no caso de algoritmos de aprendizado de máquina, complexidade da amostra. O que temos testemunhado

nos últimos anos foi a ascensão do Deep Learning (DL), confiando na poderosa função de aproximação e representação das propriedades de aprendizagem das Deep Neural Network (DNN)s, nos forneceu novas ferramentas para enfrentar esses problemas [1].

RL tem uma ampla gama de aplicações, como: jogos, robótica, processamento de linguagem natural, visão computacional, gerenciamento de negócios, finanças e outros. Jogos e robótica, são duas áreas de aplicação clássica de RL [2].

No contexto de jogos eletrônicos, agentes inteligentes podem criar situações mais desafiadoras ao jogador. Em específico, a elaboração de estratégias de acordo com o comportamento do jogador no ambiente podem fazer com que as ações dos agentes autônomos deixem de ser obvias e previsíveis.

Uma categoria dos jogos de estratégia propício para estudar aprendizagem supervisionada é o Tower Defense (TD). Esse tipo de jogo requer pensamento estratégico ao longo do tempo para formular um plano que maximize o tempo de sobrevivência do jogador. Como é impraticável rotular as ações que o agente deve tomar a cada instante de tempo, a aprendizagem por reforço é uma estratégia de aprendizagem viável para o problema.

Explorar aplicações de IA em jogos não oferece risco à vidas. Portanto, é uma aplicação fértil para a discussão de novas ideias na área, atualmente muitos trabalhos sobre RL são aplicados em jogos, como [3], [4], [5], [6].

Muitas vezes, a entrada fornecida ao agente são *pixels* da tela [3], [7], e [8]. Essa abordagem faz com que o agente tenha que fazer um mapeamento sensorial para processar os dados na rede. Neste trabalho a abordagem utilizada para entrada de dados para o agente são de metadados, simplificando o modelo e economizando processamento.

## 1.1 Objetivos

O objetivo geral é desenvolver um agente inteligente baseado em aprendizagem por reforço para jogar Tower Defense usando metadados da partida como entrada. Os objetivos específicos são: estudar os conceitos de aprendizagem por reforço baseada em redes neurais; projeto do agente inteligente usando metadados na entrada; implementação do agente; e

avaliação do agente;

## 1.2 Estrutura do Documento

Os trabalhos relacionados e as ferramentas utilizadas são apresentados no Capítulo 2. O Capítulo 3 traz as informações referentes ao método abordado neste trabalho. O capítulo 4 conta com o desenvolvimento e implementação do *Tower Defense* e do agente inteligente. O capítulo 5 contém informações relacionadas a avaliação dos resultados e discussão dos mesmos. Por fim o Capítulo 6 que contem a conclusão e os trabalhos futuros.

# Capítulo 2

# Contexto e Ferramentas

Neste capítulo são apresentados conceitos relevantes sobre RL, algumas aplicações, o algoritmo Q-Learning e, por fim, detalhes sobre Artificial Neural Network (ANN) para RLs.

## 2.1 Em Geral

Machine Learning (ML) é a prática de usar algoritmos para coletar dados, aprender com eles e, então, fazer uma predição sobre algo. Desta forma, ao invés de implementar as rotinas de software, com base em um conjunto específico de instruções para completar uma tarefa em particular, a máquina é treinada usando uma quantidade grande de dados e algoritmos que dão e ela a habilidade de aprender como executar a tarefa [9].

O aprendizado de máquina é tradicionalmente dividido em três tipos diferentes, sendo eles: aprendizagem supervisionada, aprendizagem não supervisionada e aprendizagem por reforço. Ainda é possível usar abordagens de otimização estocástica como computação evolucionária para aprendizado. Todas esses tipos são candidatos viáveis para treinar uma DNN para jogar [10].

### Aprendizagem Supervisionada

Quando se trata de ANNs, um agente aprende por meio de exemplos. Durante seu treinamento, o agente deve gerar um resultado, sendo este comparado com uma resposta correta previamente conhecida. Após a decisão do agente, uma função de erro é usada para determinar a diferença entre a resposta fornecida e a resposta correta. Esta função é usada com o objetivo de gerar custo, o agente então atualiza seu modelo. De forma geral, o objetivo de um agente que aprende com aprendizagem supervisionada é obter um modelo que generalize suas regras para que funcione em dados que não foram usados durante seu treinamento. A aprendizagem supervisionada requer um amplo conjunto de dados para treinamento do agente.

A aprendizagem supervisionada atinge bons resultados em muitas classes de problemas. No entanto, precisa de muitos dados rotulados para que funcione de forma adequada. Entretanto, a rotulação de exemplos não é praticável em algumas situações. No entanto, para alguns jogos, o próprio jogo pode fornecer os dados para o agente, podendo ser um rastreamento de ações tomadas por humanos durante o jogo.

### Aprendizagem Não-Supervisionada

Diferentemente da aprendizagem supervisionada, a aprendizagem de máquina do tipo não supervisionada não compara nenhuma resposta previamente conhecida e correta com a resposta dada pelo próprio agente. O objetivo desse tipo de aprendizagem de máquina está em reconhecer padrões no conjunto de dados fornecidos para o agente.

Uma técnica bastante conhecida para um agente que aprende de forma não supervisionada é chamada de *autoencoder*, onde a ANN mapeia a entrada em si própria. Em outras palavras, é a função identidade. Esta ANN é construída em cima de duas partes chamadas: codificador que mapeia a entrada da ANN para uma representação oculta, e

decodificador que reconstrói a entrada a partir desta representação oculta. A função de erro se baseia no quão próximo a estrutura gerada pelo decodificador conseguiu chegar da entrada. A ideia é que a representação oculta seja de menor dimensionalidade que os dados de entrada, para que a RN aprenda a comprimir os dados e consequentemente aprender uma boa representação dos dados. No âmbito de jogos, as abordagens com aprendizagem não supervisionada normalmente são aplicados em conjunto com outros algoritmos [10].

#### Aprendizagem por Reforço

Este tipo de aprendizagem de máquina se baseia na interação do agente com o ambiente para que este possa construir um comportamento em cima de seu conhecimento. No caso de jogos, os agentes (jogadores) que têm um conjunto finito de ações para tomar em cada etapa e sua sequência de movimentos determina seu sucesso [10]. O agente tem um conjunto de ações finitas sendo a função de erro a recompensa, aplicada em cima de uma ou mais ações.

O ambiente no qual o agente se encontra inserido precisa fornecer uma ou mais recompensas, que podem ser recompensas frequentes e recompensas não frequentes. Outros fatores que podem influenciar são a mudança do ambiente condicionada na ação do agente, o passar do tempo e as mudanças no ambiente que são independentes do agente [10].

## 2.2 Aprendizagem por Reforço

A teoria da aprendizagem por reforço fornece uma explicação estabelecida por regra, baseada em perspectivas psicológicas e neurocientíficas do comportamento animal, de como agentes podem otimizar o controle de um Ambiente [8].

Para entender brevemente como funciona o aprendizado por reforço podemos imaginar um agente robô, que interage com seu ambiente por meio de ações. Quando uma ação é executada, o ambiente pode ou não sofrer alterações. Tanto o ambiente quanto o robô

possuem características que interpretram o estado do mundo. Uma vez que o estado é alterado, o robô deve reconhecer e interpretar se a mudança ocorrida foi benéfica ou não para ele. Esta interpretação é baseada na recompensa, ou penalidade, da ação executada. Ao interagir com o ambiente através de ações o agente tem duas fontes de retorno que são o estado e a recompensa. A Figura 2.1 demonstra este cenário.

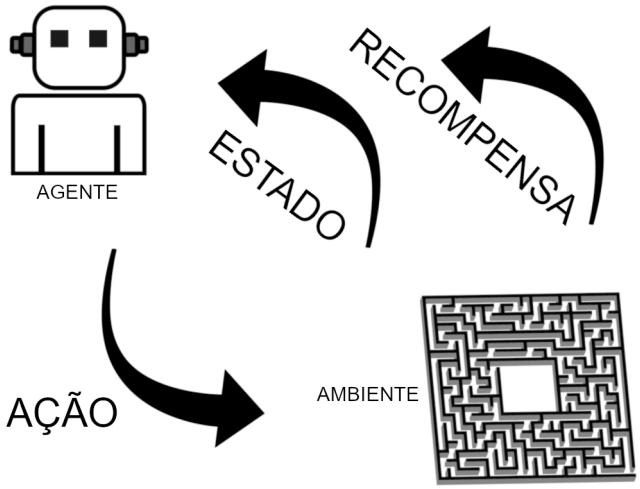


Figura 2.1: Relação entre agente e ambiente.

Considere um robô em um labirinto. O objetivo do robô é encontrar a saída do labirinto. As ações do agente baseiam-se em movimentos horizontais e verticais. Neste cenário o ambiente é estático, ou seja, não sofre modificações. O robô, por sua vez, verifica o estado que ele atingiu, averiguando as recompensas ou penalidades obtidas. Desta forma, a aprendizagem do robô dá-se à medida que ele caminha pelo ambiente e encontra diversas situações que afetam direta ou indiretamente seu estado e suas recompensas.

### 2.2.1 Aplicações

A RL pode ser aplicada em vários problemas como: agentes inteligentes para jogos eletrônicos [11], robótica [12], processamento de linguagem natural [13], visão computacional [14], entre outros [2]. Neste trabalho a aplicação está voltada ao desenvolvimento de um agente inteligente para um jogo eletrônico.

A aplicação de técnicas de IA em jogos eletrônicos é atualmente um campo de pesquisa estabelecido com várias conferências e periódicos dedicados [10]. Este ramo oferece um excelente campo de testes para a RL e a IA [2].

### 2.2.2 Q-Learning

O Q-learning é um modelo popular de RL. Seu método de aprendizagem baseia-se em recompensas por pares de ação-estado. Isso permite que o agente escolha uma ação, dado o estado. Para o Q-learning, uma política simples seria sempre executar a ação com o maior valor Q [10].

Este algoritmo é muito utilizado para RL, por se tratar de um dos pioneiros na área, mas seu diferencial está em que o algoritmo busca sempre a melhor política dado uma Finite Markov Decision Process (FMDP). Esta política é a relação que o algoritmo cria entre os inúmeros estados que o agente pode ter do ambiente e quais são as ações a serem tomadas mediante a situação.

Para qualquer FMDP, o algoritmo Q-learning encontra uma política que é considerada ótima no aspecto de que o algoritmo maximiza o valor da recompensa total sobre todos os passos sucessivos, a partir do estado atual. Este algoritmo pode identificar uma política ótima de seleção de ações para qualquer FMDP, desde que o tempo de exploração seja infinito, dentre outras restrições [15].

Um FMDP consiste em um conjunto finito de estados, cujas transições entre estados são dadas por uma distribuição de probabilidades [16]. Um FMDP possui ações que vão definir quais as probabilidades de transições entre os estados e ainda adicionar uma recompensa que servirá para que o agente tenha um *feedback* da ação tomada. A Figura 2.2 demonstra um exemplo de FMDP.

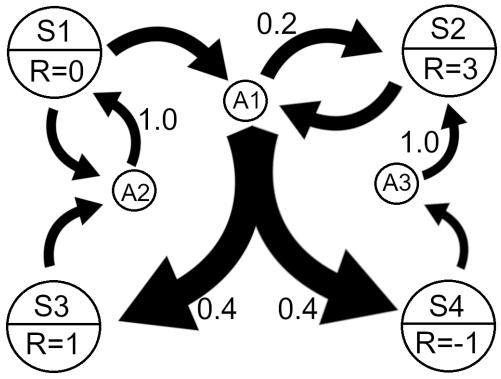


Figura 2.2: Exemplo de FMDP com quatro estados e três ações.

Entre os estados S há um conjunto de ações A para que as transições ocorram, e ainda um valor de probabilidade que significa a chance de migrar de um estado para um outro. Por fim são mostradas as recompensas R que indicam o quanto o agente ganha ao atingir cada estado. A Figura 2.2 pode ser representada em uma tabela, como a Tabela 2.1. Todas as transições a partir de todos estados S são mostradas.

| Estado Atual | Ação | Probabilidade | Estado Final | Recompensa |
|---|---|---|---|---|
| S1 | A1 | 0.2 | S2 | 3 |
| S1 | A1 | 0.4 | S3 | -1 |
| S1 | A1 | 0.4 | S4 | 1 |
| S1 | A2 | 1.0 | S1 | 0 |
| S2 | A1 | 0.2 | S2 | 3 |
| S2 | A1 | 0.4 | S3 | -1 |
| S2 | A1 | 0.4 | S4 | 1 |
| S3 | A2 | 1.0 | S1 | 0 |
| S4 | A3 | 1.0 | S2 | 3 |

Tabela 2.1: FMDP Tabular.

O aprendizado por reforço envolve um agente, um conjunto de estados S e um conjunto A de ações por estado. Ao executar uma ação qualquer dentro do conjunto A, o agente transita de um estado para outro, podendo inclusive transitar para o mesmo estado. Executar uma ação em um estado específico fornece ao agente uma recompensa R, que

normalmente é um valor numérico. É possível modelar um jogo, o mundo dele para um FMDP, facilitando o entendimento do jogo e comportamento do agente. Logo os estados do jogo variam de acordo com a ação do agente. Com o Q-learning as probabilidades de transação do FMDP são atualizadas, com o objetivo de maximizar a recompensa ao longo do tempo. Dessa forma, o processo de aprendizagem resume-se a atingir o objetivo do agente.

Em RL, o problema a ser resolvido é descrito como um Processo de Decisão de Markov (MDP). Os resultados teóricos em RL dependem da descrição do MDP como uma correspondência correta para o problema. Se o seu problema é bem descrito como um MDP, então a RL pode ser uma boa estrutura para usar para encontrar soluções. Isso não significa a necessidade de uma descrição completa do MDP (todas as probabilidades de transição), apenas que você espera que um modelo MDP possa ser feito ou descoberto.

## 2.3 Redes Neurais para Aprendizagem por Reforço

O conexionismo é uma das grandes linhas de pesquisa em IA e tem por objetivo investigar a simulação de comportamentos inteligentes por modelos baseados na estrutura e funcionamento do cérebro humano [17].

### 2.3.1 Artificial Neural Network

Uma rede neural é um processador massivo paralelamente distribuído constituído de unidades de processamento simples, que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para uso. Ela assemelha-se ao cérebro em dois aspectos [18]:

1. O conhecimento é adquirido pela rede a partir de seu ambiente e através de um processo de aprendizagem.

2. Forças de conexão entre neurônios, conhecidas como pesos sinápticos, são utilizados para armazenar o conhecimento adquirido.

Tradicionalmente, a aprendizagem em redes neurais funciona modificando os pesos sinápticos da rede de uma forma ordenada minimizando uma função de custo. Entretanto é possível que a própria rede neural altere sua topologia, o que aproxima mais ainda do cérebro humano [18].

Outra característica importante das ANN é a capacidade de generalização. Isto se refere ao fato de que uma ANN consegue produzir um resultado adequado de acordo com entradas previamente não usadas durante o treinamento. Em larga escala, para aproveitar deste poder, a ANN precisa de um conjunto de treinamento adequado e de técnicas de otimização adequadas.

Segundo [18], redes neurais oferecem as seguintes propriedades úteis:

1. **Não-linearidade**: Um nerônio artificial pode ser linear ou não-linear. Uma rede neural, constituída por conexões de nerônios não-lineares também é considerada não-linear. A não-linearidade é uma propriedade muito importante, particularmente se o mecanismo físico responsável pela geração do sinal de entrada for inerentemente não-linear como por exemplo, sinal de voz.

2. **Mapeamento de entrada-saída**: Esta propriedade está relacionada a problemas de aprendizagem supervisionada. Uma rede neural com pelo menos uma camada oculta é uma máquina de aproximação universal, ou seja, é capaz de representar qualquer mapeamento de entrada-saída, desde que hajam exemplos suficientes.

3. **Adaptabilidade**: Uma das propriedades mais chamativas das ANN é a sua capacidade inata de adaptabilidade. Podendo até mesmo serem configuradas para que os pesos sinápticos sejam atualizados em tempo real. Entretanto, esta propriedade quando não bem aproveitada ela pode acabar resultando em degradação de desempenho do sistema.

4. **Resposta a evidências**: Quando uma ANN está no contexto de reconhecimento de padrões, pode ser projetada para fornecer informações não somente sobre qual padrão particular ela deve selecionar, mas também sobre a confiança em cima da

decisão tomada. Assim, em sistemas onde a ambiguidade de padrões está presente, essa propriedade de resposta a evidencias pode desempatar e a ANN ainda aprende com isso.

5. **Informação contextual**: Dentro de uma ANN os seus nerônios podem estar conectados de maneira que eles podem ser afetados pela atividade de demais nerônios, de forma com que o conhecimento da rede se baseia na própria estrutura da rede e estado de ativação. Consequentemente a informação contextual é tratada de forma natural pela própria rede.

6. **Tolerância a falhas**: Parando para pensar em uma ANN implementada a nível de hardware, ao longo do tempo as conexões podem ser afetadas, degradadas e então a recuperação de um resultado desta rede pode ser comprometido. Felizmente devido a natureza distribuída da informação presente nas ANNs, para que a informação recuperada da rede seja afetado seriamente os danos presentes na rede devem ser extensos, caso contrário as poucas falhas geram um erro suave.

7. **Uniformidade de análise de projeto**: As ANN usam uma mesma notação em todos os domínios de aplicação. Os neurônios, de uma forma ou de outra, representam um ingrediente comum a todas as ANNs e tal uniformidade torna possível compartilhar teorias e algoritmos de aprendizagem em diversas aplicações de ANNs.

