Centro Federal de Educação Tecnológica de Minas Gerais
Campus Divinópolis
Graduação em Engenharia Mecatrônica

Leonardo Viana Teixeira

Desenvolvimento de um agente inteligente para exploração autônoma de ambientes 3D via Visual Reinforcement Learning

Divinópolis
2018

Leonardo Viana Teixeira

Desenvolvimento de um agente inteligente para exploração autônoma de ambientes 3D via Visual Reinforcement Learning

Monografia de Trabalho de Conclusão de Curso apresentada ao Colegiado de Graduação em Engenharia Mecatrônica como parte dos requisitos exigidos para a obtenção do título de Engenheiro Mecatrônico.
Áreas de Integração: Controle e Computação.

Orientador: Prof. Dr. Thiago Magela Rodrigues Dias

Divinópolis
2018

(Catalogação - Biblioteca Universitária – Campus Divinópolis – CEFET-MG)

T266d Teixeira, Leonardo Viana.

Desenvolvimento de um agente inteligente para exploração autônoma de ambientes 3D via *visual reinforcement learning*. / Leonardo Viana Teixeira. – Divinópolis, 2018.

121f. ; il.

Orientador: Prof. Dr. Thiago Magela Rodrigues Dias.

Trabalho de Conclusão de Curso (graduação) – Colegiado de Graduação em Engenharia Mecatrônica do Centro Federal de Educação Tecnológica de Minas Gerais.

1. Computação. 2. Controle. 3. Inteligência Computacional. 4. *Reinforcement Learning*. 5. *Deep Learning*. 6. Ambiente 3D. I. Dias, Thiago Magela Rodrigues. II. Centro Federal de Educação Tecnológica de Minas Gerais. III. Título.

CDU: 62(043)

Leonardo Viana Teixeira

Desenvolvimento de um agente inteligente para exploração autônoma de ambientes 3D via Visual Reinforcement Learning

Monografia de Trabalho de Conclusão de Curso apresentada ao Colegiado de Graduação em Engenharia Mecatrônica como parte dos requisitos exigidos para a obtenção do título de Engenheiro Mecatrônico.
Áreas de Integração: Controle e Computação.

Comissão Avaliadora:

Prof. Dr. Thiago Magela Rodrigues Dias
CEFET-MG *Campus* Divinópolis

Prof. Dr. Luís Filipe Pereira Silva
CEFET-MG *Campus* Divinópolis

Prof. Dr. Alisson Marques da Silva
CEFET-MG *Campus* Divinópolis

Divinópolis
2018

Centro Federal de Educação Tecnológica de Minas Gerais
CEFET-MG / Campus Divinópolis
Curso de Engenharia Mecatrônica

Monografia intitulada “DESENVOLVIMENTO DE UM AGENTE INTELIGENTE PARA EXPLORAÇÃO AUTÔNOMA DE AMBIENTES 3D VIA VISUAL REINFORCEMENT LEARNING“, de autoria do graduando Leonardo Viana Teixeira, aprovada pela banca examinadora constituída pelos seguintes professores:

_____

Prof. Dr. Thiago Magela Rodrigues Dias - CEFET-MG / Campus Divinópolis - Orientador

_____

Prof. Dr. Luís Filipe Pereira Silva - CEFET-MG / Campus Divinópolis

_____

Prof. Dr. Alisson Marques da Silva - CEFET-MG / Campus Divinópolis

_____

Prof. Dr. Lúcio Flávio Santos Patrício
Coordenador do Curso de Engenharia Mecatrônica
CEFET-MG / Campus Divinópolis

Divinópolis
2018

Dedico este trabalho aos meus pais Enes e Odete pela confiança e incentivo, aos meus irmãos Enes e Talessa pela amizade e fraternidade e à Taise por sua cumplicidade e companheirismo.

Agradecimentos

Agradeço,

aos meus pais, Enes e Odete, aos meus irmãos, Enes e Talessa, e à minha companheira Taise por todo seu amor, confiança, carinho e apoio incondicional durante todas as etapas da minha vida.

A todos os meus amigos pelo companheirismo.

A todos os meus professores que contribuíram para minha formação profissional, em especial ao professor Thiago, por sua orientação e amizade.

Aos membros da banca examinadora pelos comentários, sugestões e contribuições, que ajudaram a melhorar a qualidade e a redação final do manuscrito.

A todos que de alguma forma contribuíram com o meu progresso como aluno e como ser.

A imaginação é mais importante que a ciência, porque a ciência é limitada, ao passo que a imaginação abrange o mundo inteiro.

Albert Einstein

Resumo

O seguinte trabalho consistiu no desenvolvimento de um agente inteligente para a exploração autônoma de ambientes 3D via *visual reinforcement learning*, englobando duas das grandes áreas da engenharia mecatrônica: controle e computação. Ensinar uma inteligência computacional a navegar de forma eficiente em ambientes complexos nos trará a um passo mais próximo do uso de robôs inteligentes em nosso dia a dia, como por exemplo carros autônomos. Portanto, para fins de avaliar algumas das tecnologias atuais que buscam esse objetivo, foi realizado um estudo, desenvolvimento e treinamento de uma inteligência computacional via *visual reinforcement learning* para a navegação ambientes virtuais tridimensionais, possuindo como entrada os pixels da tela destes ambientes. A técnica de *visual reinforcement learning*, também chamada de *deep reinforcement learning*, consiste na utilização de algoritmos de *reinforcement learning* em conjunto com *deep learning* para o aprendizado de um agente dentro de um ambiente normalmente estocástico. Ao longo desse trabalho, foram avaliados as influência dos chamados hiperparâmetros, parâmetros definidos pelo treinamento, e de diferentes arquiteturas de redes neurais na aprendizagem do agente em diversos ambientes 2D e 3D. Além disso, foi testada a capacidade de generalização de aprendizagem de um agente que aprendeu a navegar em um ambiente tridimensional ao ser colocado em outro distinto. O processo de aprendizagem foi acompanhado de perto, com a demonstração dos mapas de ativação, filtros e zonas de atenção aprendidas ao longo dos episódios. Em conjunto com tudo isso, foi buscado para a implementação do algoritmo a maximização da velocidade de processamento junto com uma robustez e flexibilidade para rápidas e fáceis experimentações.

Palavras-chave: Controle; Inteligência Computacional; Navegação; Reinforcement Learning; Deep Learning.

Abstract

The following paper constituted in the development of a smart agent for the autonomous exploration of 3D environments using the visual reinforcement learning technique, utilizing the following major fields of the mechatronics engineering: control and computing. Teaching an artificial intelligence how to navigate efficiently throughout complex environments will bring us closer to the use of autonomous robots in our daily life, as for example, autonomous cars. Hence, for evaluation purposes of some of the current techniques that seek for this goal, it was made a study, development and training of an artificial intelligence using the visual reinforcement learning technique to navigate in a tridimensional environment, only having as input the screen pixels of this environment. The technique of visual reinforcement learning, also called of deep reinforcement learning, consists in the utilization of the reinforcement learning algorithms together with the deep learning for the learning of an agent in an environment that is normally stochastics. Throughout this paper, the influence of the hiperparameters, parameters that were defined by the training, and different neural networks architectures were evaluated in the learning of an agent in various 2D and 3D environments. Besides that, the capacity of learning generalization of an agent that learned how to navigate in a tridimensional environment was evaluated when it was placed in a distinct 3D environment. The learning process was closely watched, with the demonstration of the activation maps, filters and attention zones learned throughout the episodes. Together with all that, it was sought for the implementation of the algorithm the maximum computational processing speed along with robustness and flexibility for rapid experimentations.

Key-words: Control; Artificial Intelligence; Navigation; Reinforcement Learning; Deep Learning.

Sumário

Lista de Figuras

Lista de Acrônimos e Notação

RL	Reinforcement Learning (aprendizado por reforço)
IA	Inteligência Artificial
DL	Deep Learning (aprendizado profundo)
NN	Neural Networks (redes neurais)
CNN	Convolutional Neural Networks (redes neurais de convolução)
RNN	Recurrent Neural Networks (redes neurais recorrentes)
LSTM	Long-Short Term Memory Networks (NN de memoria de longo e curto prazo)
SGD	Stochastic Gradient Descent (gradiente descendente estocástico)
ML	Machine Learning (aprendizado de máquina)
GPU	Graphics Processing Unit (unidade de processamento gráfico)
MDP	Markov Decision Process (Processo de decisão de Markov)
DQN	Deep Q-Network
DRQN	Deep Recurrent Q-Network
LR	Learning Rate
APIs	Application Programming Interface
FPS	First Person Shooter (Jogo de tiro em primeira pessoa)

a	um escalar A
$\boldsymbol{a}$	um vetor A
$\boldsymbol{A}$	uma matriz A
$\mathcal{A}$	Conjunto A
A	Subconjunto de $\mathcal{A}$
$\mathbb{R}$	Conjunto dos números reais
p_i	Probabilidade de algum evento ocorrer no momento i
$\mathbb{E}[X]$	Expectativa de uma variável X. $\mathbb{E}[X] \doteq \sum_{i=1}^{\infty} x_i p_i$
$\mathbb{P}[X \mid x]$	Probabilidade da variável x ir para o valor X
$\in$	Pertencente a um conjunto
$\subset$	Subconjunto
$\forall$	Para todos
$\doteq$	Definição
$\{0,1,...,n\}$	Conjunto de todos os inteiros entre 0 e n
a_i	Elemento i de um vetor $\boldsymbol{a}$

Capítulo 1

Introdução

Provavelmente o primeiro pensamento quando questionamos sobre a natureza do aprender é a ideia de que aprendemos pela interação com o ambiente. Quando uma criança nos seus primeiros anos de vida brinca, mexe seus braços ou olha em volta, ela não tem nenhum professor explicando como realizar cada uma dessas tarefas, mas tem uma conexão sensor motora com seu ambiente. Exercitando essas conexões, produzimos informações ricas sobre a causa e o efeito, sobre as consequências das ações e sobre o que fazer de forma a atingir objetivos. Em toda a nossa vida, tais interações são sem dúvida as maiores fontes de conhecimento sobre o nosso mundo e sobre nós mesmos. Logo, aprender por interação é uma ideia fundamental presente em quase todas as teorias de aprendizagem e inteligência[2].

Uma abordagem computacional de aprendizado por interação é o *Reinforcement Learning* (RL), traduzido, literalmente, como aprendizado por reforço. RL consiste em um algoritmo criado com bases em observações psicológicas [3] e neurocientíficas [4] de como seres humanos aprendem, com finalidade de controle e aprendizado de um agente artificial em um ambiente estocástico.

O objetivo do *Reinforcement Learning* é aprender como fazer para maximizar o valor numérico do sinal de premiação ao final de uma tarefa, ou seja, ele deve aprender como mapear situações em ações. Não é dito ao agente quais ações a serem tomadas, mas cabe ele descobrir quais ações geram a maior pontuação ao tentá-las. Nos casos mais interessantes e desafiadores, as ações não afetam somente a recompensa imediata mas também as próximas situações e, assim, todos as próximas recompensas. Essas duas características - tentativa e erro e recompensa com atraso - são as duas características mais importantes e distintas do RL[2].

A maior dificuldade da aplicação do RL e outros algoritmos de inteligência computacional em problemas do mundo real é que muitos fatores externos podem influenciar cada pedaço de dados observado do ambiente. Por exemplo, cada pixel individual de uma

imagem de um carro vermelho pode parecer preto a noite ou como forma da silhueta de um carro muda, dependendo do ângulo de visão. A maioria das aplicações requerem a separação desses fatores de variação e o descarte de características que não são úteis. Entretanto, pode ser muito difícil extrair características de alto nível ou abstratas dos dados brutos. Muitos desses fatores de variação, como por exemplo um sotaque de uma pessoa em um discurso, só podem ser identificados usando dados sofisticados, perto do entendimento humano[5].

Problemas como esses são resolvidos com o *Deep Learning* (DL), traduzido como aprendizado profundo, devido à sua habilidade de construir representações complexas por meio de combinações de conceitos mais simples. Para exemplificar, considere a tarefa de classificar uma imagem, através de seus pixels, em uma classe como um carro. Ao invés de realizar um mapeamento direto de pixels, o aprendizado profundo permite ao computador construir conceitos mais complexos sobre imagem pela união de representações mais simples. Em outras palavras, para definir se uma imagem é um carro ou não, o algoritmo verifica presença de conceitos complexos como pneus ou retrovisores, já esses conceitos foram aprendidos pela união em uma certa ordem de contornos e texturas, que por sua vez também foram aprendidas pela composição de conceitos mais simples como bordas ou borrões. Assim, por sua representação em forma de conceitos hierárquicos, o DL se torna mais robusto aos problemas discutidos no parágrafo anterior. Para realizar o mapeamento mencionado, o aprendizado profundo utiliza-se do seu principal componente: as redes neurais, do inglês *Neural Network* (NN). Uma rede neural é uma função matemática composta de muitas funções mais simples mapeando valores de entrada para valores de saída[5].

A figura 1.1 mostra um exemplo simplificado de uma rede neural. Em uma NN, a entrada é representada como a camada visível *(visible layer*), pois a mesma contém as variáveis que podem ser vistas. Em sequência, temos as camadas escondidas (*hidden layers*) que são responsáveis por extrair características abstratas dos dados. Nestas camadas, o modelo deve determinar quais são os conceitos que são úteis para explicar a relação entre os dados observados. No caso da figura 1.1, a primeira camada escondida é responsável pela identificação bordas, a segunda identifica quinas e contornos e a última detecta partes de objetos. Por fim, pela interação entre a entrada e as camadas escondidas temos a saída, que neste caso é definir se uma imagem é pessoa, carro ou um animal[5].

Outro termo importante e necessário no DL é sua profundidade. Definindo profundidade como o número de camadas em uma NN, no DL as redes neurais possuem várias camadas escondidas, ou seja, são redes neurais profundas *(deep neural networks).* Quanto maior a profundidade do modelo, maior será a quantidade de abstração que o mesmo conseguirá extrair dos dados de entrada para definir suas saídas[5].