8. **Analogia neurobiológica**: Sabemos que o cérebro humano é uma prova viva, real de que o processamento paralelo tolerante a falhas não somente é possível, mas também rápido e poderoso. Para os engenheiros, a visão se volta para questões biológicas, afim de descobrirem novas técnicas para solucionar problemas mais complexos. Já para os neurobiólogos, estudar o funcionamento de ANNs pode gerar novas interpretações de fenômenos da própria neurobiologia.

Em uma ANN, o neurônio é a unidade capaz de processar uma informação e é fundamental para a operação de uma rede neural. A Figura 2.3 demonstra um dos modelos existentes de neurônios artificiais encontrados na literatura [18].

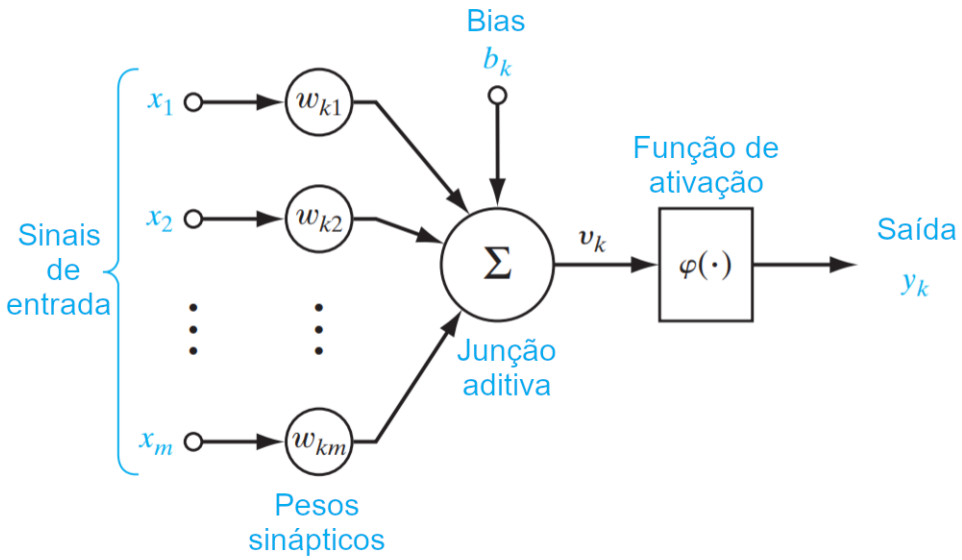


Figura 2.3: Modelo de Um Neurônio Artificial (adaptado de [18])

Os neurônios virtuais são constituídos por três componentes básicos:

1. **Conjunto de sinapses**, onde cada uma é caracterizada por um peso. Especificamente, um sinal $x_j$ na entrada da sinapse $j$ conectada ao neurônio $k$ é multiplicado pelo peso sináptico $w_{kj}$. É importante notar a maneira como são escritos os índices do peso terminal de entrada da sinapse à qual o peso se refere [18]. Diferentemente de uma sinapse encontrada no cérebro humano a sinapse artificial pode assumir valores positivos e negativos.

2. **Um combinador linear**, que é composto de um somador dos sinais de entrada, ponderados pelas respectivas sinapses do neurônio [18].

3. **Uma função de ativação** para restringir a amplitude da saída de um neurônio. A função de ativação é também referida como função restritiva já que restringe (limita) o intervalo permissível de amplitude do sinal de saída a um valor finito

[18]. Normalmente esse valor se encontra no intervalo unitário fechado $[0, 1]$ ou até mesmo $[-1, 1]$. Usualmente as funções de ativação são não-lineares e deriváveis.

Há ainda um outro fator interessante que aparece nos neurônios, o Bias que na Figura 2.3 é representado por $b_k$. O Bias influencia diretamente no campo local induzido ou potencial de ativação $v_k$ do neurônio $k$ e a saída do combinador linear $u_k$ é modificada [18]. Esta modificação altera o limiar de ativação do neurônio como apresentada na Figura 2.4.

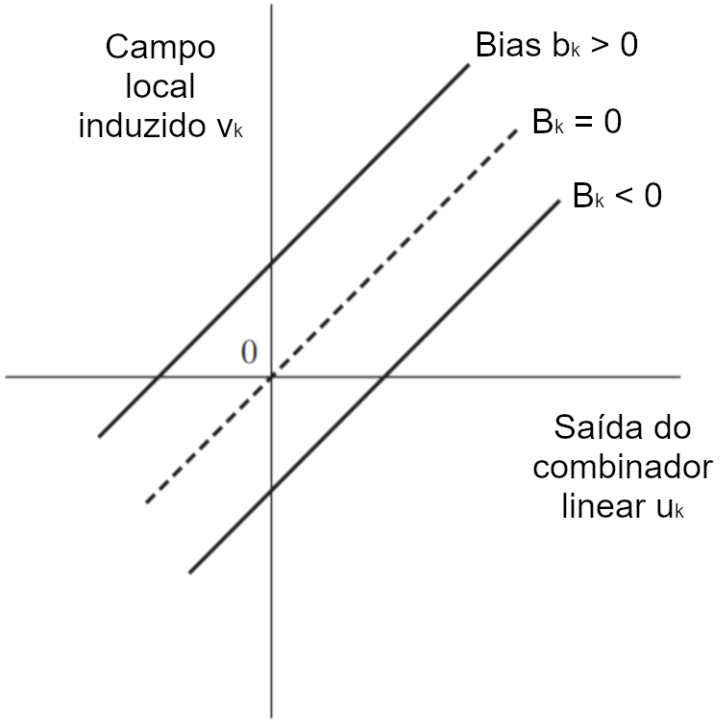


Figura 2.4: Representação gráfica do Bias.

A Figura 2.5 demonstra graficamente duas funções de ativação utilizadas em redes neurais. Dois exemplos comuns utilizados são [18]:

- Função de limiar, onde a saída do neurônio assume valor 1 se o campo local induzido é não-negativo, e 0 caso contrário. Esta função não é contínua, portanto não é derivável em todo seu domínio. Comumente usada em perceptrons;

- Função sigmoide, dada por $\frac{1}{1+e^{-\alpha x}}$. Esta função é derivável em todo seu domínio.

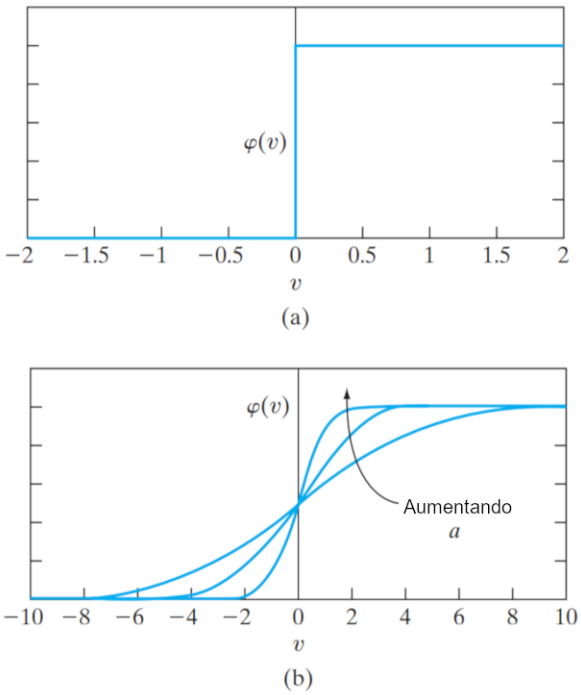


Figura 2.5: (a) Função Limiar (b) Função Sigmóide. (adaptado de [18])

Na literatura, existem vários problemas que foram resolvidos usando algum tipo de ANN, como: jogar StarCraft [6], Pac-man [3], jogos da plataforma *Atari* [11] entre outros [10].

### 2.3.2 *Deep Learning*

Nos princípios da inteligência artificial, o campo lidou e resolveu rapidamente problemas que são intelectualmente difíceis para os seres humanos, mas relativamente diretos para os computadores. Estes problemas podiam ser descritos por uma lista de regras formais e equações matemáticas. [19].

Uma vez descritos, isso permite que os computadores aprendam com a experiência e entendam o mundo em termos de uma hierarquia de conceitos, com cada conceito definido

por meio de sua relação com conceitos mais simples. Ao reunir conhecimento a partir da experiência, essa abordagem evita a necessidade de operadores humanos especificarem formalmente todo o conhecimento de que o computador precisa. A hierarquia de conceitos permite que o computador aprenda conceitos complicados, construindo-os a partir de conceitos mais simples. Se desenharmos um gráfico mostrando como esses conceitos são construídos uns sobre os outros, o gráfico é profundo, com muitas camadas. Por essa razão, chamamos essa abordagem de aprendizagem profunda, ou DL. [19]. A Figura 2.6 demonstra uma representação de comparação dentre os vários tipos de abordagem de IA inclusive a DL. Existem diferenças dentre as quatro disciplinas de IA, da entrada de dados até a saída cada disciplina possui um grau de aprendizagem de máquina (mostrada em cinza). (Adaptado de [19])

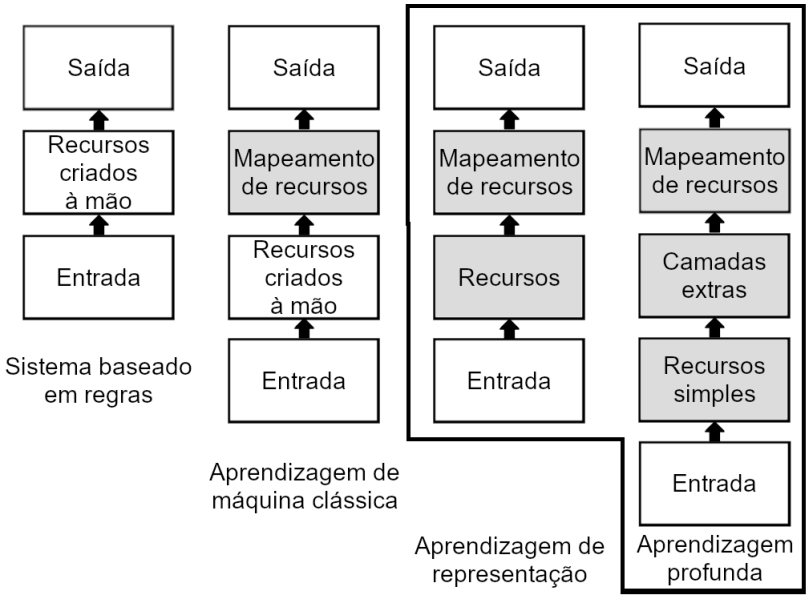


Figura 2.6: Representação de algumas disciplinas de IA.

Dentro da DL, temos as camadas extras que são responsáveis pelo processamento, entendimento de baixo-nível, como amostas de áudio e imagens, por exemplo. Na Figura 2.6 os quadrados mais escuros indicam os componentes aptos a aprender de acordo com os dados.

As DNNs aprendem representações automaticamente a partir de entradas brutas para recuperar composições hierárquicas em muitos sinais naturais, ou seja, recursos de nível mais alto são compostos de níveis mais baixos, por exemplo, em imagens, hierarquias de objetos e combinações locais de bordas. A representação distribuída é uma ideia central na DL, o que implica que muitos recursos podem representar cada entrada, e cada recurso pode representar muitas entradas.

A noção de treinamento fim-a-fim refere-se a um modelo de aprendizado que usa entradas brutas sem engenharia manual de recursos para gerar saídas, por exemplo, [20] com *pixels* brutos para classificação de imagens, [21] com sentenças brutas para tradução automática e Deep Q-network (DQN) [8], com *pixels* brutos e pontuação para jogar jogos.

A visão computacional tem sido tradicionalmente uma das áreas de pesquisa mais ativas para aplicações de DL, porque a visão é uma tarefa que não requer muito esforço para humanos e muitos animais, mas torna-se desafiadora para os computadores [22]. A maioria das pesquisas que envolvem visão computacional e DL não se concentram em aplicações tão exóticos que expandem o domínio do que é possível com imagens, mas sim sobre um pequeno núcleo de objetivos de AI voltados para a replicação de habilidades humanas. A maior parte do aprendizado para visão computacional é usada para reconhecimento de objetos ou detecção de alguma forma, quer isso signifique informar objetos presentes em uma imagem, anotando uma imagem com caixas delimitadoras ao redor de cada objeto, transcrevendo uma sequência de símbolos de uma imagem ou rotulando cada pixel uma imagem com a identidade do objeto a que pertence [19].

Normalmente as ANN modernas são implementadas em unidades de processamento gráfico. As unidades de processamento gráfico são componentes de hardware especializados desenvolvidos originalmente para aplicativos gráficos. O mercado consumidor para sistemas de jogos eletrônicos impulsionou o desenvolvimento de hardware de processamento gráfico. As características de desempenho necessárias para os bons sistemas de videogames também se tornam benéficas para redes neurais [19].

Os algoritmos de rede neural requerem as mesmas características de desempenho que os algoritmos gráficos em tempo real para processamento paralelo massivo e realização de

operações matriciais. As redes neurais geralmente envolvem grandes quantidades de parâmetros, valores de ativação e valores de gradiente, cada um dos quais devem atualizados durante cada etapa do treinamento [19].

Os algoritmos de treinamento de redes neurais normalmente não envolvem muita ramificação ou controle sofisticado, portanto são apropriados para hardware de Graphics Processing Unit (GPU). Como as redes neurais podem ser divididas em múltiplos "neurônios" individuais que podem ser processados independentemente dos outros neurônios na mesma camada, redes neurais beneficiam do paralelismo da computação da GPU [19].

Existem inúmeras outras aplicações para DL, alguns outros ramos na qual elas podem ser aplicadas podem ser conferidas em [2, p. 26-36] e [19, p. 438-481].

## 2.4 Deep Q-network

Quando [8] apresentou o novo conceito chamado: DQN, os métodos conhecidos de RL acabaram sendo impactados, pois essa nova abordagem tem suas peculiaridades. Para algumas aplicações ela apresenta um ganho significante de desempenho do sistema. A DQN junta implementações de Q-learning com ANN.

A abordagem que [8] propôs tem várias vantagens sobre o Q-learning, são elas:

- Cada etapa da experiência é potencialmente usada em muitas atualizações de peso, permitindo então uma maior eficiência de dados, dado que a rede tem uma pequena memória.

- A aprendizagem direta a partir de amostras consecutivas é ineficiente, devido às fortes correlações entre as amostras. O fator aleatório das amostras quebra essas correlações e, portanto, reduz a variação das atualizações.

- Os parâmetros atuais determinam a próxima amostra de dados na qual os parâmetros são treinados. Por exemplo, se a ação de maximização for mover para a esquerda, as amostras de treinamento serão dominadas por amostras do lado esquerdo; Se a ação de maximização for alterada para a direita, a distribuição de

treinamento também será alternada.

Ao usar o *replay* da experiência, a distribuição do comportamento é calculada pela média de muitos de seus estados anteriores, suavizando o aprendizado e evitando oscilações ou divergências nos parâmetros. O agente utiliza a política gananciosa para atualizar os valores Q.

## 2.5 Double Deep Q-network

A Equação 2.1 apresenta o alvo utilizado no algoritmo DQN tradicional. O termo $\gamma Q(S_{t+1}, \text{argmax}_a\, Q(S_{t+1}, a; \theta_t); \theta_t)$ indica que tanto a escolha da ação a ser executada quanto o valor estimado de Q para o cômputo do alvo dependem do mesmo conjunto e parâmetros (pesos) $\theta_t$ da rede neural. Isto faz com que seja mais provável que os valores da função Q subjacente sejam superestimados [7]. Para evitar essas estimativas super optimistas, van Hasselt *et al* [7] propõem desacoplar a escolha da ação do valor utilizado para o cômputo do alvo.

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \max_a Q(S_{t+1}, a; \theta_t); \theta_t) \tag{2.1}$$

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, \max_a Q(S_{t+1}, a; \theta_t); \theta'_t) \tag{2.2}$$

A Equação 2.2 mostra a proposta de van Hasselt *et. al*. Note que agora o valor estimado de Q para o cômputo de alvo está parametrizado por $\theta'_t$, chamados de parâmetros do alvo. Como agora dois conjuntos de parâmetros são usados na rede neural, com efeito temos duas redes neurais, portanto a técnica é chamada de Double Deep Q-Network (DDQN). Note que a tomada de decisão ainda é feita com base nos parâmetros da rede sendo treinada (on-line) $\theta_t$. Os parâmetros $\theta'_t$ são atualizados a cada $\tau$ *replays* executados pela rede *on-line*. Neste caso, os parâmetros de $\theta'_t$ são sobrescritos com os parâmetros $\theta_t$ da rede *on-line*.

Mnih *et al.* [8] discutem que o alvo da otimização não é estacionário durante todo o treinamento do agente, uma vez que depende dos pesos da própria rede neural. Os algoritmos de otimização baseados em descida do gradiente, usados extensivamente no treinamento de redes neurais, supõem que o alvo da otimização é estacionário durante todo o processo de treinamento. Logo, o problema de otimização que usa o alvo mostrado na Equação Xa não é bem-definido. No entanto, ao congelar os parâmetros alvo $\theta'_t$ por $\tau$ *replays*, obtemos uma sequência de problemas de otimização bem-definidos. Isto ajuda na convergência e na estabilidade das soluções encontradas.

## 2.6 Ferramentas

Nesta secção apresentamos as ferramentas que foram utilizadas para a implementação deste trabalho.

### 2.6.1 Python

Python é uma linguagem de programação poderosa e fácil de aprender. Possui estruturas de dados eficientes de alto nível e uma abordagem simples, mas eficaz, da programação orientada a objetos. A sintaxe elegante do Python e a digitação dinâmica, juntamente com sua natureza interpretada, o tornam uma linguagem ideal para *scripts* e desenvolvimento rápido de aplicativos em muitas áreas da maioria das plataformas. O interpretador Python é facilmente estendido com novas funções e tipos de dados implementados em C ou C ++ (ou outras linguagens que podem ser chamadas de C). O Python também é adequado como uma linguagem de extensão para aplicativos personalizáveis [23].

Linguagem de programação escolhida como base para desenvolvimento da IA, assim como o TD. Junto ao Python temos as bibliotecas que foram utilizadas para a condução deste trabalho.

### 2.6.2 Keras

Keras é uma biblioteca de rede neural, de código aberto escrita em Python. Ela é capaz de ser executada sobre o TensorFlow, Microsoft Cognitive Toolkit, R, Theano ou PlaidML [24]. Keras é a API de alto nível do TensorFlow 2.0. Uma interface acessível e altamente produtiva para resolver problemas de ML, com foco no moderno DL. Ele fornece abstrações e blocos de construção essenciais para o desenvolvimento e envio de soluções de aprendizado de máquina com alta velocidade de iteração [24]. Desenhado para capacitar a experimentação rápida de DNN, focado em ser de fácil uso, modular e extensível [25].

Esta biblioteca foi escolhida para orquestrar a ANN do agente implementado neste trabalho.

### 2.6.3 TensorFlow

*TensorFlow* é uma biblioteca *open-source*, uma interface para expressar algoritmos ML [26]. Usado neste trabalho para ser o *backend* da biblioteca *Keras*.

### 2.6.4 Pygame

Pygame é um conjunto de módulos Python projetados para escrever videogames. Permite criar jogos e programas multimídia com todos os recursos na linguagem Python. [27].

Biblioteca Python, utilizada para construção do TD deste trabalho.

### 2.6.5 Matplotlib.Pyplot

Matplotlib é uma biblioteca de plotagem para a linguagem de programação Python e a extensão matemática e numérica chamada Numpy. Provê uma API orientada a objeto para incorporar plotagens em aplicativos usando kits de ferramentas Graphical User Interface (GUI) de uso geral, tais como Tkinter, wxPython, Qt ou GTK [28].

O pyplot é uma interface baseada em estado para o matplotlib. Ele fornece uma maneira semelhante ao Matlab de plotagem [29].

Biblioteca para Python, usada para construir gráficos para melhor entendimento do rendimento e desenvolvimento da IA.

### 2.6.6 Scikit-learn

Biblioteca Python para ML construído sobre o *SciPy*. Ferramentas simples e eficientes para análise preditiva de dados [30].

Utilizado para realizar normalização de dados usados no *input* da IA.

## 2.7 Trabalhos Relacionados

Nesta seção são apresentados exemplos de sistemas que usam RL para treinar agentes inteligentes em jogos eletrônicos.

### How to match DeepMind's Deep Q-Learning score in Breakout

Neste trabalho o autor mostra experimentos com Deep Q-Learning para treinar agentes para os jogos *Pong* e *Breakout*. No caso do Pong, o agente atingiu seu primeiro ponto após 30 minutos de treinamento. Para o *breakout*, o autor apresentou modificações no trabalho de [8], atingindo 421 pontos. Estas alterações foram: usar a função de custo de Huber [11] e *thresholding* nos gradientes.

### Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning

Neste trabalho o autor apresenta três experimentos com Deep Q-Learning para treinar agentes para o jogo *Pac-man*. No primeiro caso, o agente utiliza uma DQN *vanilla* ou padrão, conseguindo percorrer pelo labirinto, entretanto com um alcance limitado dentro do labirinto. Já no segundo caso, agora utilizando uma rede *Double* Q-learning, o agente apresenta movimentos menos aleatórios segundo o autor. O autor salienta que o agente não consegue elaborar pequenas estratégias, o que demonstra uma satisfação parcial. Por

fim no último caso, o agente agora é equipado com uma rede *Noisy* de [31], apresentando modificações no comportamento e criação de estratégias.

### Deep Learning for Video Game Playing

Neste artigo de revisão, os autores apresentam inúmeras abordagens na literatura envolvendo DL e ANN. Várias categorias de jogos são abordados nesse trabalho, cada qual com explicação e exemplificação de agentes inteligentes que resolveram situações. Os agentes da maioria dos trabalhos envolvidos neste artigo tem como entrada *pixels*. Algumas dificuldades, apresentadas incluem: o que fazer quando o conjunto recompensa é esparso? Múltiplos agentes trabalhando em conjunto? Como será o futuro dos jogos? Tais questões são discutidas nesse artigo.

## 2.8 Sumário

Neste capítulo foram apresentados conceitos relevantes para compreender a teoria de aprendizagem por reforço e redes neurais. Além disto, foram apresentados três trabalhos que usam DQN para treinar agentes inteligentes para jogos eletrônicos. Neste contexto, todos os trabalhos apresentados utilizam pixels como representação de entrada para as redes neurais. No próximo capítulo são apresentados os modelos que serão utilizados para treinar agentes inteligentes para jogar Tower Defense utilizando metadados providos pelo próprio jogo.

# Capítulo 3

# Modelação

Este capítulo apresenta a trajetória da modelagem do jogo TD, assim como as diferenças entre as abordagens tradicionais e a abordagem utilizada neste trabalho. Também é abordado a modelagem do agente.

## 3.1 Tower Defense

Os Real-time Strategy (RTS) são jogos em que o jogador gerencia vários personagens ou unidades e o objetivo do jogo é prevalecer em algum tipo de conflito ou alcançar alguma conquista [10].

O principal desafio em jogos RTS é planejar e executar planos complexos envolvendo várias unidades. Em geral, esse desafio é significativamente mais difícil do que o desafio do planejamento em jogos de tabuleiro clássicos, como o xadrez, principalmente porque várias unidades devem ser movidas a qualquer momento e o espaço de estados é normalmente enorme. O horizonte de planejamento pode ser extremamente longo, onde as ações tomadas no início de um jogo afetam a estratégia geral. Além disso, existe o desafio de prever as jogadas de um ou vários adversários, que possuem múltiplas unidades. RTS são jogos de estratégia que não progridem em turnos discretos, mas onde ações podem ser tomadas a qualquer momento. Os jogos RTS adicionam o desafio da priorização de tempo aos desafios já substanciais de jogar jogos de estratégia [10].

Os jogos RTS são usados em trabalhos de aplicação IA com abordagem de DL [32]. O TD é um subgênero RTS de jogo. Esta categoria de jogo normalmente consistem em uma trilha, caminho por onde alguns inimigos se movimentam, e, em torno dessa trilha, o jogador pode posicionar algumas torres, construções que são responsáveis por causar dano aos inimigos. Na maioria das vezes existem ondas de inimigos, grupos de inimigos que tentaram completar todo o trajeto. Se um inimigo consegue atingir seu objetivo de chegar ao fim do trajeto, o jogador costuma ser punido de alguma maneira, como perdendo pontos de vida, perdendo pontuação geral dentre outras mais formas de punição.

Nenhum dos trabalhos pesquisados com agentes para jogos RTS se propuseram a usar RL para jogar um TD.

### Modelagem Tower Defense

Modelar e construir o TD para um agente, deve-se primeiramente focar em como transferir os dados do jogo para o agente de forma que ela os utilize para realizar ações naquele ambiente. Portanto, a construção do TD tem o foco a facilitar o fornecimento de dados para o *input* do agente.

*Gym* é um kit de ferramentas para desenvolver e comparar algoritmos de aprendizado por reforço. Oferecendo suporte a todos os agentes de ensino, desde caminhadas até jogos como *Pong* ou *Pinball* [33]. Observando na literatura, temos alguns exemplos, [34], que usam a biblioteca para construção de um agente para aprender a interagir e jogar um jogo. Assim, [35], descreve um caso onde o autor constrói não só o agente, mas o jogo em si.

O planejamento do TD se tornou mais claro ao analisar a literatura da área. Foi possível observar/identificar que os dados necessários que o agente precise para jogar são flutuantes. Por isso foi modelado e construído o TD de maneira que um dado poderia ser facilmente manipulado e inserido no *input* do agente.

## 3.2 Abordagem Tradicional

Autores como [3], [5] apresentam em seus trabalhos uma característica em comum: o processamento dos *pixels* da tela. No entanto, não são todos os trabalhos que utilizam como entrada os *pixels* como é o caso do agente criado para o jogo StarCraft, como exemplo, [6], [36].

Diversos trabalhos usam jogos de *Atari* para explorar aprendizagem por reforço em jogos. Nestes trabalhos os agentes tornam-se muitas vezes até melhores que jogadores humanos profissionais, sem que haja necessidade de codificar explicitamente a lógica do jogo e suas regras [4]. Em outras palavras, o agente aprende sozinho apenas olhando os *pixels* do jogo, a pontuação e a habilidade de escolher uma ação (ativar botões de um controle) assim como qualquer jogador humano seria capaz de fazer [4].

Os modelos das transições do estado do jogo podem ser organizados em um FMDP. O jogador olha para a tela do jogo e escolhe os diferentes botões e posições do *joystick* em um esforço para aumentar sua pontuação [3]. Entretanto, por mais que essa tomada de decisão aparenta ser simples, ela esconde a difícil tarefa que qualquer jogador que já jogou um jogo provavelmente notou, que é o processo de tomar decisões instantâneas com base em uma das combinações de *pixels* entre uma grande quantidade de combinações de *pixels* que podem ser mostrados na tela a qualquer momento. Além disso, há também a chance de que o jogador encontre situações novas, devido o enorme número de possibilidades.

Além disso, os ambientes dos jogos são parcialmente observáveis. Isto porque o jogador é forçado a fazer escolhas baseadas em uma representação indireta do jogo (a tela) ao invés de conhecer os parâmetros que governam a lógica do jogo. Desta forma, o jogador não conhece totalmente o ambiente o qual está inserido. Além disto, as ações a serem tomadas pelo jogador também são indiretas, pois as decisões do jogador estão em termos de botões a serem pressionados, não de ações semânticas no jogo [3].

Então, o desafio é complexo, mas definível: como usamos a nossa experiência de jogar um jogo para tomar decisões que aumentam a pontuação e generalizamos esse processo de tomada de decisão para novas posições em que nunca estivemos antes [3]?

Os DQNs aproveitam das ANNs para resolver esse desafio. A arquitetura pode ser dividida em duas partes. Primeiro, uma série de camadas convolucionais aprende a detectar características cada vez mais abstratos da tela de entrada do jogo. Em seguida, um classificador denso mapeia o conjunto desses recursos presentes na observação atual para uma camada de saída com um nó para cada combinação de botões no controle [3]. Uma melhor representação desse processo é mostrado na Figura 3.1. A entrada consiste nos *pixels*, que são mapeados em características sensoriais que sumarizam o que está sendo percebido na tela por meio de camadas convolucionais. Estas características são usadas por camadas posteriores que realizam o processamento e escolhem as ações a serem tomadas.

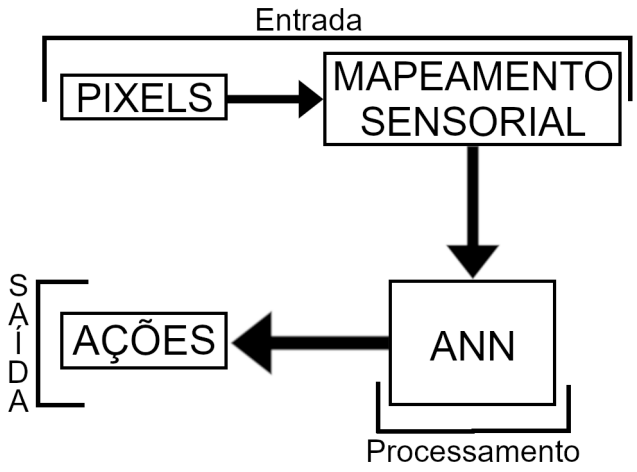


Figura 3.1: Modelo Simplificado de um agente.

Algumas outras abordagens encontradas na literatura não recolhem informações da tela para serem interpretadas antes de processadas. Essas abordagens incomuns, entretanto são desejáveis em algumas situações.

A Figura 3.2 demonstra um exemplo da curva de aprendizagem obtido por [3] durante o treinamento de seu agente. O agente de forma geral tende a sempre otimizar a taxa de recompensa. Fonte: [3].

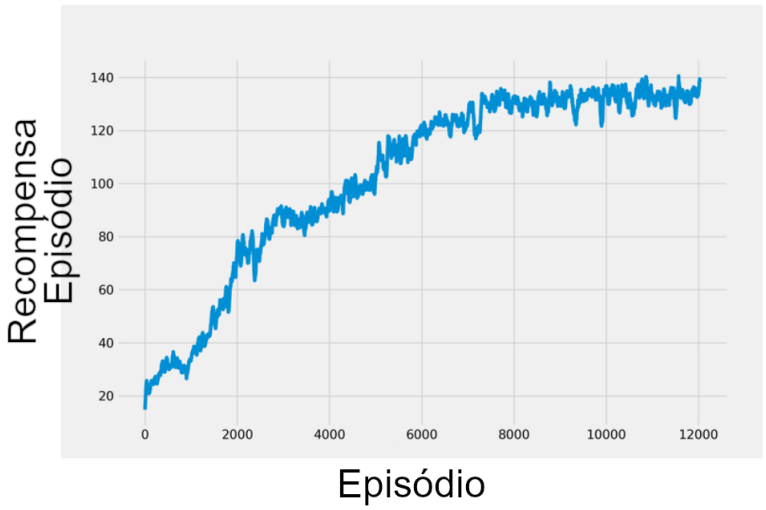


Figura 3.2: Relação de recompensa e o número de episódios que o agente teve durante seu treinamento.

De forma geral a IA aplicada ao campo de aprendizagem tendem a seguir o mesmo comportamento crescente da relação de recompensa da Figura 3.2.

Neste trabalho propomos uma arquitetura simplificada para o agente. Ao invés de receber *pixels* como entrada, propomos a entrada de metadados no sistema. Estes metadados consistem em informações relevantes proporcionadas pelo próprio jogo. Desta forma, o mapeamento sensorial mostrado na Figura 3.1, que depende do processamento de uma grande quantidade de *pixels*, é substituído por dados que são gerados pelo próprio jogo, diminuindo significativamente o poder computacional necessário.

## 3.3 Modelo proposto

O aumento da resolução das telas dos computadores e consoles faz com que o treinamento de um agente cuja entrada da rede são *pixels* se torna mais lento. Os jogos *Atari* são executados em uma janela com resolução muito baixa, quando comparados com as resoluções 4K presentes nos jogos de hoje. Sendo assim, a primeira camada de neurônios fica

imensamente maior em altas resoluções, tornando o processo de aprendizagem mais lento.

Seguindo a mesma linha de raciocínio de jogos RTS apresentado por [10], pode-se considerar que os dados de entrada são os recursos do jogo e outros componentes de alto-nível, não *pixels* na tela. Desta forma, é vantajoso usar esses tipos de dados, uma vez que as informações importantes estão todas disponíveis e o volume de dados é bem menor para ser avaliado em tempo real.

Escolher a categoria do jogo já foi de interesse particular, jogos RTS sempre chamaram muita a atenção. Devido tamanha complexidade e riqueza na variedade de ações que o jogador consegue tomar. O TD é uma subcategoria de jogos de estratégia em tempo real, que tem sua complexidade diretamente ligada ao tempo de jogo, tornando uma experiência desafiadora para quem joga esse tipo de jogo. Servindo de motivação para criar um agente capaz de jogar um TD.

Para este modelo proposto, possíveis metadados escolhidos foram: *Score*, inimigos na tela, quantidade de dinheiro disponível, quantidade de vidas restantes, posições de construção das torres. A saída foi composta por ações como: construir uma torre em um determinado lugar, não fazer nada, evoluir torre, destruir torre. A Figura 3.3 demonstra como ficou o modelo proposto.

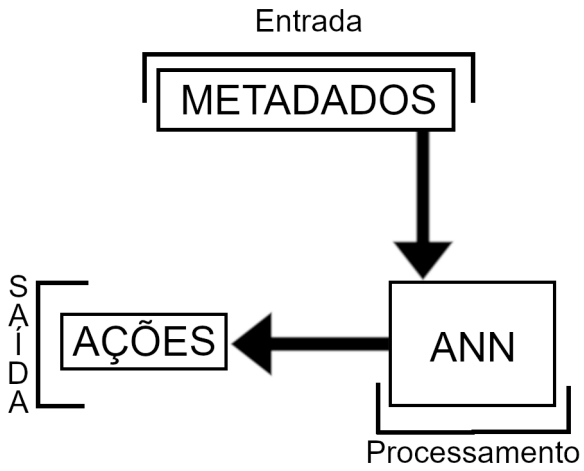


Figura 3.3: Modelo proposto. Observe que não temos mais na entrada os *pixels* e o mapeamento sensorial, apenas os metadados fornecidos pelo jogo.

Esta aplicação possui outro diferencial dos outros trabalhos de RL para jogos. Jogos TD não há uma representação real do agente no mundo, ou seja, não há um avatar representando o agente no mundo se movendo de um lado para o outro, realizando disparos em inimigos. Somente os resultados das ações tomadas pelo agente no ambiente são mostrados. Possibilitando observar quais são os locais que o agente construirá as primeiras torres, quais delas ele deverá melhorar ou até mesmo destruí-las. Serão apresentados apenas o resultado das estratégias que o agente inteligente realizará para atingir o objetivo final de sobreviver as ondas de inimigos.