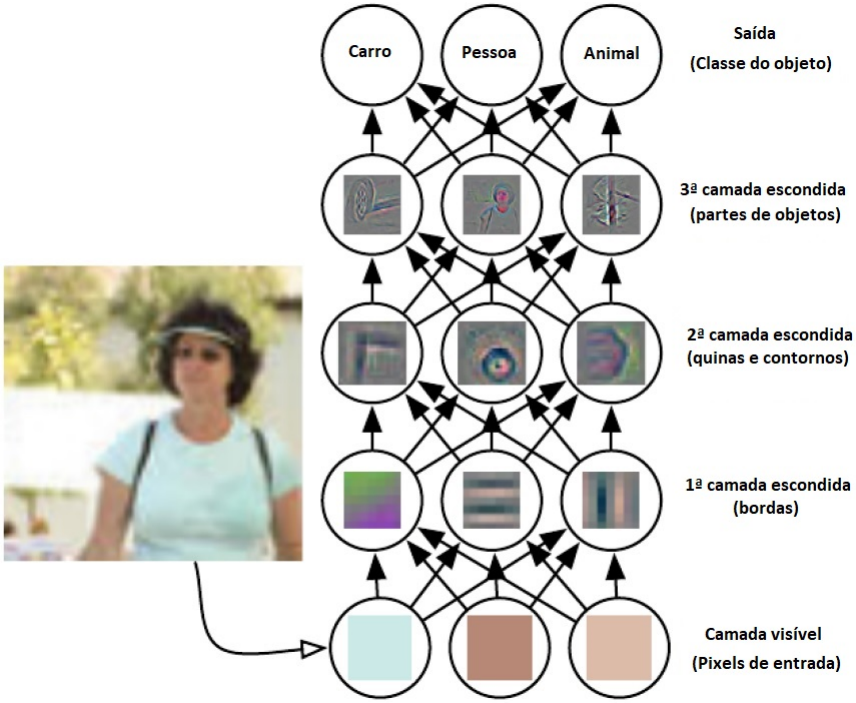

Figura 1.1: Exemplo simplificado da representação de um objeto dentro de uma rede neural. Adaptado de: Goodfellow et al. [5, pg. 6]

Ao se juntar *Reinforcement Learning* e o aprendizado por reforço, foi feito um pequeno passo rumo ao objetivo de desenvolver uma inteligência computacional genérica. Uma inteligência computacional será dita genérica quando com o mesmo algoritmo conseguirmos realizar tarefas intelectuais diversas do nível de um ser humano [6]. Seguindo com esse conceito, podemos destacar o artigo publicado na revista Nature em 2015 sob o título de ***Human-level control through deep reinforcement learning***, que desenvolveu um agente que foi capaz de aprender como jogar vários jogos de Atari 2600 possuindo como entrada apenas os pixels da imagem da tela. Mesmo com a variedade de estilos de jogos, o algoritmo conseguiu aprender a jogar e superar em grande parte desses jogos um jogador humano. Essa generalização foi possível pela utilização de imagens como representação dos estados do ambiente, como se o agente visualizasse o mundo do jogo como um ser humano[1].

A figura 1.2 mostra a arquitetura básica da rede neural de convolução, do inglês *Convolutional Neural Networks* (CNN), usada por Mnih et al. [1] para mapear os estados (imagens da tela do jogo de Atari 2600) para ações. As ações, então, eram escolhidas com base no algoritmo de RL conhecido como *Deep Q-learning*.

Entretanto, no trabalho anterior o agente possuía conhecimento total do ambiente, o que não acontece em tarefas mais perto do mundo real, como a navegação em um mundo tridimensional. Nesse tipo de ambiente, o agente limitado pelo seu ângulo de visão possui apenas conhecimento parcial do seu mundo, resultando em um fator dificultante para

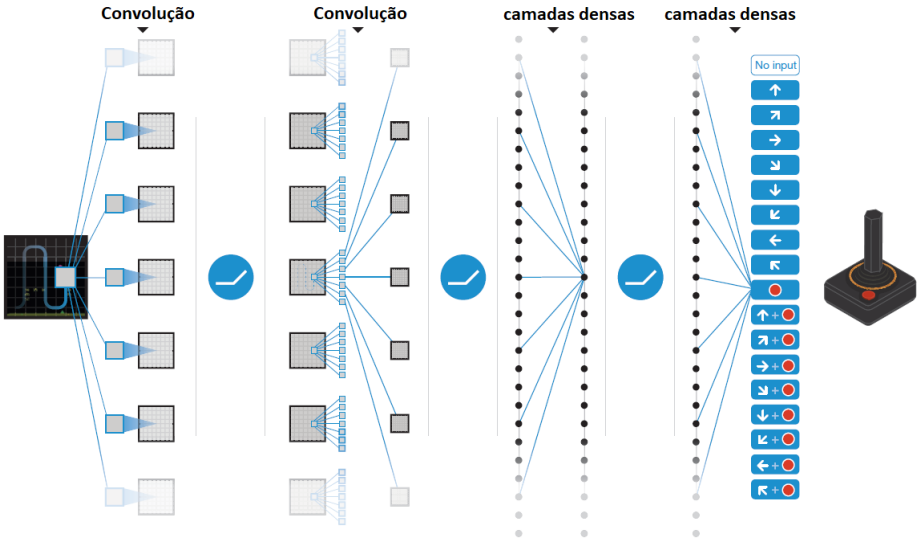

Figura 1.2: Esquemático da rede neural de convolução utilizada para mapear o estados (imagens da tela) para as ações no algoritmo DQN. Adaptado de: Mnih et al. [1]

tomada de ações consideradas ótimas.

Um dos motivos para tamanha dificuldade são as características do ambiente em possuir premiações escassas, há casos nos quais o objetivo pode estar em somente uma localização. Outra razão é que o ambiente abrange elementos dinâmicos, exigindo que o agente use memória em diferentes escalas de tempo: memória instantânea para localização do objetivo, junto com memória recente para integração temporal dos sinais de velocidade e as observações visuais, e, memória de longa duração para aspectos constantes do ambiente[7]. Além disso, a limitação em seu ângulo de visão pode dificultar a localização dos objetivos, por exemplo alguns objetos podem estar escondidos por outros, ou atrás do agente, ou até mesmo a quilômetros de distância.

Logo, conseguir ensinar uma inteligência computacional a navegar em ambientes com tamanha complexidade e de forma eficiente será mais um passo para criação de uma inteligência computacional genérica e futuras aplicações como robôs autônomos. Para fins de avaliar algumas das tecnologias atuais que buscam esse objetivo, foi realizado um estudo, desenvolvimento e treinamento de uma inteligência computacional via *visual reinforcement learning* para a navegação ambientes virtuais tridimensionais, possuindo como entrada os pixels da tela destes ambientes. A técnica de *visual reinforcement learning* também chamada de *deep reinforcement learning*, consiste na utilização de algoritmos de *reinforcement learning* em conjunto com *deep learning* para o aprendizado de um agente dentro de um ambiente normalmente estocástico. Ao longo desse trabalho, foram avaliados as influência dos chamados hiperparâmetros, parâmetros definidos pelo treinamento, e de diferentes arquiteturas de redes neurais na aprendizagem do agente em diversos ambientes 2D e 3D. Além disso, foi testada a capacidade de generalização de aprendizagem de um

agente que aprendeu a navegar em um ambiente tridimensional ao ser colocado em outro distinto. O processo de aprendizagem foi acompanhado de perto, com a demonstração dos mapas de ativação, filtros e zonas de atenção aprendidas ao longo dos episódios. Em conjunto com tudo isso, foi buscada para a implementação do algoritmo a maximização da velocidade de processamento junto com uma robustez e flexibilidade para rápidas e fáceis experimentações.

Neste trabalho foram utilizados da área de controle a parte de processamento de sinais para o pré-processamento de imagens e a técnica de controle *Reinforcement Learning*. Já a grande área computação foi englobada pelas aplicações de redes neurais e otimização dos hiperparâmetros.

1.1 Paradigma

Devido à falta de literatura em português sobre o tema, a maioria dos termos utilizados ao longo desse trabalho foram mantidos em inglês, tendo suas traduções feitas quando conveniente. Os termos mantidos em inglês também possuem a função de facilitar a leitura dos códigos em anexo. Como as bibliotecas implementadas neste trabalho possuem a terminologia de seus parâmetros baseados no conceitos em inglês, isso torna a leitura do código mais didática. Além disso, os termos importantes sobre cada tópico estarão em negrito a primeira vez que forem citados. Com isso, objetiva-se evidenciar importância dos mesmos ao longo do trabalho.

1.2 Definição do Problema

O problema consiste em avaliar o desempenho de arquiteturas e parâmetros distintos na aprendizagem de uma inteligência computacional para a tomada de decisões consideradas ótimas em um espaço tridimensional estocástico possuindo como entrada somente os pixels da tela deste ambiente. Um fator dificultante é que essas imagens de entrada serão constituídas somente pela visão do agente e não a visão total do ambiente. Além disso, outro problema levantado é a capacidade de generalização de aprendizado de um agente que foi ensinado em um ambiente ao ser colocado em um novo.

Para confrontar esses problemas, serão testados diversos ambientes 2D com algoritmos de *reinforcement learning* e *visual reinforcement learning*. Além disso, serão construídos dois ambientes tridimensionais virtuais levemente distintos com o objetivo de ensinar ao agente a atingir uma localização, pensada como uma base de recarga de um robô autônomo, em menor tempo possível e de forma a evitar colisões. Por fim, com o objetivo de verificar sobre a generalização de aprendizagem, após o aprendizado da navegação no primeiro ambiente, o agente será realocado para o segundo.

1.3 Motivação

Ensinar uma inteligencia computacional a navegar de forma eficiente em ambientes complexos nos trará a um passo mais próximo do uso de robôs inteligentes em nosso dia a dia, como por exemplo carros autônomos. Assim sendo, o melhor conhecimento das qualidades e limitações dos algoritmos de *reinforcement learning* atuais se faz necessária para o melhoramento dos mesmos. Além disso, o autor desse trabalho aprofundou seus conhecimentos em inteligencia computacional, área na qual o mesmo pretende seguir carreira acadêmica.

1.4 Objetivos do Trabalho

A seguir serão apresentados os objetivos desse projeto.

1.4.1 Objetivos Gerais

Avaliar o desempenho e a capacidade de generalização de aprendizagem de arquiteturas e parâmetros distintos no treinamento de uma inteligência computacional, utilizando algoritmos do estado da arte em *reinforcement learning* e *deep learning*, para navegação em ambientes tridimensionais via simulação possuindo como entrada os pixels da tela destes ambientes.

1.4.2 Objetivos Específicos

A seguir são apresentados os objetivos específicos desta pesquisa:

- Desenvolver 2 ambientes tridimensionais para o aprendizado do agente.
- Modelar todos ambientes 2D e 3D testados como um Processo de decisão de Markov, do inglês *Markov Decision Process* (MDP);
- Realizar o processamento das imagens de entrada;
- Desenvolver os algoritmos de *visual reinforcement learning* para uma maximização de velocidade de processamento e flexibilidade para rápidas experimentações.
- Avaliar e otimizar os hiperparâmetros (parâmetros específicos de treinamento) para um aprendizado mais rápido e robusto;
- Acompanhar o aprendizado das redes neurais, parte essencial dos algoritmos de aprendizado, através dos filtros aprendidos, mapas e zonas de ativação.
- Avaliar o desempenho na velocidade de aprendizado e capacidade de generalização de diferentes arquiteturas de redes neurais.

1.5 Organização do Documento

Este trabalho está dividido em cinco capítulos. O presente capítulo apresenta a terminologia usada, definição do problema estudado, a motivação para a realização do projeto, os objetivos do trabalho e a organização deste documento.

No segundo capítulo, nomeado Fundamentos, estão dispostas a revisão de literatura, o estado da arte, a metodologia e a fundamentação teórica do trabalho.

Já no terceiro capítulo, chamado Desenvolvimento, é apresentado o ambiente tridimensional usado pelo agente, a API escolhida para o uso do *deep learning*, o desenvolvimento dos algoritmos de RL usados, as etapas de pré-processamento das imagens, a criação dos mapas tridimensionais.

No quarto capítulo, nomeado Processo de aprendizagem, são apresentados as modelagens em forma de MDP dos ambientes Pong e dos dois mapas tridimensionais criados na plataforma VizDoom e as análises do processo de aprendizagem do agente nestes ambientes.

O quinto capítulo retrata às considerações finais e as perspectivas de trabalho futuros.

Capítulo 2

Fundamentos

Neste capítulo são mostrados a revisão da literatura, o estado da arte e a fundamentação teórica necessária ao trabalho.

2.1 Revisão de Literatura

A abordagem via *Visual Reinforcement Learning* é uma tecnologia recente, resultando da confluência de diferentes áreas do conhecimento: *Deep Learning* (DL) e *Reinforcement Learning* (RL).

As primeiras arquiteturas de *Deep learning* surgiram na década de 1940 motivados por uma perspectiva neurocientífica. O Neurônio de McCulloch-Pitts foi o primeiro modelo das funções cerebrais. Era um modelo linear capaz de reconhecer duas categorias distintas de entrada $\boldsymbol{x}$ testando quando $f(\boldsymbol{x},\boldsymbol{w})$ era positivo ou negativo. No entanto, o modelo não aprendia por si mesmo seus pesos $\boldsymbol{w}$, sendo necessário ao operador humano defini-los[5].

Na década de 1950 o Perceptron se tornou o primeiro modelo que podia aprender seus pesos para cada categoria dentro dos exemplos. Aproximadamente na mesma época, o ADALINE (*adaptive linear element*) era capaz de aprender a predizer números reais com base nos dados de entrada. O algoritmo de treinamento usado para aprender os pesos do ADALINE era um caso especial do algoritmo *stochastic gradient descent* (SGD). Versões levemente modificadas do SGD são os principais algoritmos de treinamentos dos modelos de *deep learning* atualmente. Entretanto, modelos lineares como o perceptron e o ADALINE possuem muitas limitações. Como o próprio nome diz, modelos desse tipo só podem aprender representações lineares dos dados. Essa e outras limitações fizeram as redes neurais e algoritmos inspirados na biologia caírem em desuso por um tempo, terminando assim a primeira grande onda do *deep learning*.[5].

Nas décadas subsequentes tivemos avanços com a criação de novas arquiteturas que não possuíam as limitações dos sistemas lineares. Na década de 1980 tivemos o começo da segunda grande onda de popularidade do DL, com a criação de vários conceitos chave que

são utilizados até hoje. Essa grande onda surgiu em parte pela criação do processamento distribuído paralelamente (*parallel distributed processing*) ou *connectionism*. A ideia central desse conceito é que um grande número de unidades computacionais simples podem atingir comportamento inteligente quando conectadas em conjunto. Esse ponto de vista se aplica igualmente aos neurônios do nosso sistema nervoso e as unidades escondidas nos modelos computacionais[5].

Na mesma década tivemos a criação do Neocognitron, que introduziu uma poderosa arquitetura para o processamento de imagens. Este modelo foi inspirado na estrutura de funcionamento do sistema visual dos mamíferos e posteriormente se tornou a base para as redes neurais convolucionais, *Convolution Neural Networks* (CNN), modernas. Na mesma época tivemos a popularização do algoritmo chamado *back-propagation* para o treinamento de redes neurais com varias camadas chamadas de redes neurais profundas (*deep neural networks*). Atualmente, esse algoritmo é a abordagem dominante para o treinamento de modelos *deep learning*[5].