## 3.4 Estrutura do software

O software desenvolvido foi separado em dois grandes módulos, o primeiro sendo o próprio TD e o segundo sendo o agente. Sendo um RTS, as ações tomadas deveriam ser em tempo de execução do jogo, fazendo com que a execução do agente fique simultaneamente ao jogo.

Para facilitar no desenvolvimento do todo o *software*, o uso de técnicas Object-oriented programming (OOP). Tornando mais rápido e fácil a mutação do código, a alocação e remoção de novos métodos e atributos, eventualmente usados no jogo.

O TD pode ser dividido em algumas partes onde cada uma dela tem sua função, a primeira parte do *loop* é responsável pelo reconhecimento das teclas pressionadas, logo após vem todo o controle dos inimigos e das torres existentes no jogo e por fim o agente faz todos seus controles, inclusive treinar a rede, gerar um *output*.

Todo e qualquer dado que seja proveniente do TD pode servir como entrada para o agente. Entretanto, usar dados demais para o *input* da rede pode fazer com que o processo de aprendizagem do agente fique mais lento. Porem, dados de menos pode ser que o agente nunca consiga convergir já que ele não possui informações suficientes para conseguir realizar seu propósito.

A variação do *input* da rede foi um processo comum neste trabalho. Não havia alguns valores, ou ideias de como seria o *input* ideal para a rede.

Diversas modificações foram feitas nas camadas ocultas da rede. Algumas vezes

fazendo-a mais profunda e outras vezes mais raso. Assim como o *input*, a estrutura da rede sofreu inúmeras modificações.

O modelo do agente, seja no *input*, as camadas escondidas ou o *output*, foram testadas diversas combinações dentre elas. Sempre buscando um agente que apresentasse resultados promissores.

Algumas informações são coletadas durante o final de cada jogo e então são gerados gráficos, para facilitar o acompanhamento do desenvolvimento do agente. Podendo observar se o agente está conseguindo ou não jogar o TD.

## 3.5 Sumário

Neste capítulo foi apresentado a modelagem do TD, além do diferencial que este trabalho possuí em relação à abordagem tradicional de agentes que jogam jogos eletrônicos. Utilizando os metadados como entrada para a rede neural ao invés de *pixels*. No próximo capítulo são apresentados o desenvolvimento e implementação dos agentes inteligentes.

# Capítulo 4

# Desenvolvimento e Implementação

Este capítulo apresenta o projeto e a implementação do jogo e do agente inteligente, conforme a modelagem apresentada no Capítulo 3. Também são apresentados os resultados de experimentos preliminares do desenvolvimento do agente.

## 4.1 Projeto do Tower Defense

Inicialmente o TD foi planejado com inúmeros recursos, como melhorias globais para as torres, aplicação de lentidão nos inimigos, chance de realizar ataques que causariam o dobro ou triplo de dano e um meio de recuperar vida.

Durante o processo de desenvolvimento do jogo, notou-se que vários conceitos previstos inicialmente deveriam ser abandonados em função do aumento da diversidade de jogadas, por isso o jogo foi simplificado.

Em um jogo TD a atribuição de vidas para o jogador é um meio de fazer com que o jogo tenha um fim. Todos os inimigos possuem um número de dano que é descontado da vida, caso eles cheguem ao final da trilha. Neste caso, o agente tem um valor de 20 vidas o qual atinja zero o jogo acaba.

No modelo construído para este trabalho existe apenas uma onda infinita de inimigos que gradualmente são fortalecidos.

Para que o nível de dificuldade seja aumentada de maneira progressiva, existem dois

mecanismos no jogo. O primeiro consiste em aumentar a quantidade de vida dos novos inimigos gerados conforme o tempo passa. O segundo consiste em desbloquear mais tipos de inimigos conforme a progressão do agente no jogo. Inicialmente somente *slimes* são gerados e, conforme o passar do tempo, os demais inimigos são desbloqueados oferecendo novos desafios ao jogador.

Algumas definições que foram utilizadas para implementação do jogo são:

1. **Inimigos:** responsáveis pelo fator dificuldade do jogo. Cada inimigo possui informações como: a quantidade de dinheiro que ele fornece ao morrer, a quantidade de pontos que ele fornece ao morrer, a quantidade de vida que ele retira ao atravessar o caminho definido, a quantidade de vida que ele possui e velocidade na qual o inimigo se movimenta. Foram criados seis tipos de inimigos onde cada um possui valores de atributos diferentes.

2. **Trilha/Caminho:** onde os inimigos percorrem, a partir de um ponto inicial até um ponto final.

3. **Agente:** controla a construção, melhoria e venda das torres. Cada uma dessas ações tem consequências que podem não ser imediatas, aumentando o desafio do jogo. O agente deve tomar ações considerando o que ele observa no ambiente. O objetivo do agente é atingir a maior pontuação possível antes de perder todas suas vidas.

4. **Torres:** responsáveis por derrotar que os inimigos que estejam percorrendo o caminho a sua volta. Cada torre tem uma posição adjacente ao caminho. Os tipos de torres diferentes são caracterizados por informações como o dano causado aos inimigos por cada ataque que efetua, o alcance de cada ataque, o preço para comprá-la, preço para fazer uma melhoria e a velocidade na qual ela executa um ataque. Foi adicionado às torres a função de melhoria, onde o alcance de acerto e o dano é aumentando. Para realizar estas melhorias um valor em dinheiro tem que ser pago.

Também é possível que as torres sejam vendidas, retornando parte do dinheiro investido nelas.

5. **Pontuação:** representa a capacidade do agente ao jogar o jogo. Quanto maior a pontuação, melhor o agente conseguiu lidar com as situações apresentadas durante o jogo. A pontuação é incrementada ao abater os inimigos ao longo do jogo.

6. **Dinheiro:** recurso para construção e melhoria das torres. O dinheiro é adquirido ao abater inimigos ou vendendo torres.

7. **Vidas:** número que representa a saúde do agente. Condição de término do jogo.

Por se tratar de um jogo de estratégia em tempo real, um equilíbrio é necessário para que o jogo seja divertido e ofereça um desafio justo. Portanto, houve uma fase de calibração dos parâmetros do jogo em busca deste equilíbrio. Abaixo são apresentados os valores que proporcionam uma experiência desafiadora, porém justa. Esta etapa foi concluída logo após o desenvolvimento da versão gráfica do jogo, que é detalhada na Seção 4.2. Nesta versão há a possibilidade de um jogador humano controlar o agente. Portanto, o jogo foi calibrado de forma que possibilite um desafio ao humano, mas que seja passível de pontuação considerável.

A projeção dos inimigos baseou-se em misturar alguns atributos. Cada inimigo possuí uma característica única, certas combinações na geração dos inimigos criariam situações desafiadoras. Por exemplo, se alguns inimigos que possuem muita vida e locomoção lenta forem gerados simultaneamente e em seguida vários inimigos rápidos e fracos forem gerados, poderiam se aproveitar para chegar até o final do caminho. Uma vez que as torres estariam com o foco em atacar os inimigos lentos que entraram primeiramente no alcance da torre. A Tabela 4.1 demonstra os números dados aos inimigos do TD.

| | Dinheiro | Pontuação | Dano | Vida Base | Velocidade |
|---|---|---|---|---|---|
| **Slime** | 3 | 5 | 1 | 10 | 1 |
| **Scorpion** | 10 | 15 | 2 | 20 | 1.25 |
| **Skeleton** | 5 | 15 | 2 | 15 | 1.5 |
| **Orc** | 15 | 30 | 3 | 30 | 0.75 |
| **Golem** | 25 | 55 | 5 | 50 | 0.5 |
| **Flyer** | 5 | 15 | 1 | 15 | 2 |

Tabela 4.1: Relação dos atributos de cada inimigo no TD.

A definição da quantidade e das propriedades das torres foi dada sempre buscando equilíbrio em relação aos inimigos. Sendo uma torre equilibrada em seus atributos e as demais com um atributo com valor alto e um outro valor baixo. Desta forma, para cada torre existiam cenários na qual ela se enquadrava de maneira mais eficiente. Por exemplo, uma torre com alcance curto, mas com velocidade de ataque alta, tem seu melhor cenário quando ela se encontra ao lado de muitos caminhos. Com isto o agente possuía diversas estratégias a serem exploradas, apenas com a combinação entre as torres e suas posições.

A Tabela 4.2 apresenta os parâmetros utilizados pelas torres.

| | Dano Base | Alcance | Preço Compra | Tempo de Ataque |
|---|---|---|---|---|
| **Basic** | 6 | 125 | 20 | 1 |
| **Mortar** | 12 | 200 | 30 | 3 |
| **Repeater** | 2 | 100 | 50 | 0.25 |

Tabela 4.2: Relação dos atributos de cada torre no TD.

Os dados apresentados na tabela 4.2 são apenas dos valores de base para cada torre. Realizar a melhoria na torre faz com que alguns desses parâmetros sejam atualizados, com intenção de possibilitar que o jogo fique mais equilibrado para o lado do agente. A Tabela 4.3 apresenta os dados que são acrescentados para cada melhoria feita em cada tipo de torre. Todas as torres têm um limite máximo de dez melhorias.

| | Dano | Alcance | Preço Adicional |
|---|---|---|---|
| **Basic** | 6 | 5 | 10 |
| **Mortar** | 12 | 15 | 15 |
| **Repeater** | 2 | 5 | 25 |

Tabela 4.3: Atributo adicional ao realizar uma melhoria na torre.

O mapa é responsável por gerar inimigos novos. O valor para essa geração foi definida em 1,5 segundos para a versão gráfica e 1,5 épocas para a versão simulador. Além de gerar novos inimigos o mapa também é responsável por aumentar a dificuldade do jogo de maneira gradativa. A dificuldade do jogo é aumentada a cada 30 segundos ou 30 épocas.

## 4.2 Implementação do *Tower Defense*

A primeira versão desenvolvida do jogo possui interface gráfica. Possibilitando tanto o agente quanto um humano jogar o jogo. Também durante esta versão foram feitas inúmeras calibrações dos parâmetros, baseadas em observações do jogo.

A Figura 4.1 demonstra a tela do jogo onde pode ser observado que existem oito pontos cinzas e três deles já estão ocupados com torres. Estes são os possíveis pontos nos quais o jogador pode construir as torres. O número de pontos e as posições das torres foram mantidas em todos os experimentos.

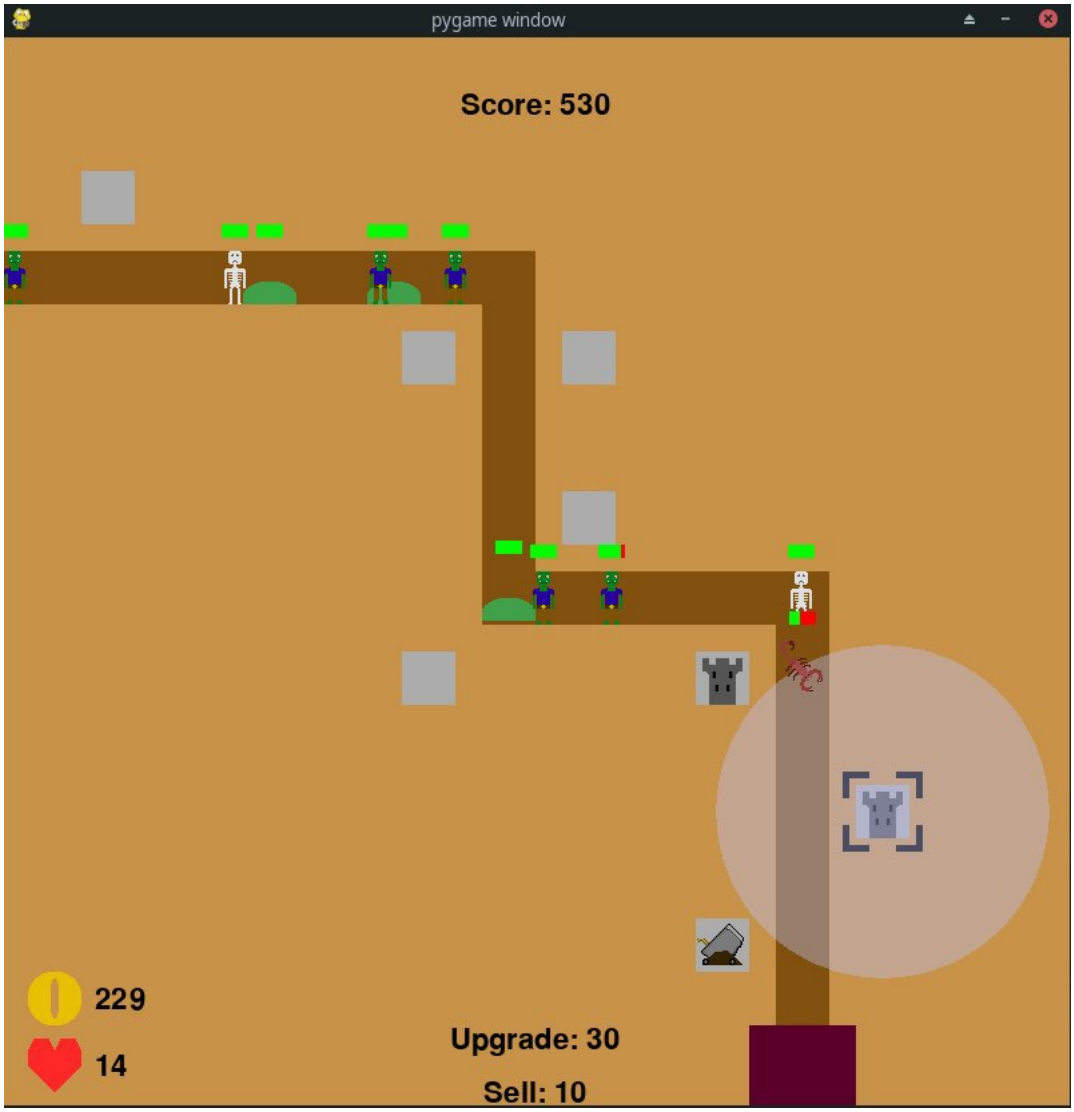


Figura 4.1: Primeira versão do jogo, com interface gráfica.

Em testes preliminares foi observado que cada partida do jogo com interface gráfica demorava para terminar, o que impediria a execução de uma quantidade apropriada de testes. Tipicamente agentes treinados por RL em ambientes de jogos necessitam de uma várias iterações para convergirem [11]. Portanto, foi necessário buscar uma forma eficiente de simular a execução do jogo, antes de proceder com o desenvolvimento do agente. As partidas demoravam para transcorrer em função da interface gráfica, mas também por conta dos eventos do jogo serem realizados baseando-se no relógio do mundo real. A

dependência do relógio foi necessária para que os humanos conseguissem jogar, possibilitando a observação e a calibração dos parâmetros, tornando o jogo equilibrado e passível de formulação de estratégias. Para tornar as execuções eficientes, o jogo foi reimplementado sem a interface gráfica. Também foi eliminada a dependência do relógio do mundo real do jogo.

Com esta alteração o TD pode ser visto como um simulador. Na versão gráfica, a temporização dos eventos do jogo usava como base o relógio do mundo real. Por exemplo, novos inimigos apareciam a cada um segundo e meio do relógio do mundo real. Portanto, não importa o quão eficiente for a implementação, seria necessário esperar um segundo e meio para que um novo inimigo entrasse no jogo. Como todos os eventos do jogo estavam atrelados ao relógio do mundo real, isto fazia com que a execução de uma partida fosse demasiadamente longa. Embora a ideia do simulador seja não depender do relógio do mundo real, é necessário usar alguma medida de tempo para sincronizar os eventos que acontecem no jogo. Desta forma, chamamos cada iteração do jogo de Ciclo.

Para preservar o equilíbrio do jogo simulado com a calibragem obtida na versão gráfica, foi necessário encontrar uma maneira de mapear o tempo do relógio em ciclos. Para isto, criamos uma unidade chamada de Época, equivalente a um segundo do tempo de relógio, que consiste em 840 ciclos.

A transição do jogo gráfico para a simulação, teve que ser feita usando alguns dados como base. Estes dados foram retirados das velocidades dos inimigos e são: 0,5; 0,75; 1; 1,25; 1,5; 1,75 e 2. O primeiro passo para trabalhar com estes dados foi encontrar um valor comum entre todos estes dados e com este valor comum os cálculos futuros não resultariam em valores flutuantes. Para que todos eles se tornassem valores inteiros, multiplicou-se cada um deles por quatro, obtendo como resultado os valores: 2, 3, 4, 5, 6, 7 e 8. Com estes novos valores foi realizado um mínimo múltiplo comum dentre todos eles, resultando em 840, portanto a cada 840 ciclos temos uma época.

Na primeira versão do simulador, todos os ciclos eram executados. Em cada ciclo, varias verificações eram feitas para determinar se era tempo de disparar os possíveis eventos. Entretanto, na grande maioria dos ciclos não havia nenhum evento para disparar,

o que tornava as verificações exaustivas e ineficientes. Como consequência, o tempo de execução não diminuiu drasticamente em relação à versão gráfica do jogo.

Em uma segunda versão do TD sem interface, a ideia foi de pular os ciclos que não teriam eventos para serem executados. Para isto foi criado uma lista de execução, onde cada objeto do jogo calculava em qual ciclo o próximo evento seria disparado.

A fila de execução consiste em uma coleção de eventos ordenada pelo número de ciclos até sua próxima execução. O descritor de cada evento consiste em uma tupla contendo uma referência ao objeto, o número de ciclos até a próxima execução e o número de ciclos entre as execuções dos eventos. No exemplo a seguir, o próximo evento será disparado pelo objeto SLIME_1 em 10 ciclos. Isto quer dizer que não é necessário simular os próximos 10 ciclos pois nenhum evento acontecerá. Desta forma, SLIME_1 dispara seu evento e 10 ciclos são subtraídos do número de ciclos para a próxima execução de todos os objetos da lista. Além disto, SLIME_1 é reinserido na lista com 20 ciclos restantes até sua próxima execução, que corresponde ao seu período.

```
Lista antes executar SLIME_1:
[[ SLIME_1, 10, 20 ], [ SCORPION_1, 15, 15 ], [ BASIC, 420, 840 ]]
Lista depois de executar SLIME_1:
[[ SCORPION_1, 5, 15 ], [ SLIME_1, 20, 20 ], [ BASIC, 410, 840 ]]
```

Experimentos preliminares mostraram que o tempo de execução das partidas foi reduzido drasticamente. Isto permitiu que testes extras fossem realizados no desenvolvimento do agente inteligente.

## 4.3 Desenvolvimento do Agente Inteligente

O desenvolvimento de um agente inteligente usando uma abordagem de RL é uma tarefa experimental. Esta abordagem requer um projeto cuidadoso dos componentes do sistema tais como entradas, saídas e do mecanismo de aprendizagem. Além disto, existem vários componentes no sistema que são parametrizáveis, o que faz com que sejam necessários

vários experimentos para determinar as melhores combinações de parâmetros. Nesta seção, são apresentados os componentes do agente inteligente bem como a exploração dos seus parâmetros.

Haviam inúmeras formas de extrair estes metadados e usá-los como entrada para o agente, como o modelo proposto para o agente, mostrado na Figura 3.3, qual teve seu *input* caracterizado por metadados retirados do TD. Sendo assim tivemos que testar várias combinações de metadados que serão apresentadas na sessão 4.3.1. A etapa seguinte foi a definição de um ANN cujo o mecanismo de aprendizagem deveria ser capaz de aproximar a função Q. Por fim, a etapa de saída, em que o agente poderia executar ações no jogo como: construção, melhoria e vendas de torres. Fizemos uma exploração buscando diversas formas de representar a saída, esta exploração é apresentada na próxima seção.

Em nossa arquitetura atualizamos os parâmetros da rede para estimar a função de valor diretamente da política exemplares de experiência, que são extraídos das iterações com o ambiente. A Equação 4.1 de [8] demonstra estes parâmetros.

$$e_t = (s_t, a_t, r_t, s_{t+1}) \tag{4.1}$$

Para arquitetura feita neste trabalho ainda adicionamos a experiência um parâmetro, $d$, para considerar o término do jogo.

Um dos desafios de agentes que usam DQN é que a rede neural usada tende a esquecer as experiências anteriores, pois as substituem por novas experiências [8]. Portanto, é necessário manter uma coleção de experiências e observá-las para treinar o modelo com essas experiências anteriores [34]. Esta coleção de experiências é conhecida como memória.

Todos os agentes implementados neste trabalho tiveram sua memória com tamanho definido de 2000. Cada exemplar de memória é composto por:

```
[state, action, reward, next_state, done]
```

*State* e *next_state* são as informações que o agente tem do ambiente no estado atual e o próximo estado, respectivamente. Cada estado representa as entradas do agente. *Action* é a ação que o agente realizou quando estava em *state*. *Reward* diz o quanto de recompensa

o agente acumulou entre *state* e *next_state*. *Done* é usado apenas para informar o término do jogo após esta ação ser realizada.

A DQN aproxima a função Q, que relaciona um estado e uma ação com a recompensa esperada. Para aproximar a função Q usando uma rede neural, uma função de erro a qual possa ser otimizada é necessária. Tal função relaciona o quanto que uma predição diverge do valor real da função. A Equação 4.2, proposta por [8], apresenta a função de erro utilizada no treinamento do agente. $s$ é o estado atual e $a$ é a ação realizada. $s'$ e o estado alcançado a partir da ação $a$ no estado $s$. $a'$ é a ação que maximiza a recompensa no estado $s'$. $r$ é a recompensa obtida entre $s$ e $s'$. A equação é o erro quadrático entre a predição Q(s,a), em relação ao valor da r + $\gamma$ Q(s',a'). Em outras palavras, representa a divergência entre o valor predito para a recompensa ao tomar a ação $a$ no estado $s$ em relação a uma aproximação precisa, obtida após a obtenção da recompensa e da previsão da recompensa no próximo estado.

$$\text{erro} = \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)^2 \tag{4.2}$$

Normalmente, a recompensa está atrelada à pontuação que o agente está obtendo, [7], [8], [34]. Neste trabalho seguimos esta mesma ideia. A recompensa foi considerada em três cenários. Sendo uma recompensa periódica, uma recompensa vinculada aos inimigos e por fim uma penalidade quando o agente perde o jogo. A recompensa ligada aos inimigos permaneceu durante todos os testes, enquanto as demais sofreram modificações ao longo dos testes.

### 4.3.1 Experimentos Preliminares

Seguindo a literatura [34], [3], [5], manteve-se os seguintes hiper-parâmetros iguais em todos os testes realizados.

- Camadas ocultas com ativação "*relu*".
- Camada de *output* com ativação "*linear*"

- Otimizador da rede, "*ADAM*".
- Equação de erro da rede, "*mse*", 4.2.

A literatura de RL para jogos mostra que o desenvolvimento de um agente inteligente é um processo experimental. Não há como ter certeza do desempenho dos agentes sem os avaliar no ambiente. Portanto, as modificações constantes no projeto de acordo com os resultados obtidos leva a um processo de desenvolvimento incremental. Nesta seção são apresentadas modificações realizadas no projeto dos agentes, bem como a exploração dos parâmetros tanto na versão gráfica, quanto na versão simulada do jogo.

Para facilitar a apresentação dos testes, adotamos uma nomenclatura para diferenciar os agentes desenvolvidos com a versão gráfica. Os agentes da versão gráfica são identificados com o prefixo VG. Os números indicam a ordem que os agentes foram desenvolvidos e avaliados.

### Experimentos Versão Gráfica

#### Hiper-parâmetros

A construção dos hiper-parâmetros do VG1 deu-se com base em buscas na literatura, [11] e [7]. VG1 foi nosso primeiro experimento na versão definitiva do TD, por isso a necessidade de buscar na literatura alguns dados para dar início aos nossos experimentos.

Estes hiper-parâmetros foram utilizados em todos os agentes criados nessa versão. Tendo $\epsilon$, um número indicando o limiar para que o agente decida fazer uma ação totalmente aleatória ou não, fator que determina a exploração do agente. Inicialmente o agente faz diversas ações aleatórias e, à medida que os jogos acontecem, este valor diminui. Os hiper-parâmetros são mostrados na Tabela 4.4.

| $\epsilon$ Decaimento | $\epsilon$ Min. | Taxa Aprendizagem |
|---|---|---|
| 0,995 | 0,01 | 0,001 |

Tabela 4.4: Hiper-parâmetros usados em todos experimentos da versão gráfica.

### Normalização

Analisando detalhadamente a respeito dos dados utilizados nos agentes da literatura, em [11], [7] e [10] observamos que não existia nenhum processo de normalização dos dados. Os valores usados já estavam na mesma escala, descartando a necessidade da normalização. Entretanto para este trabalho, este processamento se fez necessário uma vez que as entradas, discutidas adiante, estão em escalas distintas. Usamos *z-score* pra normalizar cada *batch* separadamente.

### Amostragem da memória

Seguindo a literatura, [7] e [8], as amostras utilizadas no treino dos agentes VG1, VG2, VG3 e VG4 são obtidas de posições aleatórias da memória dos agentes. Isso permite que o agente reutilize e aprenda com experiências passadas e não relacionadas, reduzindo a variação das atualizações [10].

### Atraso do *replay*

O *replay* só é executado quando a quantidade de experiências na memória excede o tamanho do *batch* de treinamento. O agente realiza o *replay* todas as vezes antes de tomar uma decisão.

### Saídas

Tivemos uma saída de tamanho oito que foi utilizada para todos os experimentos desta versão, com as seguintes características:

- Saída: 1. Corresponde a construir uma torre básica.
- Saída: 2. Corresponde a construir uma torre morteiro.
- Saída: 3. Corresponde a construir uma torre repetidora.
- Saída: 4. Corresponde a selecionar o próximo *spot* de torre.

- Saída: 5. Corresponde a selecionar o *spot* anterior de torre.
- Saída: 6. Corresponde a fazer melhoria na torre.
- Saída: 7. Corresponde a vender a torre.
- Saída: 8. Corresponde a não agir.

As ações que um humano podia executar no jogo são correspondentes à saída definida para o agente, permitindo equivalência e justiça para os jogadores seja ele humano ou máquina.

**Entradas**

As entradas que são passadas ao agente descrevem o ambiente atual que o mesmo deve considerar ao tomar suas decisões. A maioria dos agentes que usam DQN na literatura recebem como entrada as imagens pré-processadas dos *frames* de vídeo do jogo [8]. Nossa proposta utiliza metadados oferecidos pelo ambiente.

A Tabela 4.5 mostra os dados de entrada para todos os agentes avaliados na versão gráfica. Para o agente VG1, os dados estavam focados em refletir o estado do jogo como um todo. No agente VG2, os dados estavam focados em informações relacionadas com cada uma das 8 torres individualmente, ignorando o estado do resto do jogo. A Figura 4.2 mostra a evolução da pontuação ao longo das partidas. Após 650 partidas, a média máxima atingida pelo agente VG1 foi de 902,35 pontos, enquanto a pontuação máxima foi de 5455 pontos. O agente VG2 atingiu uma média máxima de 1415,2 pontos e sua pontuação máxima foi de 5610 pontos. Antes mesmo dos primeiros agentes jogarem o TD partidas com humanos jogando foram realizadas. A pontuação média atingida foi de 18680 pontos.

| Tamanho da Entrada | Dados da Entrada | | Agente |
|---|---|---|---|
| 20 | Geral | Vida, Dinheiro, Pontuação, Nível dificuldade, Torre selecionada (8), Quantidade de inimigos | VG1 |
| | Torre Selecionada | Nível torre (4), Tipo torre, Custo melhoria, Retorno venda | |
| 16 | Por Torre | Tipo da torre, Nível da torre | VG2 |
| 65 | Geral | Dinheiro | VG3 VG4 |
| | Por Torre | Inimigos abatidos por tipo (6), Tipo da torre, Nível da torre | |

Tabela 4.5: Entradas dos agentes da versão gráfica

Os resultados obtidos pelos agentes VG1 e VG2 não foram satisfatórios. A Tabela 4.5 mostra que VG1 e VG2 não tem informações sobre o desempenho das torres em relação ao número de inimigos derrotados até então. Isto impede que o agente possa tomar decisões baseadas na mudança do perfil de inimigos enfrentados por conta do aumento da dificuldade, por exemplo. Por isso outro conjunto de dados de entrada foi avaliado utilizado nos agentes VG3 e VG4. Eles incluem, por torre, a contagem de inimigos abatidos por tipo.

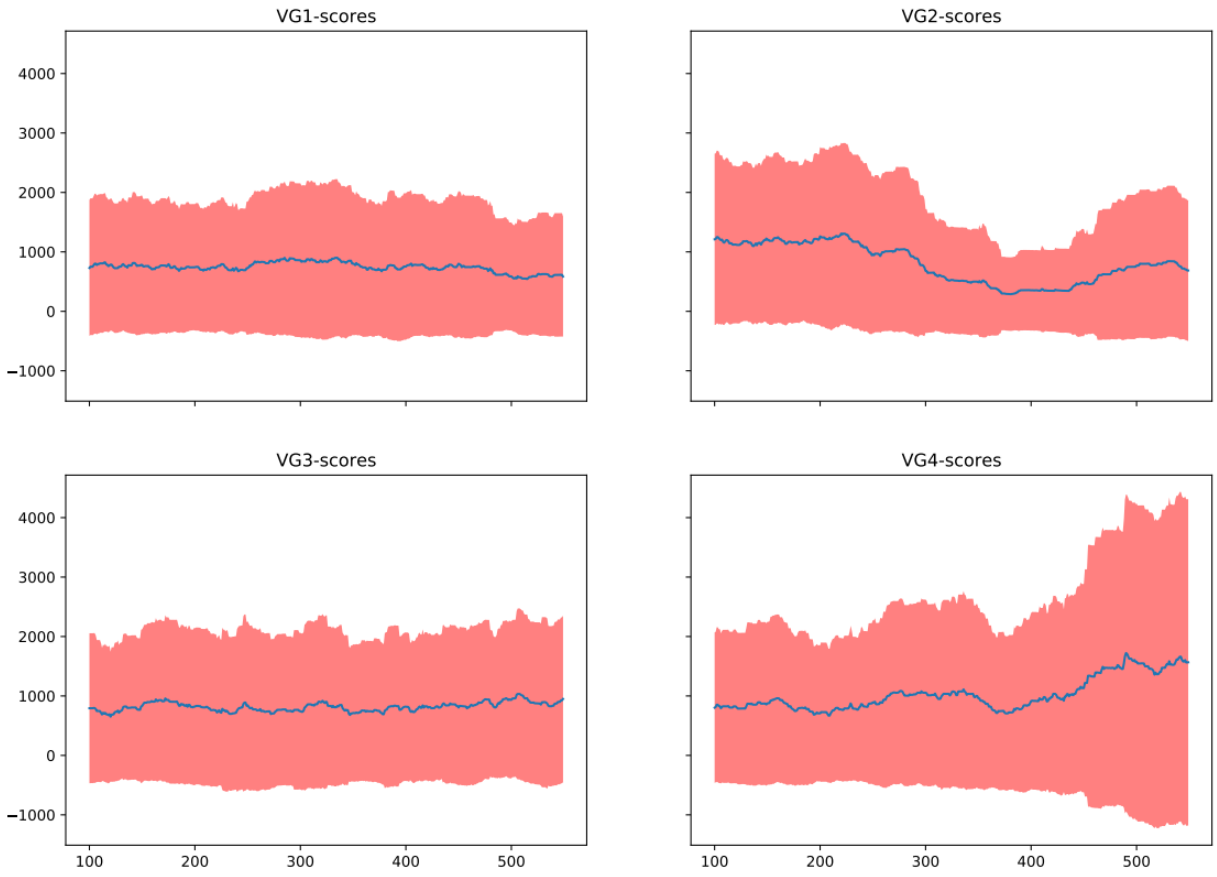


Figura 4.2: Gráfico da pontuação de todos agentes da versão gráfica com 650 partidas. A linha azul representa a média móvel máxima em 100 partidas e a área avermelhada representa o desvio padrão.

Após 650 partidas, os agentes VG3 e VG4 atingiram a média máxima de 1034,9 e 1722,15, respectivamente. A pontuação máxima atingida pelos agentes VG3 e VG4 foi de 8705 e 12220, respectivamente. A Figura 4.2 mostra que os agentes VG1 e VG2 não apresentam uma tendência para melhorar os resultados, enquanto VG3 e VG4 apresentam uma leve tendência para melhorar os resultados. Nota-se que todos os agentes que consideram a contagem de inimigos abatidos melhoraram a pontuação em relação aos agentes VG1 e VG2. Isto indica que considerar o desempenho das torres pode ser significativo nas tomadas de decisão dos agentes.

**Recompensas periódicas**

As recompensas fornecem ao agente retornos que significam o quão positivo foi realizar determinada ação. Agentes presentes na literatura, [7], fazem uma recompensação apenas no final do jogo. A Tabela 4.6 apresenta os tipos de recompensas usadas neste trabalho. Além da recompensa apenas no final do jogo, acrescentamos duas outras formas de recompensa. A recompensa periódica e a recompensa ao abater inimigos.

| Recompensa Periódica | Punição Perda | Agente |
|---|---|---|
| -1 | -50 | VG1<br>VG2 |
| 0 | -50 | VG4 |
| | 0 | VG3 |

Tabela 4.6: Recompensa periódica e punição ao perder, fornecida aos agentes da versão gráfica.

O TD é um jogo do gênero RTS, o agente deve tomar ações sempre que necessário para se adaptar aos desafios que variam conforme o tempo. Assim, uma punição (recompensa de -1) pode ser atribuída em todos *loops* do jogo. Isto tem como objetivo estimulá-lo a agir com frequência. Essa estratégia foi utilizada em ambos agentes VG1 e VG2.

Em certas situações decidir não agir é realmente a melhor ação a tomar. Desta forma, a punição pela inação do agente foi removida para os agentes VG3 e VG4.

Penalizar o agente por perder o jogo foi uma abordagem feita em três experimentos, VG1, VG2 e VG4. Com essa abordagem esperava-se que o agente interpretasse o fim do jogo como sendo ruim. No experimento VG3 decidimos não punir o agente quando ele perdesse o jogo. Após 650 partidas, a média máxima obtida foi de 902,35 pontos para VG1, 1415,2 pontos para VG2, 1034,9 pontos para VG3 e 1722,15 pontos para VG4. De acordo com estes resultados, teve-se um indicativo de que não punir o agente periodicamente poderia ser benéfico para a aprendizagem.

### Tamanho do lote de exemplos

O tamanho do lote de exemplos representa quantos exemplares o agente receberá de sua memória para realizar o treinamento. Assim como [7], decidiu-se iniciar os experimentos com seu tamanho definido em 32. A Tabela 4.7 apresenta os diferentes tamanhos do lote utilizados nos experimentos VG1, VG2, VG3 e VG4.

| Tamanho *batch* | Agente |
|---|---|
| 32 | VG1, VG2 |
| 64 | VG4 |
| 256 | VG3 |

Tabela 4.7: Tamanho do *batch* usado nos experimentos da versão gráfica.