Durante a década de 1990, pesquisadores fizeram avanços importantes para modelar sequências com as redes neurais recorrentes. Em 1997, foi introduzida a rede *Long Short Term Memory* (LSTM). A LSTM foi proposta para resolver os problemas em modelar longas sequências de dados. Hoje, a LSTM é largamente usada para tarefas de modelamento de sequências de dados, incluindo grande parte do processamento de linguagem natural feita por empresas como o Google[5].

A segunda grande onda de popularidade do *Deep Learning* durou até o final da década de 1990. A falta da dados suficientes e poder de processamento, promessas ambiciosas e o avanço considerável em outros campos do *machine learning* (aprendizado de máquinas) fizeram a popularidade das redes neurais caírem novamente[5]. É importante notar que neste período a maioria dos trabalhos sobre o poder de aproximação de uma rede neural, utilizavam arquiteturas com apenas uma camada escondida, este tipo de arquitetura hoje é conhecida como *shallow networks* (redes neurais rasas). Dados empiricos da época, mostravam que as redes neurais rasas tinham performance similar as redes neurais profundas, e em comparação exigiam menos poder de processamento[8].

Em 2006 começou a terceira onda de popularidade das redes neurais com avanços feito por George Hinton. Ele mostrou que uma rede neural profunda chamada *deep belief network* podia ser eficientemente treinada usando uma estrategia de treinamento por camada[5]. No entanto, o grande avanço do *Deep Learning* veio em 2011. Com o aumento da velocidade de processamento das placas gráficas, GPUs *(Graphics Processing Unit)*, foi possível o treinamento de redes neurais convolucionais sem ser necessário o pré-treinamento de cada camada. Com esse aumento em velocidade de computação ficou evidente que o *Deep Learning* possuía diversas vantagens de eficiência e velocidade em

relação a outras técnicas de aprendizado de máquina, principalmente em análise de grandes bancos de dados. Isto foi provado em 2012 com a CNN AlexNet ganhando inúmeras competições de visão computacional. Nesta CNN foram usados *Rectified Linear Units* (ReLu) como funções de ativação e a técnica de *Dropout* para aumentar a velocidade e generalização da rede neural[9]. Atualmente a importância teórica da palavra profundidade provinda do termo *deep learning* é um dos conceitos chaves para o desempenho das redes neurais, geralmente quanto mais profundo mais abstrações conseguiremos extrair dos dados e melhor será precisão da rede neural[5].

Quanto a área de *Reinforcement Learning*, podemos dizer que a mesma também é confluência de duas grandes áreas: aprendizado por tentativa e erro e controle ótimo. Talvez a primeira definição sucinta do aprendizado por tentativa e erro seja de Edward Thorndike em 1898. Thorndike definiu o que chamou de a Lei do Efeito na qual descreve quanto maior o estimulo positivo ou negativo com uma ação, maiores serão as chances de um animal repetir ou não a mesma ação, ocorrendo assim o aprendizado[3]. Com base nas linhas da Lei do Efeito, desde a década de 1930, vários dispositivos eletromecânicos foram construídos para demonstração do aprendizado por tentativa e erro. Essas máquinas foram guias para as primeiras implementações de algoritmos de aprendizagem em computadores na década de 1950. Entretanto, com o advento dos algoritmos de generalização e reconhecimento de padrões houve um certo esquecimento dos algoritmos de puro aprendizado por reforço (*reinforcement learning*)[2].

A parte de controle ótimo teve seu inicio década de 1950. O termo controle ótimo surgiu para descrever o problema de desenvolver um controlador para minimizar as medidas de comportamento de um sistema dinâmico no tempo. Uma abordagem a esse problema foi desenvolvida por Richard Bellman e outros através da extensão das teorias do século dezenove de Hamilton e Jacobi. Essa abordagem usava os conceitos do estado de um sistema dinâmico e de uma função de valor, ou função de retorno ótimo, para definir uma equação funcional conhecida como equação de Bellman. Bellman também introduziu a versão discreta do controle ótimo para um ambiente estocástico conhecida como *Markov Decision Processes* (MDPs), ou processos de decisão de Markov. Seguindo essa linha de pesquisa, em 1960 Ronald Howard cria o método de iteração de política (*policy iteration*) para MDPs. Esses métodos são elementos essenciais envolvendo a teoria e os algoritmos do RL moderno[2].

As conexões entre o controle ótimo e o aprendizado por tentativa e erro demoraram a serem descobertas. Somente no final da década de 1980 tivemos uma integração total entre ambas as áreas e a criação do RL moderno através do trabalho de Chris Watkins. Neste artigo foram mostradas evidências das relações entre o controle ótimo e o aprendizado por tentativa e erro, o tratamento matemático do *Reinforcement Learning* através das MDPs

e o desenvolvimento do algoritmo *Q-Learning*[2].

Já no final da década de 90 temos as primeiras aplicações de RL com redes neurais simples[2]. Entretanto, apenas em 2013 vimos os campos de *Deep Learning* e *Reinforcement Learning* confluírem para a criação do *Visual Reinforcement Learning*, ou *deep reinforcement learning*, com o desenvolvimento do algoritmo *Deep Q-Network* (DQN). O trabalho que demonstrou esse algoritmo foi publicado na revista Nature em 2015 sob o título de *Human-level control through deep reinforcement learning*, e desenvolveu um agente que foi capaz de aprender como jogar vários jogos de Atari 2600 possuindo como entrada apenas os pixels da imagem da tela. Mesmo com a variedade de estilos de jogos, o agente conseguiu aprender a jogar e superar em grande parte desses jogos um jogador humano. Essa generalização foi possível pela utilização imagens como representação dos estados do ambiente através do *deep learning*, como se a IA estivesse vendo o mundo do jogo como um ser humano[1]. A figura 2.1 mostra de forma resumida as linhas do tempo das áreas do *Reinforcement Learning* e *Deep learning* até sua confluência para o V*isual Reinforcement Learning/Deep Reinforcement Learning*. Vale notar na linha do tempo do *Deep learning* os períodos chamados de AI winters . Essas épocas, traduzidas literalmente como inverno da inteligencia artificial, foram os períodos de estagnação entre as grandes ondas citadas anteriormente.

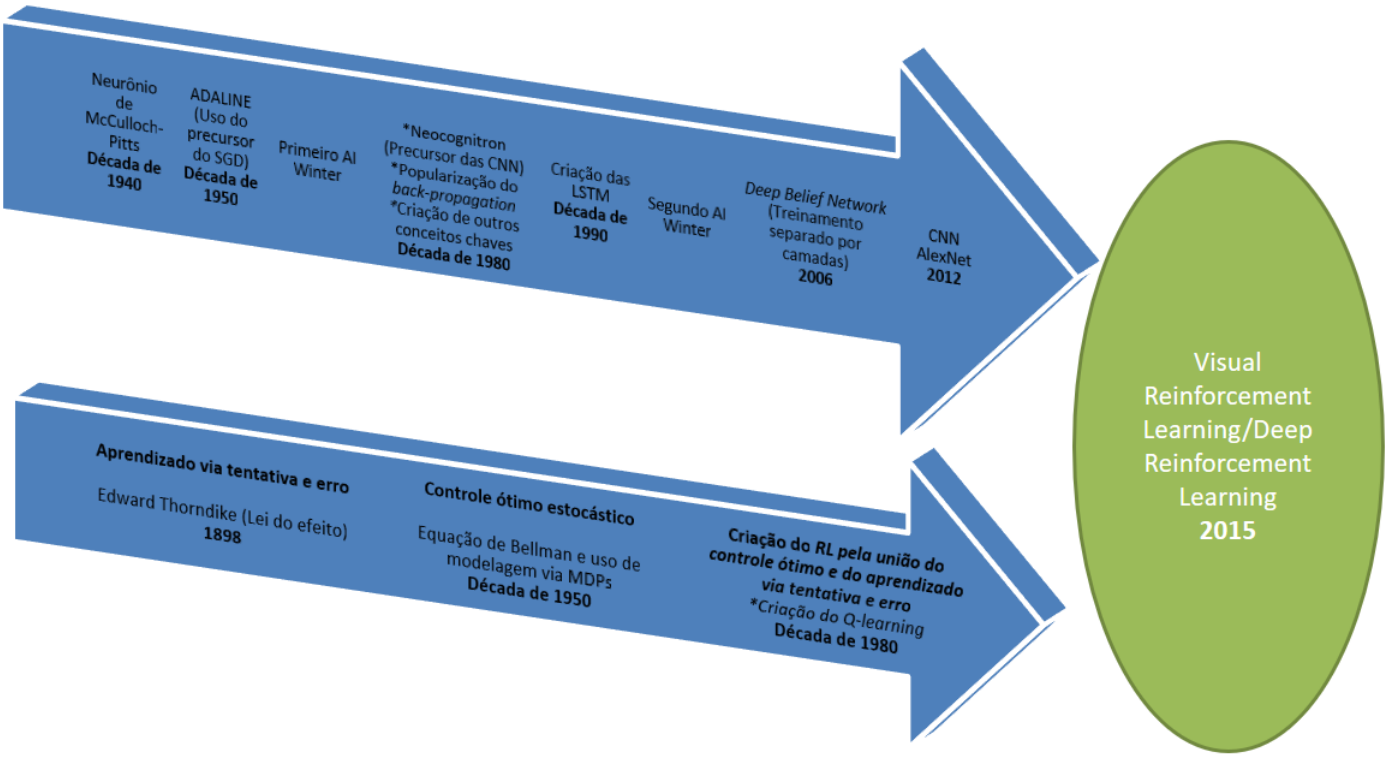

Figura 2.1: Linha do tempo resumida de ambos *Reinforcement Learning* e *Deep Learning* até sua união para a criação do *Visual Reinforcement Learning*.

Neste contexto, vemos que a abordagem via *Visual Reinforcement Learning*, alvo deste

trabalho, é um campo relativamente novo que veio da união de diversas áreas do conhecimento e é um campo de pesquisa ativo.

2.2 Estado da arte

Na literatura, nos últimos anos, o campo de *Visual Reinforcement Learning/Deep Reinforcement Learning* é uma área de pesquisa ativa, incluindo sua aplicação para navegação autônoma de agentes.

Em 2015 o algoritmo *Deep Q-networks* (DQN) trouxe uma quebra de paradigma com seus excelentes resultados ao ensinar um agente a jogar jogos de Atari 2600 no nível superior ou igual a um ser humano [1]. Entretanto, DQN são limitadas no sentido que elas aprendem o mapeamento de estados para ações com um número limitado de estados passados. Além disso, nesse tipo de mundo, o agente tem visão total do estado atual do ambiente, o que normalmente não é verdade em cenários no mundo real.

Em 2015, Hausknecht and Stone [10] adicionou uma rede neural recorrente (RNN) do tipo *Long-Short-Term-Memory* (LSTM) ao algoritmo DQN para jogar jogos de Atari 2600 introduzindo assim a *Deep Recurrent Q-Network* (DRQN). Foerster et al. [11] em 2016 considerou o cenário multi-agente no qual foi usado rede neural recorrente distribuída para a comunicação de diferentes agentes em ordem a atingirem seus objetivos. O uso de uma RNN é efetivo em ambientes nos quais os estados são parcialmente observáveis devido a sua habilidade de lembrar informações por um longo tempo. No entanto, ambos os trabalhos foram feitos em ambientes 2D, o que ocorre raramente no mundo real.

Já em 2016, Mirowski et al. [7] obtiveram progresso no uso de modelos *deep learning* para a localização. Eles mostraram que ao adicionar um conjunto de camadas do tipo LSTM o agente conseguia se auto-localizar em um ambiente tridimensional. O modelo também usou de diversos objetivos auxiliares, como a predição de profundidade e a detecção se o agente está preso em um loop, para auxiliar no treinamento. Por fim, o agente conseguiu com sucesso navegar dentro de labirintos tridimensionais. Além disso, os estados escondidos aprendidos pelo modelo mostraram-se precisos em predizer a posição do agente, embora as camadas LSTM não tenham sido treinadas explicitamente para tal tarefa.

Lample and Chaplot [12] em 2017 mostraram que a performance de um agente em um jogo de tiro, *First-Person Shooter* (FPS), no ambiente VizDoom pode ser substancialmente aumentado com a utilização de objetivos auxiliares. Além disso, neste trabalho os autores introduziram a divisão do modelo em duas partes: navegação (explorar o mapa para coletar itens e achar inimigos) e ação (atirar nos inimigos quando eles são encontrados). Para cada parte foi usada uma arquitetura diferente, para a primeira parte foi

usada DQN e para a segunda uma DRQN e ambas foram treinadas simultaneamente. Em uma competição, o agente foi capaz de ganhar de agentes do jogo e de jogadores humanos em uma partida de mata-mata em um mapa nunca jogado pelo mesmo antes.

Em 2018, Chaplot et al. [13] criaram um modelo para localização global que usa componentes estruturados para um filtro de Bayes, como a propagação de uma crença na qual a política será aprendida para se localizar precisamente de forma eficiente. Além disso o agente treinado com esse modelo foi capaz de realizar a adaptação de domínio. O agente foi treinado em fases do Doom com texturas aleatórias e foi capaz de navegar em um ambiente tridimensional foto-realístico no ambiente de simulação UNREAL.

No entanto, mesmo que todos os recentes trabalhos apresentados tenham conseguido aprender a navegar com sucesso em seus respectivos ambientes, ainda necessitam de um tempo considerável de treinamento. Logo, técnicas para um aprendizado mais rápido e eficiente são uma linha de pesquisa ativa.

2.3 Fundamentação Teórica

A presente seção descreve a fundamentação teórica necessária à realização do trabalho. A princípio será descrita a base teórica por trás do *Reinforcement Learning*, em sequência será feita a fundamentação teórica sobre as redes neurais e seus tipos e por fim será mostrado os algoritmos provindos da fusão de ambas etapas mostradas anteriormente.

2.3.1 Reinforcement Learning

Reinforcement learning (RL) se refere a tarefa de aprender a controlar um sistema (normalmente estocástico) de forma a maximizar algum valor numérico que represente um objetivo de longo prazo. Neste tipo de tarefa, o controlador recebe o estado controlado do sistema e o *reward* (premiação) associado com a última transição de estados. Então ele calcula uma ação a ser tomada sobre o sistema e em resposta o sistema realiza uma transição a um novo estado e o ciclo é repetido. Logo, o problema é aprender a controlar o sistema de forma a maximizar o *reward*[14]. A figura 2.2 demonstra como reinforcement learning tipicamente funciona.