Com os resultados de VG1 e VG2, decidiu-se aumentar o tamanho do lote. O tamanho do lote poderia ser insuficiente para o agente fazer seu treino. Por isso, os agentes VG3 e VG4 tiveram o tamanho do lote alterado. Aumentando então a quantidade de experiências que o agente usa para fazer seu treino. A média máxima de pontuação atingida pelos agentes VG1 e VG2 foi de: 902,35 e 1415,20 respectivamente. Para os agentes VG3 e VG4 teve-se essa média máxima de pontuação de 1034,90 e 1722,15. VG4 apresentou uma média máxima superior aos demais agentes, entretanto o mesmo agente apresentou um desvio padrão de 3508,18. A Figura 4.2 mostra o perfil do desvio padrão ao longo da partida para os quatro agentes. Em geral os quatro agentes possuem comportamento instável, no entanto VG4 é o mais instável entre todos. Descarto o lote de tamanho 64, a comparação foi feita entre VG3 com 256 no tamanho do lote contra VG1 e VG2 com 32 de tamanho de lote. VG3 teve como resultado uma média máxima entre VG1 e VG2, mas com sua pontuação máxima de 8705 contra 5455 e 5610 para VG1 e VG2 respectivamente. Estes resultados indicam que o tamanho de lote maior impacta significativamente na tomada de decisões do agente.

### Atraso na recompensa

Assim como em [3], [7] e [34], os agentes VG1 e VG2 receberam suas recompensas instantaneamente após suas ações. A Tabela 4.8 apresenta a relação do atraso para fornecer a recompensa ao agente.

| Atraso Recompensa | Agente |
|---|---|
| 0 | VG1, VG2 |
| 20 | VG3, VG4 |

Tabela 4.8: Atraso da recompensa, em segundos, utilizada nos experimentos da versão gráfica.

Para os agentes VG3 e VG4 a recompensa foi fornecida com atraso em relação a realização das ações. Todas as recompensas entre a ação e o atraso da recompensa foram acumuladas. Ao final do atraso, entrega-se ao agente a recompensa acumulada.

Após 650 partidas notou-se que os agentes VG3 e VG4 apresentavam um aumento no desvio padrão da pontuação, VG1 e VG2 com 2299,5 e 2411,49 para 2718,15 e 3518,08 em VG3 e VG4, um aumento considerável. O período de acúmulo de recompensa fez com que o agente memorizasse inúmeros experiências com recompensa negativa, já que a recompensa periódica fornecida aos agentes era de menos um, dificultando a aprendizagem do agente. Estes resultados mostram que o acúmulo de recompensa não surtiu efeito positivo para os agentes VG4 e VG5 em termos de estabilidade das pontuações alcançadas.

### Modelos da rede neural

Os modelos construídos para os agentes definem qual a sua profundidade e complexidade. A proposta não seguiu nenhum exemplo na literatura para construção dos modelos, uma vez que nossa abordagem baseada em metadados é uma das contribuições deste trabalho.

A Tabela 4.9 demonstra o número de neurônios nas camadas ocultas da rede, bem como o número de camadas ocultas. Para ambos agentes VG1 e VG2, o modelo da rede foi constituído por apenas duas camadas ocultas. As dimensões foram escolhidas de

acordo com a entrada da rede. O agente VG1 teve sua primeira camada oculta definida como o dobro da entrada. O agente VG2 teve essa definição aumentada para o triplo. Em ambos experimentos a segunda camada oculta teve seu tamanho igual ao da camada de entrada da rede.

| Neurônios nas Camadas Ocultas | Agente |
|---|---|
| [40\|20] | VG1 |
| [48\|16] | VG2 |
| [320\|320\|200] | VG3, VG4 |

Tabela 4.9: Camadas e número de neurônios nas camadas ocultas dos agentes da versão gráfica.

Tendo os resultados não satisfatórios para os agentes VG1 e VG2, decidiu-se expandir o modelo da rede. Aumentando significativamente a quantidade de neurônios e acrescentando mais uma camada oculta. Aumentando a quantidade de combinações, permitindo que a rede faça decisões mais complexas.

Depois de executar 650 partidas, a média máxima da pontuação a cada 100 partidas subiu de 902,35 do VG1 e 1415,20 do VG2 para 1722,15 do VG4. Enquanto o agente VG3 teve sua média máxima entre VG1 e VG2, com 1034,90.

### 4.3.2 Migração para versão simulada

A execução dos agentes VG1, VG2, VG3 e VG4 estava vinculada ao relógio do tempo. Portanto, o treinamento dos agentes estava muito demorado. Como a natureza de agentes baseados em aprendizagem por reforço requer um número mais alto de jogos para melhorar o desempenho dos agentes, foi implementada uma versão mais rápida. Note como a baixa velocidade das execuções com a versão gráfica, mostrada na Figura 4.2, não permite avaliar a evolução dos agentes por mais partidas. Esta nova versão é um simulador do jogo, que não possui interface gráfica e não depende do relógio do mundo real. Sendo

assim, como descrito na sessão 4.2, foram tomadas precauções para que a versão simulada do jogo seja equivalente em termos de parâmetros do jogo a esta versão inicial. Isto foi feito para manter o equilíbrio obtido durante a calibração do TD.

Os experimentos na versão simulada são apresentados e discutidos no Capítulo 5.

## 4.4 Sumário

Neste capítulo foram apresentados a projeção, a implementação do TD e o desenvolvimento dos agentes inteligentes utilizados como experimentos preliminares. A discussão dos resultados dos agentes VG1, VG2, VG3 e VG4 e a modificação dos parâmetros de acordo com os resultados coletados. No próximo capítulo são avaliados e discutidos os resultados de mais agentes implementados. Assim como as diversas modificações dos parâmetros, buscando otimizar os agentes.

# Capítulo 5

# Avaliação/Discussão

Este capítulo apresenta os parâmetros dos experimentos realizados com a implementação de agentes na versão simulada do Jogo. Também são apresentados os resultados e comparações dentre estes experimentos.

## 5.1 Configuração geral

Para facilitar a apresentação dos testes, adotamos uma nomenclatura para diferenciar os agentes desenvolvidos com a versão simulada. Os agentes da versão simulada são identificados com o prefixo DQN e DDQN. Os números indicam a ordem que os agentes foram desenvolvidos e avaliados.

Nestes experimentos extraímos informações extras dos agentes para facilitar os comparativos dentre eles. Na versão gráfica tinha-se apenas informações relacionadas a pontuação do agente. Já nesta versão, foram gravados o número de *replays* e o número total de ciclos executados de cada partida.

A adaptação da versão gráfica para a simulada permitiu a manutenção dos parâmetros do jogo, 4.2. A arquitetura das redes neurais usadas preservou, em relação à versão gráfica, as seguintes características:

- Camadas ocultas com função de ativação "*relu*".

- Camada de *output* com ativação "*linear*"
- Otimizador da rede, "*ADAM*".
- Equação de erro da rede, "*mse*", 4.2.

### Normalização

Assim como na versão gráfica, mantivemos a normalização *z-score*. Os parâmetros na normalização são atualizados *on-line*, a cada *batch*. Portanto, conforme mais normalizações são realizadas, melhor a aproximação dos seus parâmetros. Os parâmetros da normalização são salvos e mantidos entre as partidas. Em cada *batch*, a normalização só era feita sobre *state* e do *next_state*. Todos os experimentos apresentados neste capítulo utilizam este método de normalização.

### Amostragem da memória

Nestes experimentos, mantivemos amostras aleatórias.

### Recompensas periódicas

Decidiu-se mudar um pouco em relação as recompensas fornecidas ao agente. Fornecendo uma recompensa periódica de valor 1 e uma punição zerada quando o agente perde-se o jogo.

Uma recompensa periódica positiva foi uma maneira de indicar ao agente que quanto mais tempo ele permanece vivo, melhor ele está se saindo. Em relação a punição, o simples fato de não dar recompensa, indica ao agente que o estado atingido ao chegar ao final do jogo é menos desejável.

### Tamanho do lote de exemplos

O número de exemplares retirados da memória para fazer o treino, manteve-se fixado um valor de 256 para todos os agentes desta versão.

#### Atraso na recompensa

Os agentes usados tiveram o atraso na recompensados reduzidos em relação aos últimos atrasos utilizados na versão gráfica. Os resultados da versão gráfica mostraram que acumular recompensa não surtiu efeito positivo para os agentes em termos de estabilidade das pontuações alcançadas. Em todos os agentes DQN da versão simulada, o atraso na recompensa foi mantido em 0.

### 5.1.1 Experimentos DQN

#### Hiper-parâmetros

Nos primeiros experimentos da versão simulada, DQN1 e DQN2, seus hiper-parâmetros foram iguais aos da versão gráfica. Entretanto, os agentes a partir do DQN3 tiveram alterações nos seus hiper-parâmetros. A Tabela 5.1 apresenta os valores valores dos hiper-parâmetros para todos os agentes avaliados.

| $\epsilon$ Decaimento | $\epsilon$ Min. | Taxa Aprendizagem | Agente |
|---|---|---|---|
| 0.995 | 0.01 | 0.001 | DQN1, DQN2 |
| 0.999 | 0.05 | 0.0001 | DQN3, DQN4, DQN5 |

Tabela 5.1: Hiper-parâmetros usados nos experimentos da versão simulada.

Observe na Figura 5.1, que os agentes DQN1 e DQN2 apresentam divergência logo após $\epsilon$ atingir o valor mínimo.

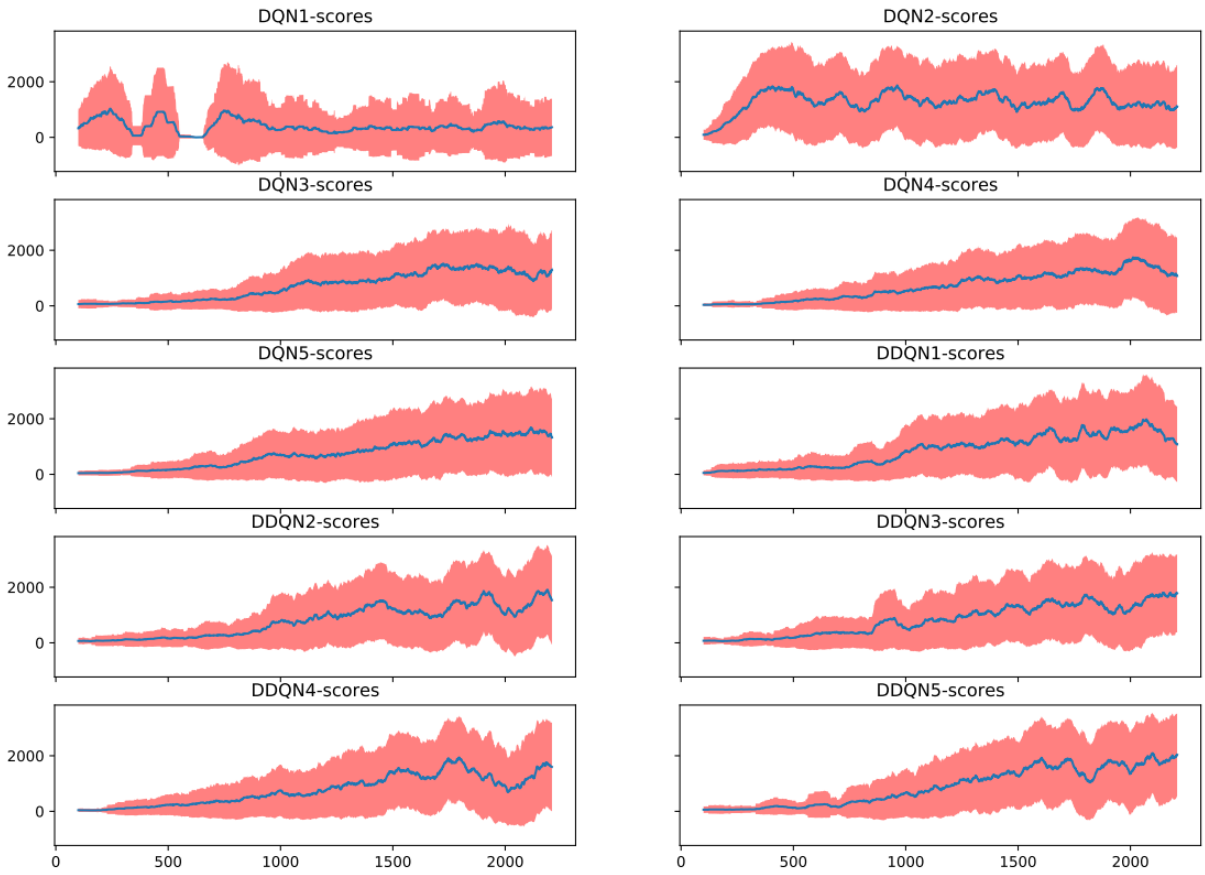


Figura 5.1: Gráfico da pontuação de todos agentes da versão simulada com 2309 partidas. A linha azul representa a média máxima em 100 partidas e a área avermelhada em volta representa o desvio padrão.

A divergência do agente ao atingir $\epsilon$ mínimo sugere que era melhor tomar ações aleatórias do que usar o conhecimento adquirido pelo agente. Isto sugere que a necessidade de diminuir a taxa de aprendizagem com o objetivo de explorar o espaço de parâmetros da DQN de forma mais cuidadosa. A Tabela 5.1 mostra os valores utilizados para os demais agentes investigados neste trabalho.

Os resultados dos agentes DQN3, DQN4 e DQN5 mostraram-se com um comportamento diferente dos agentes DQN1 e DQN2, assim como a Figura 5.1 demonstra. Este novo comportamento indicou o efeito das alterações realizadas nos hiper-parâmetros.

### Atraso do *replay*

Definir um atraso para os agentes realizarem o *replay*, evitando que o agente retome experiências muito antigas da memória e também que ele aprendesse algo obsoleto.

Os agentes DQN1 e DQN2 com atraso para o *replay*, como mostrado na Tabela 5.2.

| Atraso *Replay* | Agente |
|---|---|
| 0 | DQN3 |
| | DQN4 |
| | DQN5 |
| 25 | DQN2 |
| 50 | DQN1 |

Tabela 5.2: Atraso para execução do *replay* dos agentes da versão simulada.

Ao finalizar os testes de DQN1 e DQN2, notou-se que os agentes estavam realizando em média máxima de 6,79 e 23,48 *replays* por jogo, respectivamente. Comparando DQN1 e DQN2, notou-se que DQN2 se saiu bem melhor em seus resultados de pontuação, obtendo uma diferença de 800 pontos a mais da média máxima entre 100 partidas, em relação à DQN1. Indicando que executar mais *replays* faz com que o agente seja eficiente. A Figura 5.2 demosntra todos os resultados de *replays* dos agentes da versão simulada.

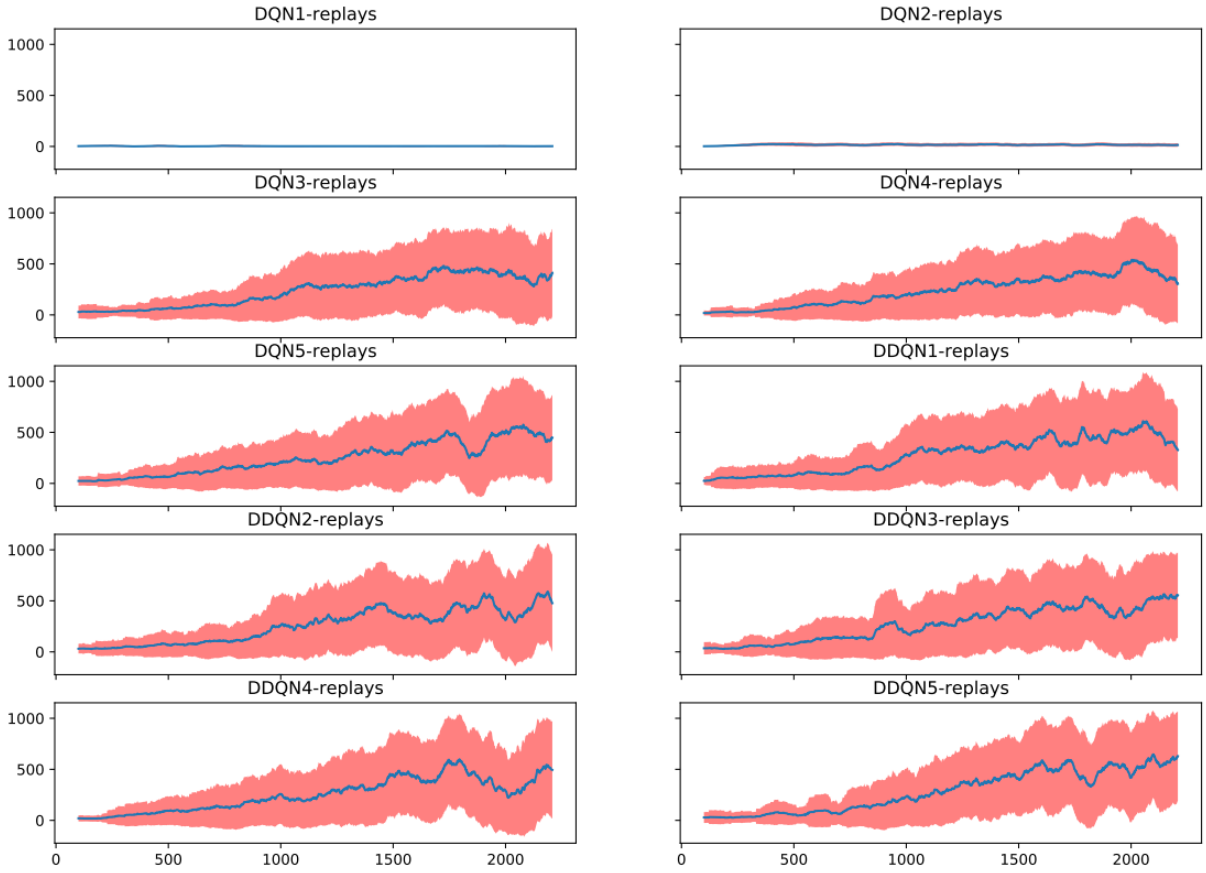


Figura 5.2: Gráfico dos *replays* de todos agentes da versão simulada com 2309 partidas. A linha azul representa a média máxima em 100 partidas e a área avermelhada em volta representa o desvio padrão.