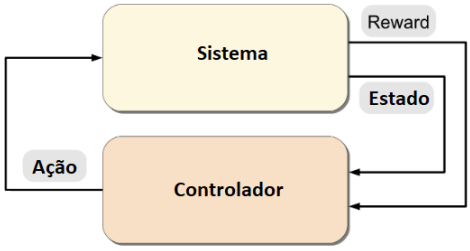

Figura 2.2: Exemplo de funcionamento do RL. Adaptado de: Szepesvári [14]

2.3.2 *Markov Decision Process* (MDP)

Problemas com as características apresentados anteriormente são formulados matematicamente como uma *Markov Decision Process (MDP)*. MDPs podem ser discretas ou contínuas no tempo, como nesse trabalho utilizaremos apenas tarefas que são discretas no tempo, adotaremos o abuso de notação e iremos nos referir a tais MDPs discretas apenas como MDPs. Neste framework matemático, o controlador é conhecido como o **agente** e o sistema no qual ele interage é conhecido como seu **ambiente**. Mais especificamente, o agente e o ambiente interagem em uma sequência temporal discreta $t = \{0,1,2,...,T\}$, no qual, em cada intervalo de tempo (*time step*) **t**, o agente recebe alguma representação do **estado do ambiente** (*state*), $S_t \in \mathcal{S}$, e toma uma **ação**, $A_t \in \mathcal{A}$. Um *time step* depois, como consequência de suas ações, o agente recebe uma premiação numérica chamada ***reward***, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ encontrando-se em um estado S_{t+1}[2]. A figura 2.3 demonstra a interação descrita acima entre o agente e seu ambiente como uma MDP.

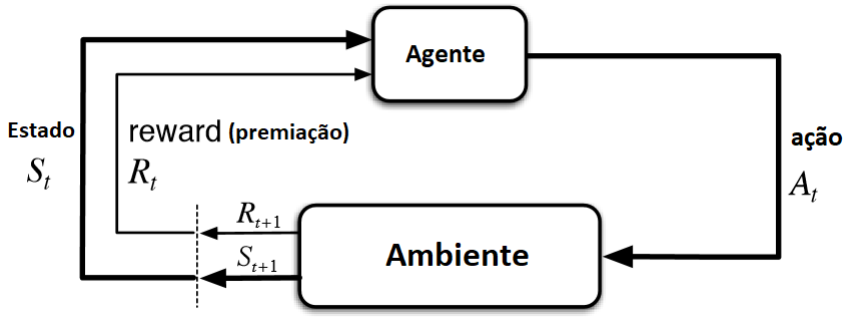

Figura 2.3: Interação entre o agente e o ambiente em uma MDP. Adaptado de: Sutton and Barto [2]

Uma MDP é caracterizada por possuir a propriedade de Markov em seus estados. Um estado será considerado de Markov, se e somente se:

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, S_2, \cdots, S_t] \tag{2.1}$$

A equação implica que $S_t = S_1, \cdots, S_t$, ou seja, um estado é considerado sendo de Markov, se ele captura todas as informações relevantes do histórico. Assim sendo, uma vez que o estado é conhecido, todo o histórico pode ser descartado. Logo, a probabilidade de cada valor possível de S_{t+1} e R_{t+1}, depende somente do estado e da ação anterior, S_t e A_t[2].

2.3.3 Equação do retorno e tarefas periódicas

Aplicações no qual a noção de um final faz sentido, ou seja, no qual a interação entre agente-ambiente se quebra naturalmente em subsequências chamadas **episódios**, como

jogos, navegações em um labirinto, são referidas como tarefas periódicas. Tarefas com essa natureza são modelas utilizando uma MDP finita. Nesse tipo de MDP, o episódio termina quando o agente atinge o estado terminal da tarefa, sendo então levado ao estado inicial ou a uma amostra da distribuição normal dos estados iniciais. A equação 2.2 mostra o formato geral de um MDP finita[2].

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \cdots, S_{T-1}, A_{T-1}, R_T, S_T \tag{2.2}$$

No qual T é o ultimo *time step* e S_T é o estado final de um episodio. O tempo de termino T, é uma variável aleatória que normalmente varia de episódio para episódio. Já, o retorno total ao final de um episódio é dado pela equação 2.3.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.3}$$

Ou seja G_t representa todos os *rewards* que o agente ganhará a partir de um determinado estado S_t até terminar o episódio. Entretanto, considerar os *rewards* de um estado até o final do episódio pode ser inviável, principalmente quando começarmos a tratar com as expectativas de G_t logo a adiante. Para isso, inserimos na equação 2.3 uma variável γ chamada ***discount rate***, no qual $0 \leq \gamma \leq 1$. A equação 2.4 mostra G_t com o *discount rate* aplicado.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.4}$$

no qual podemos ter $T = \infty$ (abuso de notação para uma tarefa sem fim) ou $\gamma = 1$, nunca ambos juntos.

A *discount rate* determina a presença dos valores dos futuros *rewards* em G_t: um *reward* ganho após k *time steps* no futuro valerá apenas γ^{k-1} vezes o que ele valeria se fosse recebido imediatamente. Se $\gamma = 0$, o agente pode ser dito "míope", apenas se preocupando em maximizar a premiação imediata, em outras palavras, escolher não abastecer um carro, pois continuar a corrida naquele momento gera *rewards* imediatos. Quando γ se aproxima de 1, a função de retorno considera mais os futuros *rewards* na hora de tomar suas ações, por exemplo, abastecer um carro para a gasolina não acabar antes do final da corrida[2].

Os retornos em sucessivos *time steps* são relacionados entre si em uma forma que é importante para a teoria e os algoritmos de reinforcement learning. Essa relação de **recursividade** é dada pela equação 2.5 e facilita para calcular os retornos de sequencias

de *rewards*.

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.5}$$

2.3.4 Políticas (*Policies*) e Funções de valor (*Value Functions*)

Quase todos algoritmos de reinforcement learning envolvem o cálculo das ***value functions***, funções de valor, dos estados (ou do par estado-ação) que estimam o quão bom é para o agente estar em um determinado estado, ou o quão bom é tomar determinada ação em determinado estado. Essa noção é definida em termos dos futuros *rewards* esperados, ou, para ser mais preciso, em termos da expectativa do retorno. Os *rewards* que o agente espera ganhar no futuro dependerão em quais ações ele tomar. Assim, as *value functions* são definidas com respeito ao jeito particular de agir do agente, chamado de *policies*.

Formalmente, uma ***policy***, ou politica de ação, é um mapeamento de estados para as probabilidades de selecionar cada ação possível. Se o agente esta seguindo a *policy* π no tempo t, então $\pi(a|s)$ é a probabilidade que $A_t = a$ se $S_t = s$, ou seja, que a ação a será selecionada no estado s atual. Por exemplo, uma *policy* pode ser sempre tomar as ações aleatoriamente, ou sempre andar para frente (sempre pressionar a ação {andar para frente}) não importando as circunstâncias em um labirinto[2].

Como tudo isso, definimos uma *value funcion* de um estado s sob uma *policy* π, denotada como $v_\pi(s)$, como o retorno esperado (expectativa do retorno) em s seguindo π. Para MDPs, definimos v_π formalmente pela equação 2.6.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \;\middle|\; S_t = s\right], \text{para todo } s \in \mathcal{S} \tag{2.6}$$

no qual $\mathbb{E}_\pi[X]$ é a operação de expectativa de uma variável aleatória vinda da estatística. Essa operação calcula a média a longo prazo de valores em repetição em um experimento. Neste caso, em uma operação que denota o valor esperado de uma variável X com o agente seguindo uma política de ações π. A função v_π é chamada *state-value function* para a *policy* π, ou seja, a função que define ao agente **o quão bom é estar naquele estado seguindo uma determinada política de ação π.** A figura 2.4 demonstra o valor de v_π para cada estado s, em um simples labirinto em duas dimensões. Neste exemplo, para cada passo do agente ele recebe um *reward* de -1, o *discount rate* é 1 e cada grade do labirinto é um estado. Logo, se o agente fosse jogado no retângulo azul, a sua expectativa, ao estar naquele lugar, de um *reward* total ao final do episodio será de -6. Da mesma forma, ao estar no retângulo vermelho, ele espera ganhar -3 de *reward* a partir daquele ponto.

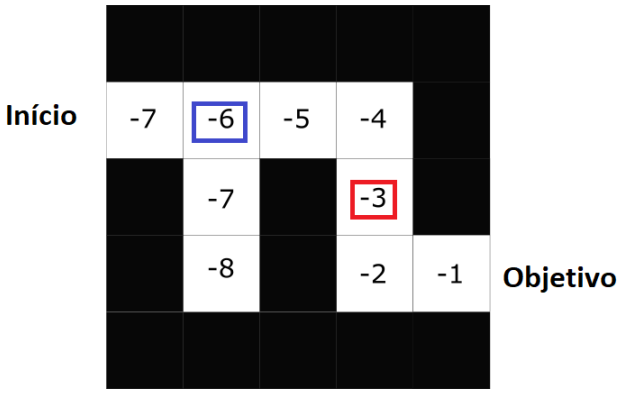

Figura 2.4: Exemplo de valores de v_π para cada estado em uma MDP. Adaptado de: Simonini [15]

Analogamente, definimos o valor da tomada de uma ação a em um estado s sob uma *policy* π denotado como $q_\pi(s,a)$, como sendo o retorno esperado começando no estado s, tomando a ação a com a politica π. Definimos q_π formalmente pela equação 2.7.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right], \text{para todo } s \in \mathcal{S} \tag{2.7}$$

Chamamos q_π de *action-value function*, ou *Q-function (quality function)* para a *policy* π, ou seja, com essa função definimos **o quão bom é o valor de cada ação** a **naquele determinado estado** s. No exemplo mostrado na figura 2.4, se o agente está no retângulo vermelho, os valores de q_π para a ação {andar para o lado esquerdo} é de -5, e para ação {andar para baixo} é de -7.

O valor das funções v_π e q_π são estimadas pela experiência do agente em seu ambiente devido a lei básica da estatística chamada de Lei dos Grandes Números. Por exemplo, se um agente seguir uma politica π e manter uma média, por cada estado encontrado, dos retorno obtidos após seguir aquele estado, a média convergirá ao valor daquele estado, v_π, a medida que o número de vezes que aquele estado for encontrado aproximar de infinito. De forma similar, se mantermos as médias para cada ação tomada em cada estado, as médias convergirão para os valores daquelas ações $q_\pi(s,a)$. Logo, o exemplo da *value function* mostrado anteriormente, todos os valores começaram em zero e a medida que o agente explorou o ambiente ele atualizou **recursivamente** o valor de cado estado.

Entretanto, se houver muitos estados na MDP fica impraticável manter os valores de média para cada estado na memória. Neste casos, o agente utilizará funções parametrizadas (Softmax, redes neurais densas, CNN) para estimar os valores de v_π e q_π e ajustar os parâmetros (através de *back-propagation*) para corresponder aos valores retornados pelo ambiente[2].

2.3.5 Funções ótimas

Podemos dizer grosseiramente, que resolver o problema de RL, significa encontrar uma política de ações que obtenha uma grande premiação ao final de uma tarefa, em outras palavras, desejamos descobrir quais são as melhores ações a serem tomadas em cada momento. Para MDPs finitas, podemos definir uma *policy* ótima da seguinte forma. Uma *policy* π é dita ser melhor ou igual a uma *policy* π' se o seu retorno esperado é maior ou igual ao de π' para todos os estados. Em outras palavras, $\pi \geq \pi'$ se e somente se $v_\pi(s) \geq v_{\pi'}(s)$ para todos $s \in (S)$. Sempre existirá no mínimo uma *policy* que é melhor ou igual a todas as outras policies, a essa *policy* chamamos de ***optimal policy***, ou a política de ações ótima[2]. Denotamos todas as *optimal policies* como π_*. As *optimal policies* compartilham a mesma ***optimal action-value function***, definida como q_*, e definida pela equação 2.8.

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a),\ \forall\ s \in \mathcal{S}\ ,\ \forall\ a \in \mathcal{A}(s) \tag{2.8}$$

Dessa forma, com a equação 2.9 e utilizando a propriedade explicada pela equação 2.5, podemos definir o que é conhecida como a equação ótima de Bellman, ou ***Bellman optimality equation*** para q_* dada pela equação 2.9.

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(s',a') \;\middle|\; S_t = s, A_t = a \right] \tag{2.9}$$

no qual a' e s' são a ação e o estado no *time step* $t+1$.

Ou seja, a equação 2.9 demonstra com o formalismo matemático, que a melhor ação a a ser tomada em um estado s é a ação que possuir a maior expectativa de retorno ao final de um episódio (valor da função Q). Dessa forma, podemos definir a política de ações ótima como sendo **a *policy* que sempre toma as ações no qual o agente espera receber a maior soma de *reward* ao final do episódio**[2]. Formalmente podemos definir a política ótima pela equação 2.10.

$$\pi_*(s) = arg\max_a q_*(s,a),\ \forall\ s \in \mathcal{S} \tag{2.10}$$

A figura 2.5 demonstra a *policy* ótima, π_*, para o exemplo dado pela figura 2.4. Para cada estado do ambiente, o agente tomará a ação que levará mais rápido ao seu objetivo, gerando *reward* maior ao final do episódio.

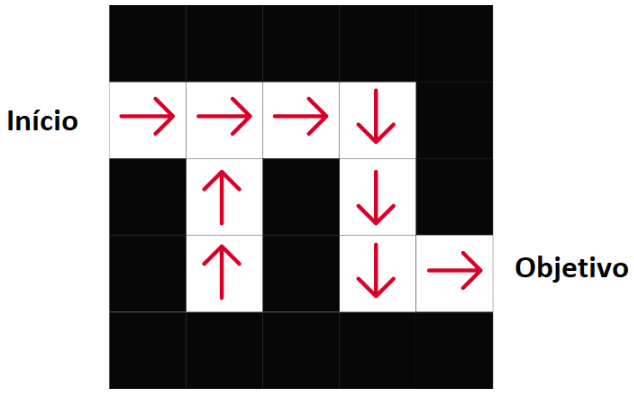

Figura 2.5: Exemplo de policy ótima π_* para cada estado em uma MDP. Adaptado de: Simonini [15]

2.3.6 Os dilemas de exploração: Exploitation e Exploration

A *optimal policy* pode ser chamada de ***greedy policy***, política gananciosa, com respeito ao valor da Q-function. O termo *greedy* é usado na ciência da computação para descrever qualquer processo de busca ou decisão que seleciona alternativas apenas baseado em considerações locais, sem considerar a possibilidade que esse tipo de seleção possa prevenir o acesso a alternativas melhores.

Dito isso, chegamos a um dos grandes desafios para o aprendizado em ambientes sem modelo (como o caso deste trabalho) o balanceamento entre ***exploration*** e ***exploitation***. Para obter uma grande quantidade de *reward*, o agente preferirá ações que ele ja tentou no passado e achou serem as mais efetivas em produzir premiações, a esse comportamento chamamos de *exploitation* (explorar o conhecimento anterior). Entretanto, para descobrir tais ações, ele deve tentar ações que não foram selecionadas antes, a esse comportamento chamamos de *exploration* (explorar o desconhecido).