Foi feito também uma avaliação sem o atraso para executar o *replay*. Note a diferença quantitativa de execução de *replays* dentre os agentes DQN1 e DQN2 contra os demais agentes como a Figura 5.2 demonstra.

### Saídas

O agente DQN1 manteve os mesmos elementos de *output* que os agentes da versão gráfica. Testamos essa mesma saída nessa versão simulada do TD para ver como o agente se comportaria. A Tabela 5.3 mostra as duas configurações avaliadas.

| Tamanho da Saída | Dados da Saída | | Agente |
|---|---|---|---|
| 8 | Geral | Selecionar ponto posterior,<br>Selecionar ponto anterior,<br>Não agir | DQN1 |
| | Torre Selecionada | Construir torre básica,<br>Construir torre morteiro,<br>Construir torre repetidora,<br>Melhorar torre,<br>Vender torre | |
| 41 | Geral | Não agir | DQN2 |
| | Por Torre | Construir torre básica,<br>Construir torre morteiro,<br>Construir torre repetidora,<br>Melhorar torre,<br>Vender torre | DQN3<br>DQN4<br>DQN5 |

Tabela 5.3: Saídas dos agentes da versão simulada com método DQN.

A Figura 5.1 mostra que o agente DQN1 não alcançou melhora significativa nas pontuações durante toda sua execução. As saídas correspondentes às ações da DQN1 estavam ligadas diretamente aos controles disponíveis aos jogadores humanos. Neste esquema, primeiro a torre deve ser selecionada. Depois, a ação é escolhida. Portanto, para tomar uma ação efetiva no jogo, são necessárias 2 ações. Pelo fato do agente não ter um mecanismo de múltiplos passos para uma única ação, este modelo parece ser inadequado. A saída de tamanho 8 foi descartada. Para lidar com esse problema, as saídas da rede foram redefinidas para representarem uma ação por torre por saída da rede neural. São 8 torres com 5 ações cada, resultando em 40 saídas. Uma saída extra que corresponde à inação também foi usada, resultando em 41 ações ao todo.

DQN2 apresentou um resultado mais satisfatório em comparação ao DQN1, como a

Figura 5.1 demonstra. Ao observar a Tabela 5.7, nota-se que a diferença da média máxima entre DQN1 e DQN2 é de mais de 800 pontos, indicando que o DQN2 foi eficiente em direcionar as ações nas torres.

Os agentes DQN3, DQN4 e DQN5 mantiveram esse novo formato de saída, conforme a Tabela 5.3 mostra.

**Entradas**

Os agentes VG3 e VG4 possuíam uma entrada na qual era fornecido informações sobre o desempenho das torres. Entretanto foi decidido expandir essa entrada para fornecer informações extras para os agentes. Expandimos então a descrição do ambiente para o agente, como demonstra a Tabela 5.4.

| Dados de Entrada | | Tamanho da Entrada |
|---|---|---|
| Geral | Dinheiro,<br>Inimigos por tipo a cada trecho do caminho (18) | 83 |
| Por Torre | Inimigos abatidos por tipo (6), Tipo de torre,<br>Nível de torre | |

Tabela 5.4: Entrada utilizada na versão simulada.

Os agentes desta versão possuem uma nova entrada, onde foi fornecida uma informação mais detalhada. Dividiu-se o caminho em três partes e para cada uma delas, uma contagem dos inimigos por tipo foi feita. Estas novas informações foram adicionadas na entrada do agente.

DQN1 em relação a VG4, teve um desempenho em pontuação inferior. Entretanto, o desvio padrão caiu de 3508,18 em VG4 para 2070,39 no DQN1. Isto indica que a pontuação média se tornou mais consistente ao longo do tempo, como mostrado na parte sombreada na Figura 5.1. O agente possuir o conhecimento da concentração dos inimigos ao longo do caminho, influenciou positivamente nas tomadas de decisão do agente.

### Modelos da rede neural

Assim como na versão gráfica, nesta versão também não seguimos nenhum exemplar da literatura para construção dos modelos. Não foi encontrado na literatura nenhum agente com RL para jogar um TD, por isso a definição do modelo não teve nenhum exemplo para comparação. A Tabela 5.5 demonstra a dimensionalidade das camadas ocultas dos modelos.

| Neurônios nas Camadas Ocultas | Agente |
|---|---|
| [100\|100] | DQN1 |
| [200\|200] | DQN2, DQN3 |
| [200\|200\|200] | DQN5 |
| [200\|150\|150\|200\|150\|150\|200] | DQN4 |

Tabela 5.5: Camadas e número de neurônios nas camadas ocultas dos agentes da versão simulada.

Os resultados de DQN1 e DQN2 se mostraram insatisfatórios, já que ambos apresentavam instabilidade como a Figura 5.1 demonstra. DQN2 e DQN3 tinham o mesmo modelo porém apresentaram, em seus resultados de pontuação, uma diferença significativa no desvio padrão com 2769,99 para DQN2 e 1885,61 para DQN3. Esta diferença indicou que os demais parâmetros, como atraso de *replay*, estavam realmente surtindo efeito nas tomadas de decisões do agente.

DQN4 e DQN5 são idênticos ao DQN3, exceto pela arquitetura da rede neural. Realizamos 2309 jogos para cada um destes três agentes, como demonstrado pelas Tabelas 5.7 e 5.9 as médias máximas analisadas de cada agente.

Os resultados mostram que DQN4 se mostrou eficaz em relação aos ciclos, além de apresentar menor desvio padrão nos *replays* e pontuação. Estas observações foram consideras no momento de decidir a construção dos agentes com DDQN, sendo um indicativo que as redes mais profundas mostraram-se mais estáveis.

### 5.1.2 Experimentos DDQN

Com os resultados apresentados pelos últimos experimentos da versão simulada, buscamos na literatura, [7], informações que discutiam sobre agentes que usam DQN. Uma proposta encontrada foi a implementação de um agente com Double Deep Q Network (DDQN). Foi então que construímos outros agentes para realizar mais testes, buscando observar melhorias no desempenho dos agentes com essa nova implementação. A DDQN pode suavizar o processo de otimização do agente, para isso mantêm-se o *target* do processo de otimização estacionário por um tempo. Alguns algoritmos de otimização assumem a estacionariedade para obterem um bom funcionamento, o que não se aplica à DQN. A rede *target* tem seus pesos copiados da rede *on-line* em intervalos regulares de tempo. O objetivo da abordagem DDQN são resultados mais consistentes, além de aumentar a performance do agente.

A implementação de DDQN para os novos agentes implicou o uso de um novo parâmetro. Este parâmetro baseia-se em quantos *replays* o agente deve esperar até realizar uma cópia dos pesos da rede *on-line* para a rede *target*. Aplicou-se esta nova abordagem para todos os agentes DDQN.

#### Configuração dos agentes DDQN

Todos os agentes DDQN preservaram diversos parâmetros dos agentes DQN sendo eles:

- **Hiper-parâmetros:** Para observar melhor as diferenças no desempenho dos agentes DDQN, manteve-se os valores de hiper-parâmetros idênticos aos dos agentes DQN. Os valores mantidos são: 0,999 para decaimento do $\epsilon$, 0,05 para o valor mínimo do $\epsilon$ e a taxa de aprendizagem de 0,0001.

- **Normalização:** A normalização foi feita como nos agentes DQN.

- **Amostragem da memória:** A amostragem da memória foi feita como nos agentes

DQN.

- **Atraso do *replay*:** O atraso para execução do *replay* permaneceu zerado, assim como os agentes DQN3, DQN4 e DQN5.

- **Saídas:** Todos os agentes DDQN mantiveram a mesma saída que os agentes DQN.

- **Entradas:** Todos os agentes DDQN mantiveram a mesma entrada que os agentes DQN.

- **Recompensas periódicas:** As recompensas periódicas usadas nos agentes DDQN, preservaram os mesmo valores dos agentes DQN. Os valores mantidos são: 1 para recompensa periódica e 0 para punição com a perca do jogo.

- **Tamanho do lote de exemplos:** Os agentes com DDQN tiveram o tamanho do lote de exemplos mantido de acordo com os agentes DQN, sendo o tamanho de 256.

- **Atraso na recompensa:** Definimos que o atraso da recompensa utilizado pelos agentes DDQN seria o mesmo do que a versão DQN.

Demais parâmetros utilizados, modelos da rede neural e cópias de pesos, para os agentes DDQN possuem diferenças dentre os agentes.

### Modelos da rede neural

Para avaliar o impacto do método DDQN, foram avaliadas duas arquiteturas idênticas as utilizadas nos agentes DQN. As arquiteturas selecionadas são mostradas na Tabela 5.6.

A ideia foi avaliar a DDQN com uma arquitetura rasa e outra mais profunda.

| Arquitetura Rede | Agente |
|---|---|
| [200\|200\|200] | DDQN2 |
| [200\|150\|150\|200\|150\|150\|200] | DDQN1, DDQN3, DDQN4, DDQN5 |

Tabela 5.6: Arquiteturas utilizadas dos agentes DDQN.

Todos agentes DDQN utilizaram mesma arquitetura que o agente DQN4, exceto o agente DDQN2 que seguiu o modelo do agente DQN5. DDQN2 foi criado para ser um experimento com uma arquitetura mais rasa. A Tabela 5.7 mostra que o agente DDQN5 obteve a maior média dentre os agentes DDQN. Entretanto a maior pontuação e o maior desvio padrão feitos, foram pelo agente DDQN2. Isto indica que a rede mais rasa se tornou mais instável diante dos agente DDQN.

| Média Máxima | Pontuação Máxima | Desvio Padrão RMS | Agente |
|---|---|---|---|
| 1032,65 | 8815 | 2068,19 | DQN1 |
| 1871,25 | 5505 | 2769,99 | DQN2 |
| 1518,70 | 5130 | 1885,61 | DQN3 |
| 1746,25 | 5515 | 1855,50 | DQN4 |
| 1879,65 | 5305 | 1970,43 | DQN5 |
| 1981,70 | 5460 | 2092,60 | DDQN1 |
| 1915,00 | 6110 | 2106,12 | DDQN2 |
| 1814,50 | 4935 | 2074,93 | DDQN3 |
| 1940,45 | 5610 | 2090,43 | DDQN4 |
| 2180,65 | 5695 | 2095,42 | DDQN5 |

Tabela 5.7: Visão geral sobre as pontuações obtidas pelos agentes DQN e DDQN

### Cópia de pesos

A estratégia da DDQN é manter os alvos da otimização estacionários por um determinado tempo. Isto satisfaz o requisito da estacionariedade dos algoritmos de otimização por descida do gradiente. Este tempo entre atualizações dos alvos é um parâmetro que deve ser otimizado. A Tabela 5.8 apresenta o intervalo entre as atualizações utilizados em cada agente.

| Contagem Satisfeita | Replays para Cópia | Agente |
|---|---|---|
| Mesma Partida | 10 | DDQN1 |
| Mesma Partida | 30 | DDQN2 |
| Mesma Partida | 100 | DDQN3 |
| Entre Partidas | | DDQN4 |
| Entre Partidas | 1000 | DDQN5 |

Tabela 5.8: Quantidade de *replays* para o agente fazer a cópia dos pesos entre as redes *on-line* e *target*.

Os agentes DDQN1, DDQN2 e DDQN3 só atualizam a rede *target* se a quantidade de *replays* necessária acontecer em uma mesma partida. Para os agentes DDQN4 e DDQN5, a contagem dos *replays* é mantida entre as partidas, possibilitando a uma frequência maior de atualização dos pesos da rede *target*.

No quesito pontuação, DDQN3 obteve uma média máxima 1814,5 e uma pontuação máxima de 4935, enquanto o agente DDQN4 teve uma média máxima de 1940,45 e uma pontuação máxima de 5610. Indicando que as cópias extras realizadas pelo agente DDQN4 influenciam positivamente no desempenho do agente. DDQN5 teve o maior intervalo de *replays* para fazer uma cópia. Percebe-se na Tabela 5.7, o único agente que obteve uma média máxima (2180,65) maior do que o desvio padrão (2095,42) foi o agente DDQN5. Tornando-o agente mais eficiente construído neste trabalho.

## 5.2 Discussão dos resultados

A Figura5.1 mostra que o comportamento da curva das pontuações dos agentes DQN1 e DQN2 não mostram uma tendência de melhoria com o tempo. Embora a Tabela 5.7 mostre que DQN1 obteve um a maior pontuação máxima entre todos os agentes, a média máxima entre 100 partidas consecutivas é de apenas 1032,65 pontos. Isto indica que a pontuação máxima não é resultado do treinamento, mas um resultado espúrio. A melhora da média máxima do de DQN2, para 1871,25 pontos em relação a DQN1 se deu pela alteração das entradas do agente. Estas novas entradas agora permitem que o agente possa perceber melhor o estado atual do jogo, usando estatísticas de eficiência das torres para escolher a a próxima ação a ser tomada. Entretanto o desvio padrão foi maior no agente DQN2 em relação ao agente DQN1, indicando que DQN2 foi menos estável que DQN1. Uma das razões para isso é provavelmente a dimensionalidade maior da camada de entrada, que requer mais *replays* para um treinamento adequado.

Ambos DQN1 e DQN2 usam um atraso entre e a ação e a execução do *replay* correspondente, conforme mostrado na Tabela 5.2. Os demais agentes DQN e DDQN não introduzem esse atraso. Como consequência, a quantidade de *replays* executados pelos agentes DQN1 e DQN2 é muito menor do que dos demais agentes DQN e DDQN, conforme mostrado na Tabela 5.9. A otimização do agente é realizada apenas quando os *replays* são executados. Portanto, com relativamente poucos *replays*, DQN1 e DQN2 não apresentam a tendência de melhoria no intervalo das 2309 partidas analisadas.

| Média Máxima | Replays Máximo | Desvio Padrão | Agente |
|---|---|---|---|
| 6,79 | 51 | 12,12 | DQN1 |
| 23,48 | 67 | 33,06 | DQN2 |
| 480,98 | 1592 | 580,05 | DQN3 |
| 539,75 | 1717 | 575,03 | DQN4 |
| 575,62 | 1622 | 605,42 | DQN5 |
| 609,78 | 1663 | 638,69 | DDQN1 |
| 591,89 | 1922 | 639,74 | DDQN2 |
| 564,37 | 1502 | 632,13 | DDQN3 |
| 598,73 | 1748 | 637,71 | DDQN4 |
| 672,85 | 1706 | 633,23 | DDQN5 |

Tabela 5.9: Resultados de todos agentes da versão simulada com 2309 partidas.

Novos valores para os hiper-parâmetros, $\epsilon$ e taxa de aprendizagem foram utilizados a partir do agente DQN3. A diminuição da taxa de decaimento de $\epsilon$ e da taxa de aprendizagem tornaram o processo de otimização mais robusto. A Figura 5.1 mostra que a partir de DQN3 a pontuação passa a seguir uma tendência de melhoria. Esta melhoria também está associada ao fato que a partir da DQN3, optamos por não colocar atrasos nos *replays*, conforme descrito acima.

Os agentes DQN3, DQN4 e DQN5 se diferenciam principalmente em relação as suas arquiteturas. DQN3 possui 2 camadas ocultas, enquanto DQN4 e DQN5 possuem 7 e 3 camadas ocultas, respectivamente. Nota-se na Figura 5.1 que DQN3 apresenta estagnação ao atingir em torno de 1500 partidas. Por outro lado, os agentes DQN4 e DQN5 mantêm o a tendência ascendente. Isto indica que as arquiteturas mais profundas nas redes DQN4 e DQN5 foram benéficas para o desempenho dos agentes. A Tabela 5.7 mostra que, no geral, DQN3, DQN4 e DQN5 apresentaram desvio padrão RMS inferior aos atingidos pelas redes DQN1 e DQN2. Isto indica melhor estabilidade ao longo do tempo nas redes com os hiper-parâmetros ajustados.

A estratégia DDQN foi avaliada para determinar se era possível aprimorar os resultados

obtidos pelos agentes DQN. A Tabela 5.7 mostra uma visão geral dos resultados, tanto dos agentes baseados em DQN quanto os agentes baseados em DDQN. Os agentes DQN5 e DDQN2 possuem exatamente a mesma configuração, incluindo a arquitetura da rede neural, exceto por usarem DQN e DDQN, respectivamente. O agente DQN5 obteve média máxima entre 100 partidas de 1879,65 pontos e uma pontuação máxima de 5305 pontos. Por outro lado, o agente DDQN2 obteve a média máxima entre 100 partidas consecutivas de 1915 pontos e média máxima de 6110.