Para mitigar esse problema, utilizamos a *policy* conhecida como ϵ-*greedy*. Nessa *policy* inserimos um parâmetro ϵ chamado de ***exploration rate***, no qual $0 \leq \epsilon \leq 1$ que será a probabilidade de tomarmos uma ação aleatoriamente dentro do set de ações [16]. A *policy* ϵ-*greedy* funciona da seguinte forma:

- Inicializamos a *exploration rate* ϵ com o valor de 1 (100%), ou seja, no começo todas as ações serão aleatórias. Desta forma, o agente explorará o ambiente atualizando o valor da *Q-function* para os estados visitados. A medida que o agente se torna confiante na estimação dos valores da *Q-function*, decaímos os valores da *exploration rate*.

- A cada estado geramos uma variável aleatória (entre 0 e 1). Caso esse número seja maior que épsilon, o agente realiza uma ação de *exploitation* (*greedy*) caso contrário *exploration*(randômica).

- Depois de um certo tempo zeramos a *exploration rate*, realizando apenas exploitation.

A figura 2.6 demonstra como funciona a *policy* ϵ*-greedy*.

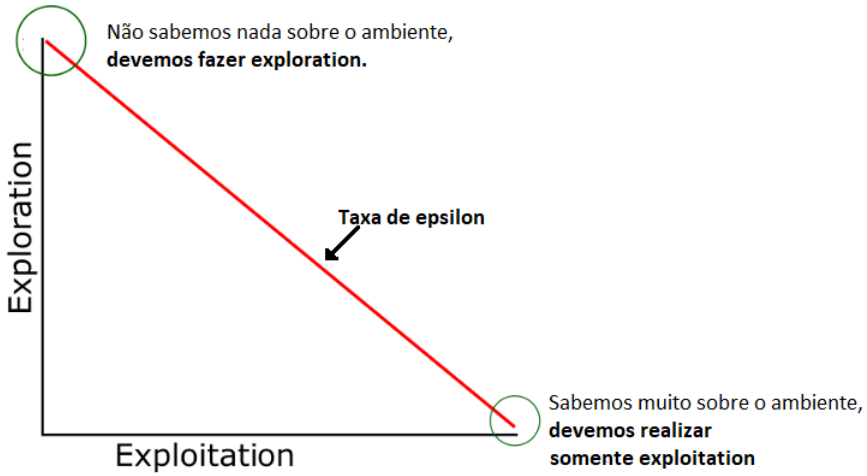

Figura 2.6: Como ajustar o parâmetro ϵ no algoritmo ϵ-greedy. Adaptado de: Simonini [16]

2.3.7 Q-Learning

Um dos grandes marcos do RL foi o desenvolvimento do algoritmo conhecido como *Q-learning*, no qual seu princípio é dado pela equação 2.11.

$$Q(s,a) = Q(s,a) + \alpha\Big[R + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big] \tag{2.11}$$

no qual, utilizaremos R no lugar de R_{t+1} para simplificar a notação neste trabalho. Já, a variável α é chamada de *learning rate*, ou taxa de aprendizado do algoritmo (não confundir com a *learning rate* das redes neurais que será apresentado nas próximas sessões), em que $0 \leq \alpha < 1$. Caso α fosse 1, obteríamos na equação 2.9, atualizando assim, os valores de q sem considerar os valores passados.

Com a 2.11 estamos apenas atualizando o valor de $Q(S_t, A_t)$ em direção a diminuir seu erro com um passo α. A parte $(R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}))$ é o *reward* obtido pela ação tomada somado ao máximo valor descontado de Q do estado para qual o agente foi. Subtraindo desse valor o valor de $Q(S_t, A_t)$ que tínhamos previamente, temos a direção de decremento erro.

Para demonstração de como o algoritmo *Q-learning* funciona, considere o exemplo mostrado na figura 2.7.

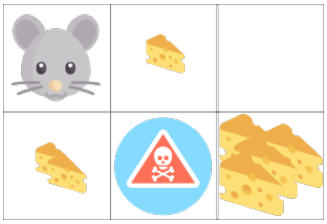

Figura 2.7: Exemplo de aplicação do Q-Learning em um labirinto 2D. Retirado de: Simonini [16]

Neste pequeno labirinto 2D, o agente (rato) deve aprender a conseguir o máximo de queijo possível tendo como estado final a pilha de queijos ou veneno. Para isso, daremos os seguintes *rewards*, para caso o rato consiga comer:

- Um queijo = +1
- Dois queijos = +2
- Pilha de queijos = +10
- Veneno = -10

Neste exemplo, utilizaremos $alpha = 0.1$ e $\gamma = 0.9$, além disso, consideraremos a *Q-function* como uma tabela, chamada *Q-table*, que possui os valores de q para cada ação (coluna) em cada estado (linha). Neste pequeno mundo, temos 6 estados e quatro ações, logo possuímos 24 valores de *Q-function*. A figura 2.8 demonstra a inicialização da *Q-table* segundo o algoritmo *Q-learning*.

	←	→	↑	↓
Início	0	0	0	0
Um Queijo pequeno	0	0	0	0
Nada	0	0	0	0
Dois queijos pequenos	0	0	0	0
Morte	0	0	0	0
Pilha de queijo	0	0	0	0

Figura 2.8: Inicialização da *Q-table* para o exemplo do labirinto 2D. Adaptado de: Simonini [16]

Em sequência, o agente realiza uma ação seguindo a política ϵ-*greedy* (aleatória nesse momento) indo para direita do estado de *start*, sendo levado para o estado de 'Um queijo' e recebendo o *reward* de +1, como demonstrado na figura 2.9.

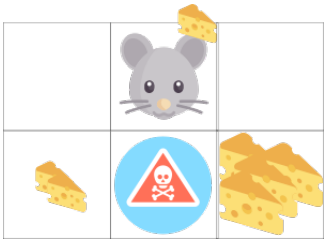

Figura 2.9: Exemplo do estado do labirinto 2D após o agente tomar uma ação. Retirado de: Simonini [16]

Em posse destes dados, realizamos o calculo do valor $Q(start, \rightarrow)$ como demonstrado na equação 2.12.

$$
\begin{aligned}
Q(start, \rightarrow) &= Q(start, \rightarrow) + \alpha\Big[R(start, \rightarrow) + \gamma \max Q'(UmQueijo,a')) - Q(start, \rightarrow)\Big] \\
&= 0 + 0.1\Big[1 + 0.9 * \max\Big(Q'(UmQueijo, \leftarrow), Q'(UmQueijo, \rightarrow), \\
&\qquad\qquad Q'(UmQueijo, \uparrow), Q'(UmQueijo, \downarrow)\Big) - 0\Big] \\
&= 0 + 0.1 * [1 + 0.9 * 0 - 0] \\
&= 0.1
\end{aligned}
\tag{2.12}
$$

Com o valor calculado, atualizamos a *Q-table* como demonstrado na figura 2.10.

	←	→	↑	↓
Início	0	**0.1**	0	0
Um Queijo pequeno	0	0	0	0
Nada	0	0	0	0
Dois queijos pequenos	0	0	0	0
Morte	0	0	0	0
Pilha de queijo	0	0	0	0

Figura 2.10: Atualização da *Q-table* para o exemplo do labirinto 2D. Adaptado de: Simonini [16]

Dessa forma, executamos esses passos descritos acima até o fim do episódio, em um *loop* de vários episódios até que o agente tenha aprendido a maximizar o número de queijos obtidos e a evitar o veneno.

2.3.8 Funções parametrizadas

Com dito na sessão de *Reinforcement Learning*, se houver muitos estados em uma MDP (*Markov Decicision Process*) fica impraticável manter os valores q para cada estado

na memória, teríamos algo como uma *Q-table* infinita. Neste casos, o agente utilizará funções parametrizadas como classificadores lineares, redes neurais densas, redes neurais de convolução para estimar os valores de v_π e q_π e ajustar os parâmetros através do algoritmo *back-propagation*) para corresponder aos valores retornados pelo ambiente[2].

A seguir será mostrado todo o problema de classificação de imagens e suas soluções utilizando diversos tipos de funções parametrizadas. Por fim, será apresentado o algoritmo *Deep Q-learning* que une o *reinforcement learning* com as funções parametrizadas, em mais especifico as redes neurais de convolução. Este algoritmo utilizará dos mesmo princípios apresentados sobre os problemas de classificação, só que ao invés de classificarmos uma imagem em uma classe como gato ou cachorro, classificaremos uma imagem em uma ação a ser tomada, como por exemplo, virar a direita ou a esquerda.

2.3.9 Classificação de Imagens

A maioria das pessoas, sem muita dificuldade, consegue ver que a imagem 2.11 mostra três gatos diferentes. Possuímos pouca dificuldade devido ao fato que em cada hemisfério do nosso cérebro, há um Córtex Visual Primário, também conhecido como V1, contendo aproximadamente 140 milhões de neurônios, com dezenas de bilhões de conexões entre eles. No entanto, a visão do ser humano não envolve apenas o V1, e sim uma série de córtex visuais -V1, V2, V3, V4 e V5 - fazendo progressivamente e hierarquicamente o processamento de imagens de forma mais complexa. Entretanto, quase todo este trabalho é feito de forma inconsciente, omitindo assim o quão complexo é uma tarefa, de por exemplo, reconhecer um gato em uma imagem. A dificuldade de reconhecimento de padrões fica aparente se tentamos escrever um programa para fazer tal tarefa[17].

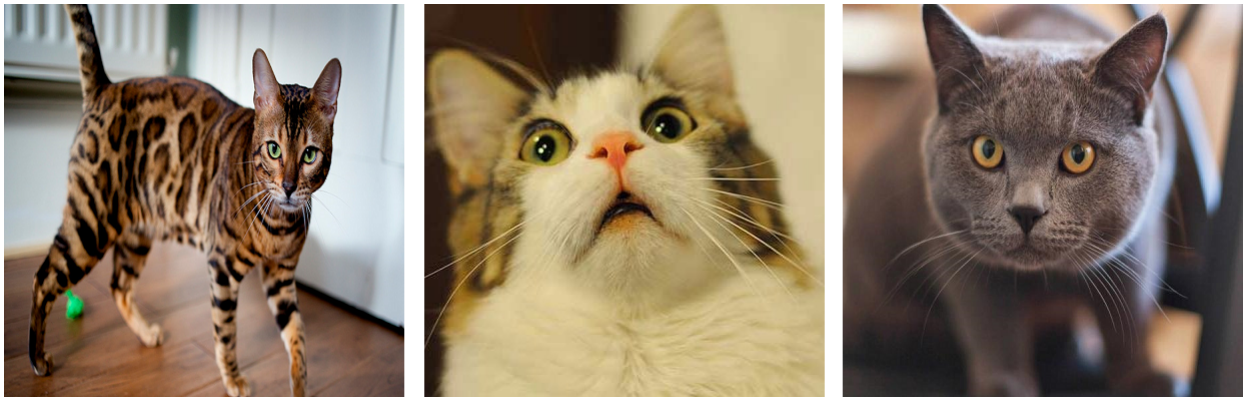

Figura 2.11: Imagem de três diferentes gatos. Retirado de: Google Imagens

Por exemplo considere a imagem 2.12 como sendo um modelo de classificação que toma como entrada uma imagem e tem como saída a distribuição probabilística de 4 classes diferentes gato, cachorro, chapéu e caneca. Como mostrado na figura 2.12, para

um computador uma imagem é uma grande matriz de três dimensões. Nesse exemplo, o gato é uma imagem de 248 pixels de largura, 400 pixel de altura, e tem 3 canais de cores Vermelho (*Red*), Verde (*Green*) Azul (*Blue*) (RGB). Portanto, a imagem é constituída de 248 x 400 x 3 elementos, ou seja, um total de 297600 números. Sendo cada número variando nos intervalos de 0 (preto) à 255 (branco). Logo, nossa tarefa de classificação é transformar essa quantidade maior que um quarto de milhão de números em rotulo simples, como gato[18].

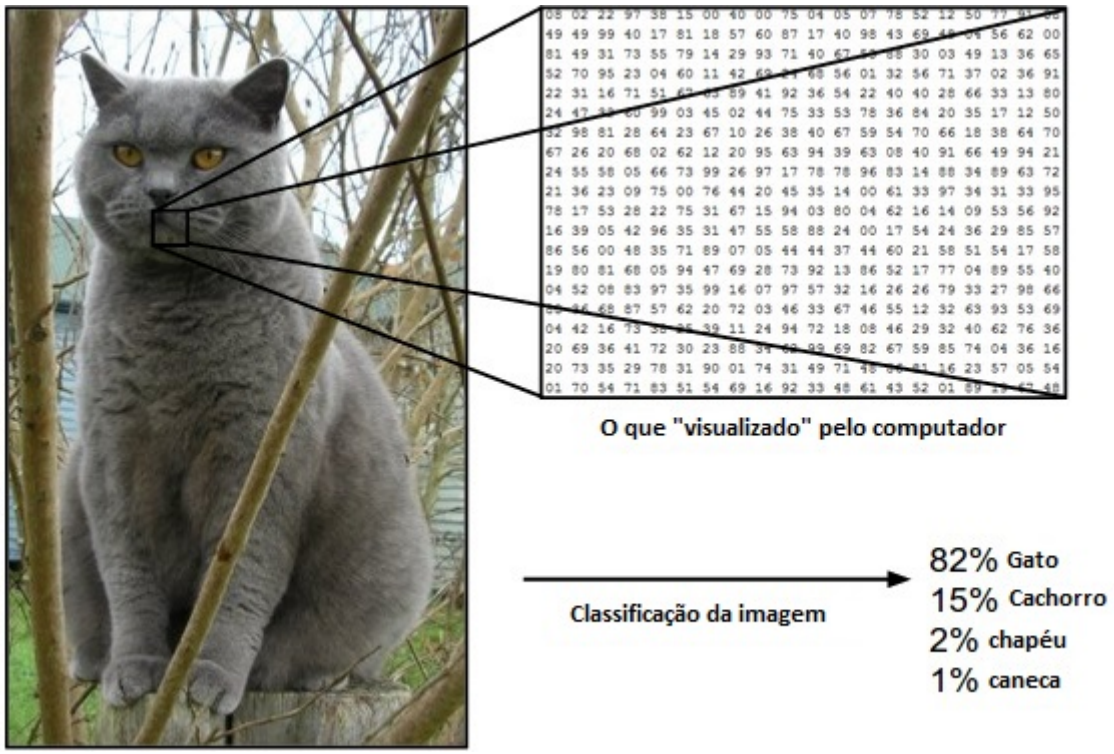

Figura 2.12: Representação computacional da imagem de um gato. Adaptado de: Karpathy [18]

A tarefa de reconhecer um conceito visual (como um gato) é relativamente trivial para um ser humano, entretanto criar um algoritmo para tal envolve inúmeros desafios. Mantendo em mente que uma imagem é um vetor tridimensional, são listados abaixo alguns problemas:

- **Variação de ponto de vista:** Uma única instância de um objeto pode ser orientado em vários jeitos com respeito a câmera.
- **Variação de escala:** Objetos dentro das classes frequentemente exibem variações em seus tamanhos.
- **Deformação:** Muitos objetos não possuem corpos rígidos e podem ser deformados em maneiras extremas.

- **Obstrução de parte da imagem:** Algumas vezes somente uma parte do objeto é visível.
- **Condições de iluminação:** Os efeitos de iluminação mudam drasticamente os pixels.
- **Confusão com fundo:** Os objetos de interesse podem estar camuflados com o ambiente.
- **Variação dentro da classe:** As classes de interesse podem ser vastas, como a classe cadeira. Existem muitos tipos destes objetos, cada um com sua própria aparência.

Um bom modelo de classificação de imagem deve ser robusto a todos essas variações, enquanto simultaneamente ser sensível as variações dentro das classes. A figura 2.13 ilustra os desafios encontrados pelo classificador.

Figura 2.13: Dificuldades da classificação de imagens em visão computacional. Adaptado de: Karpathy [18]

Diferentemente de ordenar os números em uma lista, não é óbvio como podemos escrever um algoritmo para identificar objetos em imagens. Portanto, ao invés de tentar especificar cada uma das características das categorias de interesse diretamente em código, o método utilizado é o *Data-driven*, ou seja, ser guiado pelos dados. Logo, provemos ao computador muitos exemplos de cada classe e então desenvolvemos os algoritmos que analisam e aprendem as características visuais de cada classe, ou seja, funções parametrizadas que aprendem seus parâmetros. A imagem 2.14 mostra um exemplo de um conjunto de imagens (*dataset*) para treinamento desses algoritmos[18].

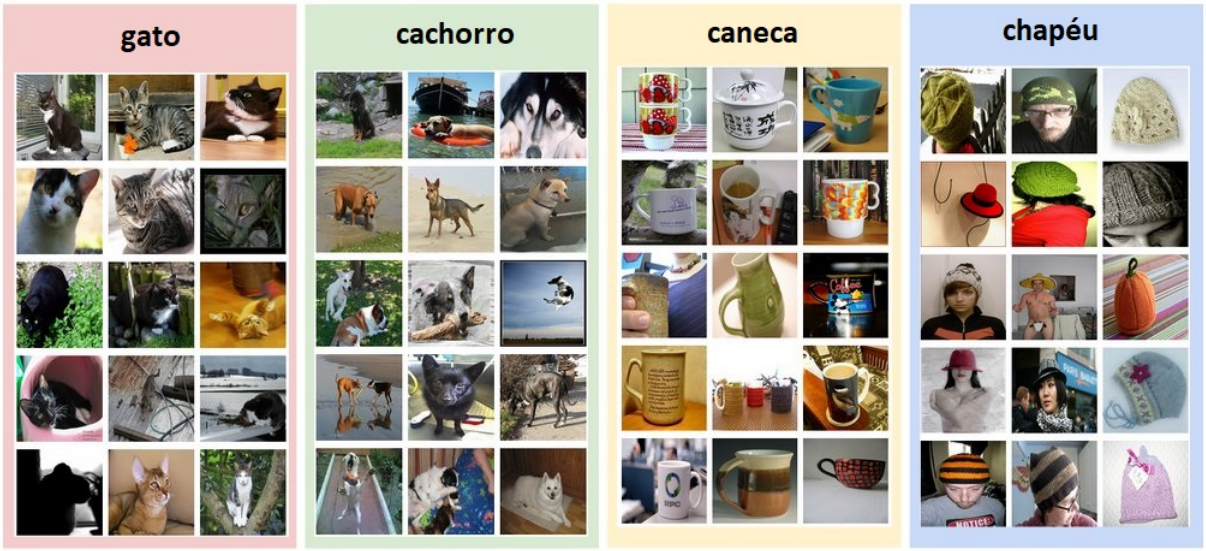

Figura 2.14: Exemplo de um banco de dados para classificação de imagens. Em prática os bancos de dados possuem centenas de milhares de imagens para cada categoria. Adaptado de: Karpathy [18]

2.3.10 Classificador Linear

O primeiro passo para o desenvolvimento de um algoritmo que aprende com os exemplos, é a definição da ***score function*** (função de pontuação) que mapeia entradas, em nosso caso os pixels de uma imagem, para a pontuação de cada classe. Para isso, seja nossas imagens dento do **dataset** definidas como $\boldsymbol{x_i} \in \mathbb{R}^{\boldsymbol{D}}$ associadas a uma *label* (rótulo) $\boldsymbol{y_i}$ com $i = \{1...\mathbf{N}\}$, $\boldsymbol{y_i} \in \{1...\mathbf{K}\}$ sendo $\mathbf{N}$ o número de exemplos, $\mathbf{K}$ o número de classes e $\mathbf{D}$ dimensão da imagem (Altura $\times$ Largura $\times$ Canais de cores), a *score function* será defina como um mapeamento $\boldsymbol{f} : \mathbb{R}^{\boldsymbol{D}} \rightarrow \mathbb{R}^{\boldsymbol{K}}$ que mapeia os $\mathbf{D}$ pixels de entrada para a pontuação das $\mathbf{K}$ classes.

Dado pela equação 2.13 o classificador Linear é considerado o mapeamento mais simples possível.

$$\boldsymbol{f(x_i, W, b) = W x_i + b} \tag{2.13}$$

Na equação 2.13, a imagem $\boldsymbol{x_i}$ tem todos os seus pixels alocados em um vetor de única coluna na forma [D x 1]. A matriz $\boldsymbol{W}$ (de tamanho [K x D]) e o vetor $\boldsymbol{b}$ (de tamanho [K x 1]) são os parâmetros da função. Os parâmetros $\boldsymbol{W}$ são chamados de *weights* (pesos) e $\boldsymbol{b}$ é chamado de vetor de bias (traduzido com preconceito ou viés) devido ao fato que ele influência a pontuação da saída, sem interagir com os dados de entrada. A matriz $\boldsymbol{W}$ (de tamanho [K x D]) e o vetor $\boldsymbol{x_i}$. Os dados de entrada $\boldsymbol{(x_i, y_i)}$ são dados e fixos, entretanto temos controle sobre os parâmetros $\boldsymbol{W, b}$. O objetivo será calcular quais são os valores ideais de tal forma que a saída do classificador seja igual ao valor de $\boldsymbol{y_i}$, ou seja, a

classe correta seja dada pelo classificador. Dependendo dos valores desses pesos, a função de mapeamento será capaz de gostar ou não de certas cores ou regiões da imagem. Por exemplo, podemos esperar que uma classe de um navio vai gostar mais dos pixels azuis que provavelmente correspondem a água.

A figura 2.15 mostra um exemplo de mapeamento de imagens para classes. Na figura, a imagem de um gato é representado por apenas 4 pixels monocromáticos (apenas um canal de cor) para fins de visualização e possuímos apenas 3 classes. Realocamos todos os pixels em uma única coluna para realizar a multiplicação matricial para conseguirmos a pontuação final de cada classe. Neste exemplo os valores de $\boldsymbol{W}$ não são bons: os pesos atribuem a nossa imagem o menor valor para classe gato e o maior valor para a classe cachorro.

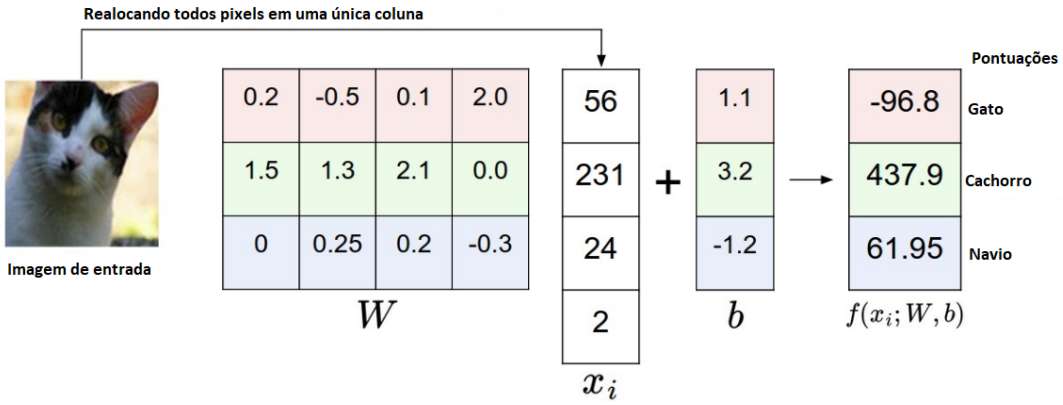

Figura 2.15: Exemplo de um classificador linear de imagens para três categorias distintas, gato, cachorro e navio. Adaptado de: Karpathy [18]

Como as imagens foram espremidas em um vetor coluna de alta dimensão podemos interpretar cada imagem como sendo um ponto nesse espaço de dimensão **D**. Representando o espaço dimensional **D** como um espaço de duas dimensões para fins de visualização, podemos ter uma breve noção do que o classificador linear faz geometricamente. Como dito anteriormente, cada linha de $\boldsymbol{W}$ é um classificador, logo a interpretação geométrica desses números é que ao mudarmos as linhas de $\boldsymbol{W}$, a linha correspondente no espaço **D** irá rodar em diferentes direções. Os bias $\boldsymbol{b}$, diferentemente, permitem a translação das linhas, caso não houvessem os bias, todas linhas passariam e rotacionariam em torno da origem.

A figura 2.16 mostra uma representação do espaço **D** em duas dimensões no qual cada imagem é representada como sendo um ponto e três classificadores são vistos. Usando o exemplo do classificador de carros (em vermelho), a linha vermelha mostra que todos os pontos que obtêm uma pontuação de zero na classe carro. A seta vermelha mostra a direção no qual a pontuação aumenta, logo todos os pontos a direita dessa linha possuem pontuação positiva (que aumenta linearmente), e todos os pontos a esquerda possuem pontuação negativa (decrementando linearmente).

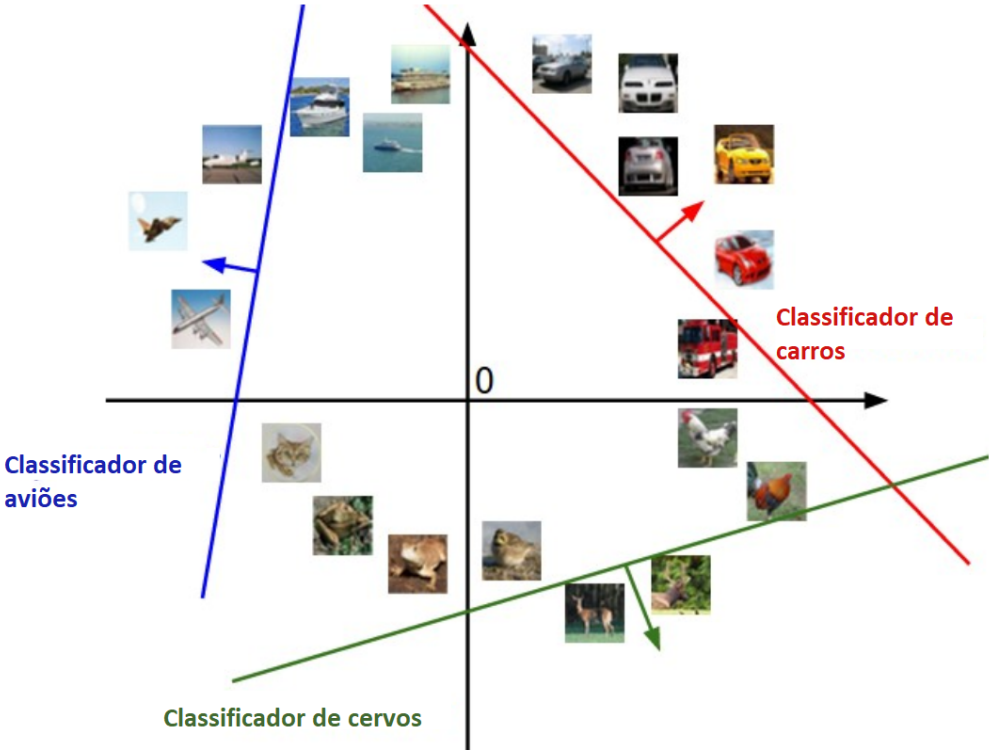

Figura 2.16: Representação de um classificador linear em duas dimensões Adaptado de: Karpathy [18]

2.3.11 Redes Neurais

Como o próprio nome diz, classificadores lineares só conseguem obter separações lineares entre seus dados de entrada, como demonstrado no exemplo ilustrativo da figura 2.16. Ao introduzir camadas intermediarias com não linearidades, as redes neurais, *neural networks* (NN), conseguem superar a limitação dos classificadores lineares e são capazes de aproximar qualquer função, sendo consideradas aproximadores universais de funções. A figura 2.17 mostra a diferença entre as capacidades do classificador linear e da rede neural em classificar dados. O modelo matemático das redes neurais pode ser visto na equação 2.14.

$$f(\boldsymbol{x_i}) = f^{(L)}(...f^{(2)}(f^{(1)}(\boldsymbol{x_i},\boldsymbol{W^1}),\boldsymbol{W^2})...),\boldsymbol{W^L}) \tag{2.14}$$

Na equação 2.14, x_i denota a entrada, e $f^{(L)}$ denota a camada L com pesos $\boldsymbol{W^L}$ da *network* e os bias foram omitidos para simplificar a notação. Logo, a equação pode ser interpretada como uma sequência de funções que transformam vetores mapeando-os de um espaço a outro, até atingir o resultado desejado [19].