Os agentes DQN4, DDQN1, DDQN3, DDQN4 e DDQN5 possuem exatamente a mesma configuração, incluindo a arquitetura da rede neural, exceto por usarem DQN e DDQN. A Tabela 5.7 mostra que o agente DQN4 teve a menor média máxima em relação aos agentes DDQN equivalentes. A Figura 5.1 mostra que o comportamento das curvas de pontuação dos agentes DQN3, DQN3 e DQN5 é semelhante aos agentes DDQN. Todas as curvas mostram uma tendência ascendente na pontuação ao longo do tempo. Entretanto, conforme indicado acima, os agentes DDQN levaram a resultados ligeiramente superiores. Isto sugere que a estacionariedade momentânea dos alvos durante a otimização ajuda obter melhores resultados ao longo do tempo.

A coluna “Desvio Padrão RMS” na Tabela 5.7 mostra que o desvio padrão RMS na pontuação dos agentes DQN3, DQN4 e DQN5 foi menor do que os resultados obtidos com DDQN. Isto mostra que, embora os agentes DDQN tenham obtido, na média, resultados superiores aos agentes DQN, eles foram ligeiramente menos estáveis.

A pontuação máxima obtida por um jogador humano usando a versão gráfica do jogo foi de 18680 pontos. Comparando com o melhor resultado obtido entre os agentes DQN3-DDQN5 mostrados na Tabela 5.7, nota-se que o agente não obteve o mesmo desempenho de um jogador humano. Outros trabalhos mostram que normalmente os agentes necessitam de uma grande quantidade de partidas para superarem jogadores humanos [3], [8]. Nos experimentos apresentados, os agentes jogaram pouco mais de 2300 partidas. Portanto é possível que os agentes sejam capazes de atingir pontuações mais competitivas se executados por mais tempo. As tendências ascendentes mostradas na Figura 5.1 indicam esta possibilidade.

## 5.3 Sumário

Neste capítulo foram apresentados diversos agentes com combinações de parâmetros diferentes para fazer a avaliação e discussão dos resultados. Ainda há a implementação de agentes com DDQN buscando expandir os experimentos. No próximo capítulo são apresentados as conclusões deste trabalhos realizado e os pontos para os trabalhos futuros.

# Capítulo 6

# Conclusões

Neste trabalhos foram desenvolvidos vários agentes utilizando técnicas de Aprendizagem por Reforço baseadas em *Deep Q-Learning* para jogar *Tower Defense*. Em outros trabalhos que usam *Deep Q-Learning*, os agentes usualmente recebem os quadros de vídeo do jogo para como entrada. Neste trabalho investigamos a possibilidade do agente receber metadados da partida como entrada. Como não encontramos na literatura outra implementação de agentes para jogar *Tower Defense*, a construção do agente foi realizada de forma experimental e incremental.

Inicialmente o jogo foi concebido com o objetivo de incentivar o jogador a criar estratégias para vencer as hordas cada vez mais desafiadoras de inimigos. Primeiramente o jogo foi implementado com uma interface gráfica, permitindo que humanos pudessem jogar. Utilizando essa versão, o jogo foi calibrado para oferecer um equilíbrio entre desafio e diversão. Vários parâmetros do agente foram inicialmente ajustados nesta versão. Usando a versão gráfica do jogo, 3 configurações de entrada foram propostas.

O agente VG1 teve a entrada do jogo focada em refletir o estado do jogo como um todo. Para o agente VG2, decidiu-se refletir o ambiente do jogo apenas usando informações ligadas as torres. Os resultados apresentados foram insatisfatórios, levando ao desenvolvimento de uma terceira configuração. Nesta configuração, utilizada em ambos VG3 e VG4, a entrada descreve o desempenho das torres em relação aos abates por tipo de inimigo. Esta nova configuração da entrada fez com que o desempenho dos agentes

melhorasse significativamente. Isto sugere que o agente passou a ter informações suficientes para interpretar melhor o ambiente na qual estava foi inserido, levando a decisões mais vantajosas ao agente ao longo do tempo.

Como a versão gráfica foi projetada para humanos jogarem, os tempos dos eventos do jogo estavam dependentes do relógio do mundo real. Isto fazia com que as partidas demorassem muito tempo para executar. Isto impediria a realização de experimentos de longa duração, que é importante para a avaliação da tendência do progresso do agente. A solução encontrada foi reimplementar o jogo em uma versão que não depende do relógio do mundo real e que não utilize uma interface gráfica. Essa implementação foi batizada de versão simulada. Todos os demais agentes foram desenvolvidos nesta nova versão.

Dentre outras modificações no comportamento do agente, uma nova configuração de entrada também foi avaliada. Nesta configuração foram adicionadas informações acerca da quantidade de inimigos em cada terço do caminho. Esta configuração da entrada deixou os agentes mais estáveis ao longo do tempo, colaborando com uma tendência ascendente na pontuação média entre as partidas.

Várias arquiteturas da rede neural foram avaliadas. Os resultados mostram que as redes mais profundas resultaram em agentes mais eficientes. Além disto, também foram avaliados parâmetros do otimizador. A diminuição da taxa de aprendizagem foi um dos fatores que beneficiaram os resultados.

Embora os agentes DQN tenham obtido bons resultados, a taxa de melhoria dos resultados é baixa. Muitos jogos são necessários para que as pontuações melhorem significativamente. Para tentar aprimorar essa taxa, agentes baseados em DDQN foram implementados. A configuração dos agentes DDQN foi a mesma dos agentes DQN, diferindo apenas no fato de usarem DDQN ao invés de DQN. Os resultados mostraram que os agentes DDQN levaram a pontuações mais altas que os agentes DQN correspondentes.

Este trabalho mostrou que é possível construir um agente baseado em Aprendizagem por Reforço capaz de jogar Tower Defense. Nenhum outro trabalho na literatura relata o desenvolvimento de agentes autônomos para aprender como jogar este tipo de jogo. Além disto, projeto das entradas baseadas em metadados permitiu utilizar dados de entrada

mais simples do que as abordagens baseadas em quadros de vídeo. Por esta razão, esta abordagem mostra-se promissora para embutir agentes baseados em aprendizagem por reforço em dispositivos com menor poder computacional.

Os resultados relatados neste trabalho sugerem novas direções para trabalhos futuros. Os agentes usados neste trabalho não modelam diretamente a relação temporal entre as sequências de ações e suas respectivas consequências. Um possível trabalho seria avaliar como que o uso de técnicas baseadas em padrões temporais, como LSTMs e cadeias ocultas de Markov podem ser usadas para considerar o comportamento temporal. Outro possível trabalho seria avaliar as estratégias criadas pelo agente. A avaliação de estratégias é uma técnica que pode ajudar no aperfeiçoamento da calibração de parâmetros de outros jogos. Explorar mais variações de entrada para os agentes também é um possível trabalho futuro, buscando fornecer informações mais relevantes para as tomadas de decisão. Por fim, uma comparação com um agente baseado em quadros de vídeo também seria interessante para avaliar a diferença de desempenho entre esta abordagem e a abordagem apresentada neste trabalho.

# Bibliografia


[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage e A. A. Bharath, "A Brief Survey of Deep Reinforcement Learning", *arxiv*, set. de 2017. URL: https://arxiv.org/pdf/1708.05866.pdf.

[2] Y. Li, "DEEP REINFORCEMENT LEARNING: AN OVERVIEW", *arxiv*, set. de 2017. URL: https://arxiv.org/pdf/1701.07274.pdf.

[3] J. Grigsby. (2018). Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning, URL: https://towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814 (acedido em 22/11/2018).

[4] D. M. Sefair. (2017). My first experience with deep reinforcement learning, URL: https://medium.com/ai-society/my-first-experience-with-deep-reinforcement-learning-1743594f0361 (acedido em 23/11/2018).

[5] F. M. Graetz. (2018). How to match DeepMind's Deep Q-Learning score in Breakout, URL: https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756 (acedido em 22/11/2018).

[6] N. Justesen e S. Risi, "Learning Macromanagement in StarCraft from Replays using Deep Learning", *CoRR*, vol. abs/1707.03743, 2017. arXiv: 1707.03743. URL: http://arxiv.org/abs/1707.03743.

[7] H. Van Hasselt, A. Guez e D. Silver, "Deep reinforcement learning with double q-learning", em *Thirtieth AAAI conference on artificial intelligence*, 2016.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg e D. Hassabis, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, 529 EP -, fev. de 2015. URL: https://doi.org/10.1038/nature14236.

[9] M. COPELAND. (2016). What's the Difference Between Artificial Intelligence, Machine Learning, and Deep Learning?, URL: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/ (acedido em 25/11/2018).

[10] N. Justesen, P. Bontrager, J. Togelius e S. Risi, "Deep Learning for Video Game Playing", *arxiv*, out. de 2017. URL: https://arxiv.org/pdf/1708.07902.pdf.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra e M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", *arxiv*, dez. de 2013. URL: https://arxiv.org/pdf/1312.5602v1.pdf.

[12] S. K. Kim, E. A. Kirchner, A. Stefes e F. Kirchner, "Intrinsic interactive reinforcement learning - Using error-related potentials for real world human-robot interaction", *Scientific Reports*, vol. 7, n.º 1, p. 17562, 2017, ISSN: 2045-2322. DOI: 10.1038/s41598-017-17682-7. URL: https://doi.org/10.1038/s41598-017-17682-7.

[13] C. Liang, J. Berant, Q. Le, K. D. Forbus e N. Lao, "Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision", *CoRR*, vol. abs/1611.00020, 2016. arXiv: 1611.00020. URL: http://arxiv.org/abs/1611.00020.

[14] V. Mnih, N. Heess, A. Graves e K. Kavukcuoglu, "Recurrent models of visual attention.", *NIPS*, 2014. URL: https://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.pdf.

[15] F. S. Melo, "Convergence of Q-learning: a simple proof", 2001. URL: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

[16] A. M. Metelli, M. Mutti e M. Restelli, "Configurable Markov Decision Processes", *arxiv*, jul. de 2018. URL: https://arxiv.org/pdf/1806.05415.pdf.

[17] G. Bittencourt, *Inteligência Artificial Ferramentas e Teorias*. Editora da UFSC, 2006.

[18] S. Haykin, "Redes neurais: princípios e prática", em, Porto Alegre: Bookman, 2001, pp. 27–137, ISBN: 978-85-7307-718-6.

[19] I. Goodfellow, Y. Bengio e A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[20] A. Krizhevsky, I. Sutskever e G. E. Hinton, "Imagenet classification with deep convolutional neural networks.", 2012.

[21] I. Sutskever, O. Vinyals e Q. V. Le, "Sequence to sequence learning with neural networks.", 2014.

[22] D. H. Ballard, G. E. Hinton e T. J. Sejnowski, "Parallel visual computation", *Nature*, vol. 306, 21 EP -, nov. de 1983. URL: https://doi.org/10.1038/306021a0.

[23] G. Rossum, *Python Tutorial: Release 3. 6. 6rc1*. CreateSpace Independent Publishing Platform, 2018, ISBN: 9781721242160. URL: https://books.google.pt/books?id=vkWltgEACAAJ.

[24] (Mai. de 2020). About Keras, URL: https://keras.io/about/.

[25] (Mai. de 2020). Why choose Keras?, URL: https://keras.io/why_keras/.

[26] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon

Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu e Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. URL: https://www.tensorflow.org/.

[27] (Mai. de 2020). About — wiki, URL: https://www.pygame.org/wiki/about.

[28] (Mai. de 2020). About matplotlib, URL: https://matplotlib.org/faq/usage_faq.html#coding-styles.

[29] (Mai. de 2020). About pyplot, URL: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot e E. Duchesnay, "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[31] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell e S. Legg, "Noisy Networks for Exploration", *CoRR*, vol. abs/1706.10295, 2017. arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

[32] P. Peng, Q. Yuan, Y. Wen, Y. Yang, Z. Tang, H. Long e J. Wang, "Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games", *CoRR*, vol. abs/1703.10069, 2017. arXiv: 1703.10069. URL: http://arxiv.org/abs/1703.10069.

[33] (Mai. de 2020). Gym library, URL: https://gym.openai.com/.

[34] Keon. (mai. de 2020). Deep Q-Learning with Keras and Gym, URL: https://keon.github.io/deep-q-learning/.

[35] M. Comi. (mai. de 2020). How to teach AI to play Games: Deep Reinforcement Learning, URL: https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a.

[36] N. Usunier, G. Synnaeve, Z. Lin e S. Chintala, "Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks", *CoRR*, vol. abs/1609.02993, 2016. arXiv: 1609.02993. URL: http://arxiv.org/abs/1609.02993.

# Apêndice A

# Configuração dos Agentes

| Agente | $\epsilon$ Decaimento | $\epsilon$ Mín. | Taxa Aprendizagem | Tamanho *batch* | Neurônios nas Camadas Ocultas | Entrada | Saída |
|---|---|---|---|---|---|---|---|
| VG1 | 0,995 | 0,01 | 0,001 | 32 | [40\|20] | E-1 | S-1 |
| VG2 | 0,995 | 0,01 | 0,001 | 32 | [48\|16] | E-2 | S-1 |
| VG3 | 0,995 | 0,01 | 0,001 | 256 | [320\|320\|200] | E-3 | S-1 |
| VG4 | 0,995 | 0,01 | 0,001 | 64 | [320\|320\|200] | E-3 | S-1 |
| DQN1 | 0,995 | 0,01 | 0,001 | 256 | [100\|100] | E-4 | S-1 |
| DQN2 | 0,995 | 0,01 | 0,001 | 256 | [200\|200] | E-4 | S-2 |
| DQN3 | 0,999 | 0,05 | 0,0001 | 256 | [200\|200] | E-4 | S-2 |
| DQN4 | 0,999 | 0,05 | 0,0001 | 256 | [200\|150\|150\|200\|150\|150\|200] | E-4 | S-2 |
| DQN5 | 0,999 | 0,05 | 0,0001 | 256 | [200\|200\|200] | E-4 | S-2 |
| DDQN1 | 0,999 | 0,05 | 0,0001 | 256 | [200\|150\|150\|200\|150\|150\|200] | E-4 | S-2 |
| DDQN2 | 0,999 | 0,05 | 0,0001 | 256 | [200\|200\|200] | E-4 | S-2 |
| DDQN3 | 0,999 | 0,05 | 0,0001 | 256 | [200\|150\|150\|200\|150\|150\|200] | E-4 | S-2 |
| DDQN4 | 0,999 | 0,05 | 0,0001 | 256 | [200\|150\|150\|200\|150\|150\|200] | E-4 | S-2 |
| DDQN5 | 0,999 | 0,05 | 0,0001 | 256 | [200\|150\|150\|200\|150\|150\|200] | E-4 | S-2 |

Tabela A.1: Dados de todos agentes construídos neste trabalho. As informações das colunas de Entrada e Saída se encontram nas Tabelas A.3 e A.4.

| Agente | Normalização | Recompensa Periódica | Punição Perda | Atraso Recompensa | Atraso Replay | Replays para Cópia | Contagem Satisfeita |
|---|---|---|---|---|---|---|---|
| VG1 | Por batch | -1 | -50 | 0 | 0 | - | - |
| VG2 | Por batch | -1 | -50 | 0 | 0 | - | - |
| VG3 | Por batch | 0 | 0 | 20 | 0 | - | - |
| VG4 | Por batch | 0 | -50 | 20 | 0 | - | - |
| DQN1 | On-line | 1 | 0 | 0 | 50 | - | - |
| DQN2 | On-line | 1 | 0 | 0 | 25 | - | - |
| DQN3 | On-line | 1 | 0 | 0 | 0 | - | - |
| DQN4 | On-line | 1 | 0 | 0 | 0 | - | - |
| DQN5 | On-line | 1 | 0 | 0 | 0 | - | - |
| DDQN1 | On-line | 1 | 0 | 0 | 0 | 10 | Mesma Partida |
| DDQN2 | On-line | 1 | 0 | 0 | 0 | 30 | Mesma Partida |
| DDQN3 | On-line | 1 | 0 | 0 | 0 | 100 | Mesma Partida |
| DDQN4 | On-line | 1 | 0 | 0 | 0 | 100 | Entre Partidas |
| DDQN5 | On-line | 1 | 0 | 0 | 0 | 1000 | Entre Partidas |

Tabela A.2: Dados usados na configuração de cada agente deste trabalho.

| Configuração | Dados de Entrada | | Tamanho da Entrada |
|---|---|---|---|
| E-1 | Geral | Vida, Dinheiro, Pontuação, Nível dificuldade, Torre selecionada (8), Quantidade de inimigos | 20 |
| | Torre Selecionada | Nível torre (4), Tipo torre, Custo melhoria, Retorno venda | |
| E-2 | Por Torre | Tipo da torre, Nível da torre | 16 |
| E-3 | Geral | Dinheiro | 65 |
| | Por Torre | Inimigos abatidos por tipo (6), Tipo da torre, Nível da torre | |
| E-4 | Geral | Dinheiro, Inimigos por tipo a cada trecho do caminho (18) | 83 |
| | Por Torre | Inimigos abatidos por tipo (6), Tipo da torre, Nível da torre | |

Tabela A.3: Configurações das entradas avaliadas

| Configuração | Dados de Saída | | Tamanho da Saída |
|---|---|---|---|
| S-1 | Geral | Selecionar ponto posterior, Selecionar ponto anterior, Não agir | 8 |
| | Torre Selecionada | Construir torre básica, Construir torre morteiro, Construir torre repetidora, Melhorar torre, Vender torre | |
| S-2 | Geral | Não agir | 41 |
| | Por Torre | Construir torre básica, Construir torre morteiro, Construir torre repetidora, Melhorar torre, Vender torre | |

Tabela A.4: Configurações das saídas avaliadas