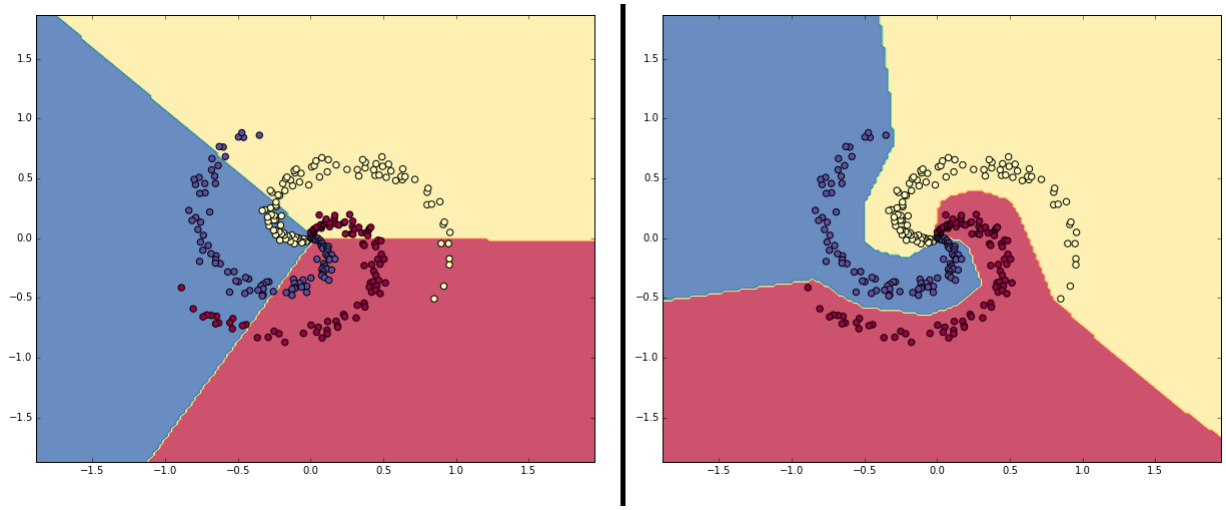

Figura 2.17: Exemplo de um classificador linear (esquerda) vs uma rede neural (direita) para a classificação de pontos 2D. Adaptado de: Karpathy [18]

As redes neurais são formadas por camadas e cada camada é formada por unidades chamadas de **neurônios**. A figura 2.18 mostra o modelo matemático de um neurônio. Essas unidades possuem esse nome devido à inspiração de seu modelo na biologia. Como demonstrado no modelo, o sinal de entrada viaja pelos axônios (x_0) interagindo multiplicativamente (w_0x_0) com os dendritos do outro nerônio baseado na força daquela sinapse (w_0). A ideia é que a força das sinapses (os pesos $\boldsymbol{w}$) são aprendidos e controlam a influência (e a direção: excitatório (pesos positivos) ou inibitórios (pesos negativos)) de um neurônio com o outro. Os dendritos carregam os sinais para o corpo da unidade, onde todos são somados e se elas estiverem acima de um limiar o neurônio é ativado. Essa ativação é controlada por uma função especial não linear chamada **função de ativação**[18].

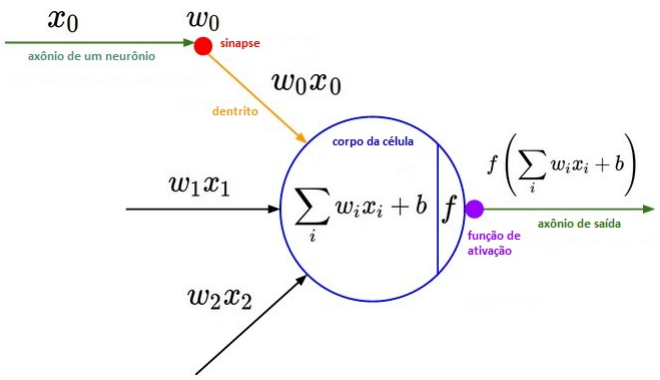

Figura 2.18: Representação de um neurônio em uma rede neural. Adaptado de: Karpathy [18]

Por razões históricas, a função de ativação mais utilizada até pouco tempo era a função de **Sigmoid** ($\boldsymbol{\sigma}$) dada pela equação 2.15, a função de Sigmoid comprime a saída de um neurônio no intervalo de 0 a 1. Outra função de ativação que era comumente utilizada era a

tangente hiperbólica (***Tanh***), cuja a equação é mostrada por 2.16 em termos da Sigmoid, esta função comprimi os valores no intervalo de -1 a 1 e tem a vantagem de ser centrada em zero, o que facilita a aprendizagem da rede neural. A imagem 2.19 mostra a curva de ambas funções de ativação, com a Sigmoid na parte direta e a Tangente Hiperbólica na parte esquerda.

Equação da função de Sigmoid:

$$\boldsymbol{\sigma(x) = \frac{1}{1 + e^{-x}}} \tag{2.15}$$

Equação da Tangente Hiperbólica em termos da função de Sigmoid:

$$\boldsymbol{tanh(x) = 2\sigma(2x) - 1} \tag{2.16}$$

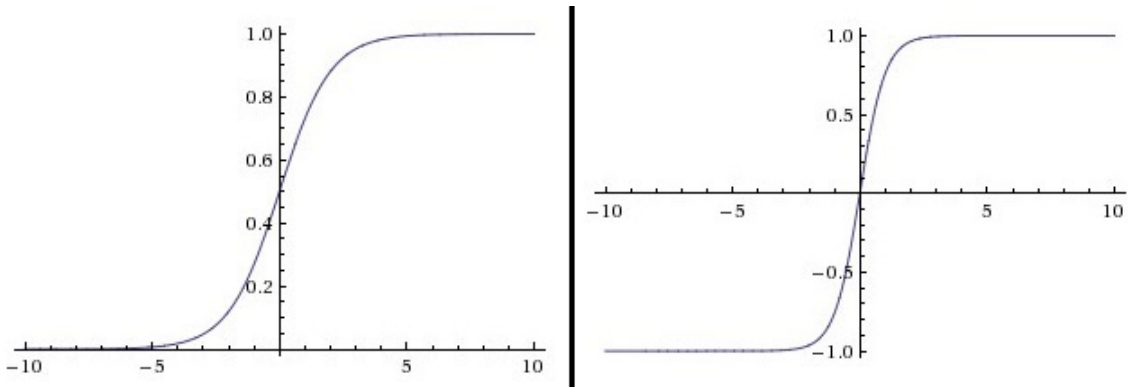

Figura 2.19: Funções de ativação: Sigmoid (esquerda) e Tangente Hiperbólica (direita). Retirado de: Karpathy [18]

Entretanto, as funções $\boldsymbol{\sigma}$ e ***Tanh*** possuem um grande problema: a saturação. Se os pesos inicias forem muito grandes a maioria dos neurônios se tornarão saturados impossibilitando a aprendizagem. Por essa e outras razões, a função de ativação mais utilizada atualmente é a *Rectified Linear Unit* (ReLu). A figura 2.20 mostra a curva da ativação ReLu, devido a sua forma de não saturação linear e sua simplicidade de cálculo (em relação a exponenciais) há um grande aumento na velocidade de aprendizagem das NN. A função matemática da ReLu é dada pela equação 2.17.

Operação matemática realizado pela função Relu:

$$\boldsymbol{f(x) = max(0, x)} \tag{2.17}$$

Como citado anteriormente, as redes neurais possuem uma arquitetura em formato de camadas, ou seja, a saída de cada camada é a entrada da próxima, essa arquitetura pode ser visto na figura 2.21. A primeira camada é chamada de camada visível ou camada de entrada (***input layer***) pois a mesma contém as variáveis que podem ser vistas, caso a entrada seja imagens, as mesmas devem ser reorganizadas como um vetor coluna

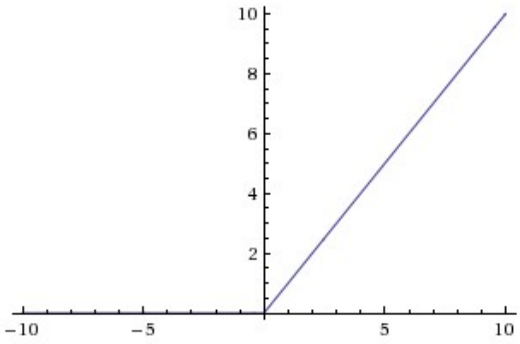

Figura 2.20: Representação da função de ativação Rectified Linear Unit (ReLu). Retirado de: Karpathy [18]

assim como no classificador linear. Em sequência, temos as camadas escondidas (***hidden layers***) que são responsáveis por extrair características abstratas dos dados. Nestas camadas, o modelo deve determinar quais são os conceitos que são úteis para explicar a relação entre os dados observados. E por final, pela interação entre a entrada e as camadas escondidas, temos a saída (***output layer***) que define a pontuação de cada classe[5].

Na figura 2.21 temos duas NN distintas. A da esquerda é uma rede neural com 2 camadas (uma camada escondida com 4 neurônios e a camada de saída com 2 neurônios). Já, a da direita é uma rede neural com 3 camadas (duas camadas escondidas com 4 neurônios cada e a camada de saída com 1 neurônio). Logo, a camada de entrada não é contada para a nomeação do número de camadas de uma NN. Definindo **D** como dimensão dos dados de entrada, $N^{(L)}$ como o número de neurônios em uma *layer* (L), as dimensões das matrizes de pesos serão $\boldsymbol{W^{(L)}}$=[$N^{(L)}$ x $N^{(L-1)}$], sendo $\boldsymbol{W^{(1)}}$=[$N^{(1)}$ x D] e os bias $\boldsymbol{b^{(L)}}$=[$N^{(L)}$ x 1].

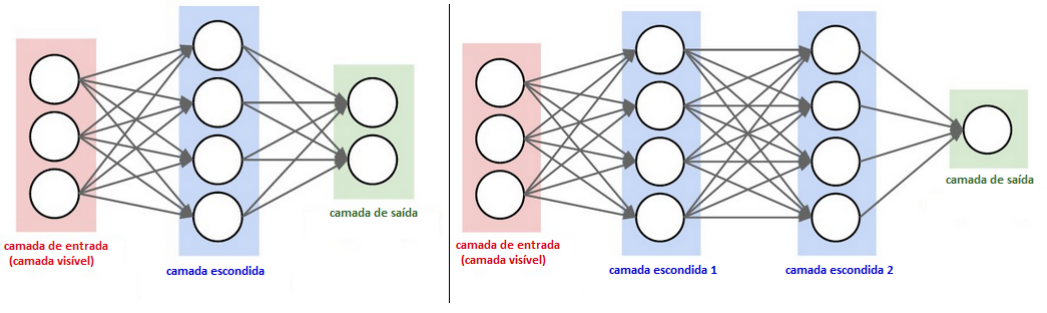

Figura 2.21: Exemplos de arquitetura de uma rede neural Densa. Adaptado de: Karpathy [18]

Este tipo de rede neural apresentado, no qual cada neurônio possui conexões com todos os neurônios da camada anterior é chamada **rede neural densa**, *feedfoward neural network* ou *multi layer perceptron* (MLP). É importante ressaltar que a função de ativação é aplicada elemento a elemento (saída de cada neurônio) e que a camada de saída não

possui uma função de ativação, logo, da última *hidden layer* para a saída o comportamento é linear. Dessa forma, um classificador linear pode ser pensando como um caso especial de uma rede neural de uma camada (sem *hidden layers*)[18]. Com todas essas considerações, um exemplo matemático de uma rede neural com 2 camadas é dado pela equação 2.18.

$$\boldsymbol{f(x) = W^{(2)} max(0,(W^{(1)}x + b^{(1)})) + b^{(2)}} \tag{2.18}$$

O número de neurônios por camadas, como número de camadas, entre outras escolhas possíveis que determinam a arquitetura do modelo são chamados **hiperparâmetros** e afetam a eficacia do mesmo. Logo, a escolha correta dos mesmos define se um modelo é bom ou não, infelizmente não existem fórmulas exatas para a escolha correta a não ser dados empíricos e a realização de testes para cada aplicação[18].

2.3.12 Função de Custo

Ao analisar a saída das equações 2.13 e 2.18 vemos que os valores não possuem nenhum significado diretamente, como mostrado na figura 2.15, logo será utilizado a função ***Sotfmax*** para obtermos as probabilidades de uma imagem ser de cada uma das classes disponíveis. A equação do *Softmax* é dada pela equação 2.19 abaixo.

$$\boldsymbol{P(y_i|x_i; W, b) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}} \tag{2.19}$$

A equação 2.19 pode ser interpretada como a probabilidade normalizada atribuída a *label* correta $\boldsymbol{y_i}$ (podendo ser de qualquer uma das outras classes disponíveis) dada uma imagem $\boldsymbol{x_i}$ e parametrizado por $\boldsymbol{W}$ e $\boldsymbol{b}$. A saída da equação 2.13 pode ser interpretada como sendo as probabilidades logarítmicas não normalizadas para cada classe. Logo, exponenciando essas quantidades obteremos as probabilidades não normalizadas e realizando a divisão pelo somatório obteremos a normalização, gerando assim as probabilidades de cada classe.

O exemplo mostrado na figura 2.15 o classificador tem como entrada uma imagem de um gato, entretanto a pontuação da classe gato foi baixa comparado com as outras classes. Com o objetivo de corrigir as saídas dos nossos classificadores, definiremos um função custo, do inglês *loss function*, que medirá o quão "feliz" estamos com as saídas do algoritmo. Intuitivamente, o custo (**loss**) será grande se o classificador estiver classificando erroneamente os dados ou baixo caso contrário. Apresentando essas características, uma das funções mais utilizadas é conhecida como *cross-entropy loss* e é dada pela equação 2.20[18].

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{2.20}$$

A *cross-entropy loss* ou *log loss* mede a performance dos modelos de classificação nos quais as saídas são as probabilidades entre 0 e 1, ou seja, como demonstrado na equação (2.20) esta função de custo realiza a operação de logaritmo sobre função *Softmax*. Como mostrado na figura 2.22, a *cross-entropy loss* será grande quando o classificador gerar uma probabilidade baixa para a classe correta e pequena caso contrário, sendo zero no caso ideal quando a classe correta possui a probabilidade de 1[20].

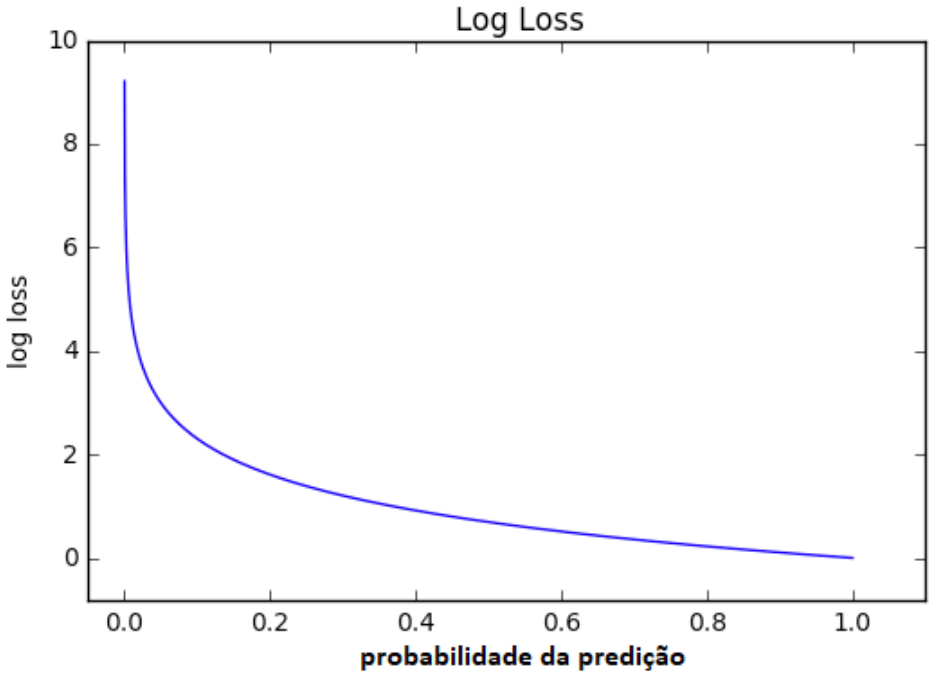

Figura 2.22: Representação da função de custo *cross-entropy loss*. Adaptado de: ML-CheatSheet [20]

A equação 2.21 mostra como utilizar *cross-entropy loss* para tratar um conjunto **N** de dados, essa versão realiza a média dos custos de cada dado do conjunto, assim essa função é chamada *avarage cross-entropy loss*.

$$L = L - \frac{1}{N} log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{2.21}$$

A figura 2.23 apresenta um exemplo ilustrativo de como é feito o cálculo da função de custo para um classificador linear (também pode ser a saída de uma rede neural). Vemos no exemplo, que a saída do classificador linear apresenta números sem significado e ao aplicarmos a função Softmax, temos a probabilidade de cada classe. Após, comparamos a probabilidade atribuída a classe correta (classe azul) e calculamos a *loss function*. Logo, o nosso objetivo será minimizar a função de custo modificando os valores de $\boldsymbol{W}$ e $\boldsymbol{b}$ (nesse exemplo 15 parâmetros) para que a classe correta obtenha uma probabilidade maior (próximo de 1) e as demais classes uma probabilidade menor (próximo de zero).

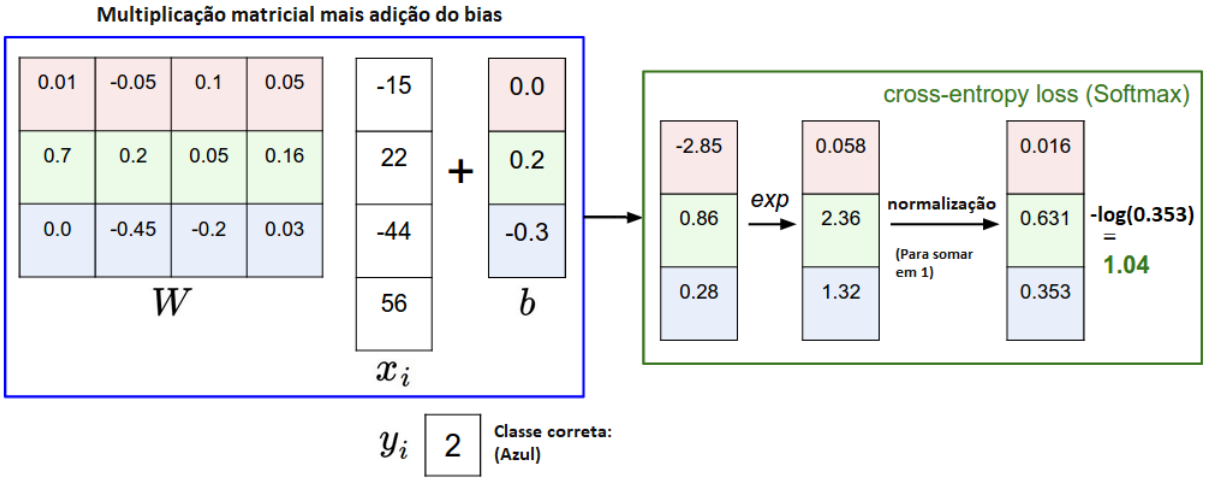

Figura 2.23: Representação de um classificador linear sendo avaliado pela cross-entropy loss. Adaptado de: Karpathy [18]

Normalmente para facilitar o cálculo da *loss function* as *ground truth labels* são colocadas em um formato conhecido como ***one-hot encode***. Neste formato, ao invés de representarmos y_i como inteiros, representamos cada classe como um vetor binário de zeros com dimensão N (número de classes), com apenas 1 na posição correspondente ao valor numérico daquela classe. Por exemplo, considere as *ground truh labels* y_i=[1,2,3], logo no formato *one-hot encode* teremos y_i = [[0,0,1],[0,1,0],[1,0,0]].

2.3.13 Treinamento

Na arquitetura de exemplo mostrada na figura 2.21 a direita, as matrizes de pesos $\boldsymbol{W^{(1)}}$, $\boldsymbol{W^{(2)}}$ e $\boldsymbol{W^{(3)}}$, possuem as respectivas dimensões [4 x 3], [4 x 4] e [1 x 4]. Logo, contando com os bias, temos um total de 41 parâmetros a serem treinados, ou seja, 41 variáveis para serem ajustadas de forma a diminuir a função de custo.

Ao iniciar uma rede neural, os valores iniciais atribuídos aos pesos são números aleatórios próximos de zero (como os pesos interagem de forma multiplicativa, caso seus valores fossem zero todas as saídas para todas entradas seriam zero, impossibilitando o aprendizado) e os bias são iniciados em zero (interagem somando, logo não há problema iniciar com zeros). Logo, é esperado que no começo a NN realize um péssimo trabalho, com transformações aleatórias. Portanto, devemos configurar cada um dos 41 parâmetros, ou seja, definir quais valores das matrizes $\boldsymbol{W}$ de cada camada devem ser aumentados e quais devem ser diminuídos (lembrando que cada camada depende das anteriores) para aumentar a probabilidade da classe correta e diminuir as demais. Em *deep neural networks* do estado da arte para classificação de imagens, temos até 60 milhões de parâmetros a serem treinados. Felizmente essa etapa de treinamento é realizada pelo algoritmo de otimização através do uso do conjunto de dados de entrada. O diagrama mostrado na figura 2.24

mostra como é realizado o processo inteiro de aprendizagem de uma rede neural, esse processo é repetido até que a rede neural esteja treinada (normalmente é repetido dezenas de vezes em milhares de dados)[21].

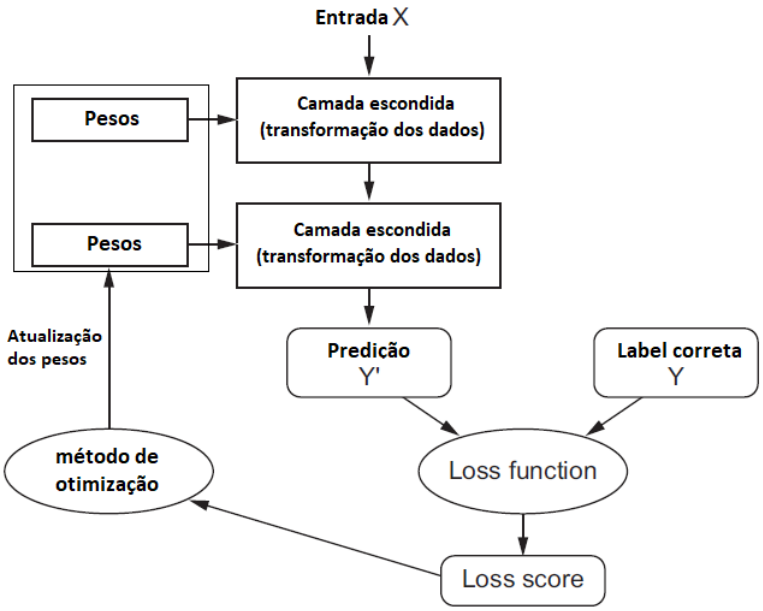

Figura 2.24: Diagrama do processo de aprendizagem de uma rede neural. Adaptado de: Chollet [21]

2.3.13.1 Gradiente Descendente Estocástico (SGD)

A maior diferença entre os modelos lineares e as redes neurais é que devido as não linearidades das NN as funções de custo são não convexas. Isso significa que as redes neurais são treinadas por métodos de otimização de gradiente iterativos, que levam o valor da função de custo a um valor muito baixo (mínimo local), ao invés dos métodos de otimização convexa que levam garantidamente a convergência para mínimo global[5].

Em uma função uni-dimensional, a derivada caracteriza a taxa de mudança instantânea de qualquer ponto dentro da função. O gradiente é a generalização multi variável da derivada, sendo um vetor de inclinações para cada dimensão do espaço de entrada. Como na derivada, o gradiente aponta a melhor direção no qual a função deve caminhar (direção que apresenta a maior taxa de variação) para maximizar um objetivo. Como desejamos minimizar a função de custo das redes neurais seguimos a direção do negativo do gradiente, ou seja, seguimos o gradiente descendente. Cada coeficiente da matriz de pesos em uma rede neural é um dimensão livre no espaço, logo, temos que otimizar dezenas de milhares ou mesmo milhões de dimensões. A figura 2.25 mostra a intuição do gradiente descendente para uma superfície 2D da função de custo (2 parâmetros a serem aprendidos).

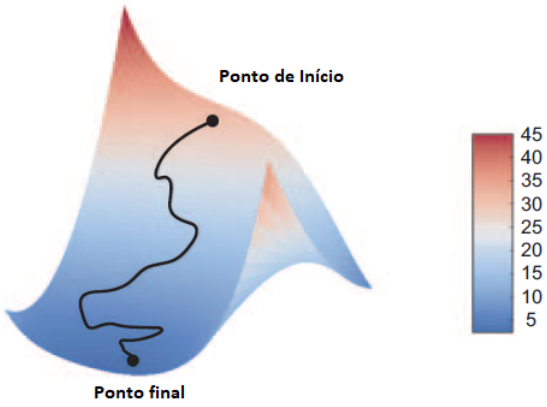

Figura 2.25: Exemplo de gradiente descendente para otimização de uma superfície 2D da função de custo (2 parâmetros a serem aprendidos). Adaptador de: Chollet [21]

Para o treinamento de uma rede neural, os dados de entrada são separados em dois conjuntos, o **conjunto de treinamento** e o **conjunto de teste**, sendo o primeiro muito maior que o segundo (a divisão é normalmente 90% para treinamento e 10% para teste). Logo, o aprendizado da *network* ocorre no conjunto de treinamento, e a avaliação do seu desempenho (capacidade de generalizar características) ocorre nos dados de teste (os dados de teste não são vistos ao longo do treinamento).

Em aplicações em larga escala temos milhares de dados de treinamento, assim percorrer todo o conjunto de treinamento (armazenando o gradiente e os dados de cada exemplo) para calcular a função de custo e atualizar apenas uma vez os parâmetros é caro computacionalmente. Assim, uma abordagem comum é utilizar a técnica ***Stochastic Gradient Descent***, traduzido como gradiente descendente estocástico (SGD) no qual subdividimos o conjunto de treinamento em subconjuntos de dados amostrados aleatoriamente chamados ***batch*** ou ***mini-batch***. Nessa abordagem, computamos e armazenamos os gradientes para cada exemplo dentro do *batch* e atualizamos os pesos quando o último exemplo do subconjunto é avaliado, assim conseguindo uma boa aproximação do gradiente para todo *dataset* (isso ocorre devido a correlação entre dados). O tamanho de um *batch* (*batch size*) normalmente é uma potência de 2 devido a otimização utilizada nas GPUs, comumente variando de 16 a 512. Quando o SGD termina seu ciclo avaliando todos os dados de treinamento através dos *batches*, é completado o que chamamos de um ***epoch***. Por exemplo, se tivermos 1600 imagens divididas igualmente entre duas categorias e desejamos treinar uma NN com SGD com *batch size* de 16 (16 imagens amostradas do conjunto aleatoriamente), serão necessários 100 *batches* (100 **iterações**) diferentes para completar um epoch.

2.3.13.2 Back-propagation

Como as redes neurais são funções compostas, para calcular os gradientes de cada camada é necessário utilizar a regra da cadeia. Para esse cálculo, o algoritmo mais utilizado é o ***Back-propagation*** por realizar as operações em uma ordem específica que é altamente eficiente[5]. A equação 2.22 demonstra a regra da cadeia para o caso escalar, no qual x é um número real, e f e g funções que mapeiam um número real para outro real. Se $y = g(x)$ e $z = f(g(x)) = f(y)$ temos:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.22}$$

Generalizando para o caso multivariável temos as equações 2.23 e 2.24. Suponha $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g uma função que mapeia de $\mathbb{R}^m$ para $\mathbb{R}^n$, e f outra função que mapeia de $\mathbb{R}^n$ para $\mathbb{R}$. Se $y = g(x)$ e $z = f(g(x)) = f(y)$ temos:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{2.23}$$

Em notação vetorial temos:

$$\nabla \boldsymbol{x}^Z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \left(\frac{\partial z}{\partial \boldsymbol{y}}\right) \tag{2.24}$$

no qual $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ é o Jacobiano de g e $\frac{\partial z}{\partial \boldsymbol{y}}$ é o Jacobiano de f. Como $\nabla \boldsymbol{x}^Z$ deve possuir a mesma dimensão de $\boldsymbol{x}$ a ordem da multiplicação matricial mostrada na equação 2.24 pode ser invertida[22].

Para exemplificar como o algoritmo *Back-propagation* atualiza os pesos de uma rede neural, considere o grafo computacional de uma rede neural com 2 camadas mostrado na figura 2.26.

Primeiramente a NN é alimentada com os dados de entrada e realiza classificação, atualizando assim a função de custo C, essa etapa é chamada ***forward pass***. Em sequência, realizamos o cálculo do gradiente em relação a função de custo para os parâmetros de cada camada como demonstrado no conjunto de equações dado por 2.25. E após o cálculo de cada gradiente dos dados de entrada do *mini-batch*, realizamos a atualização dos parâmetros da *network*.

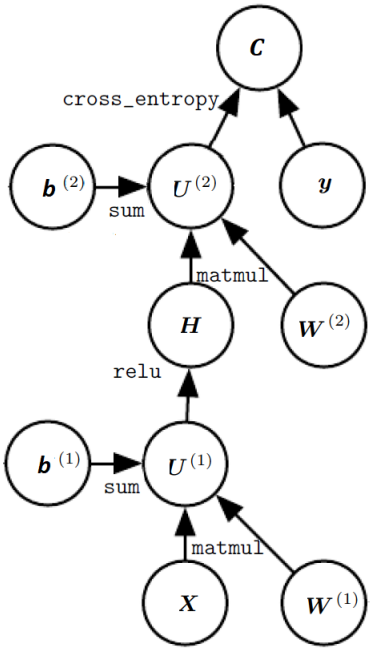

Figura 2.26: Grafo de uma rede neural de 2 camadas. Adaptado de: Goodfellow et al. [5]

$$
\begin{aligned}
\nabla \boldsymbol{U^{(2)}} &= \left(\frac{\partial C}{\partial \boldsymbol{U^{(2)}}}\right) \\
\nabla \boldsymbol{W^{(2)}} &= \left(\frac{\partial \boldsymbol{U^{(2)}}}{\partial \boldsymbol{W^{(2)}}}\right)^{\top} \nabla \boldsymbol{U^{(2)}} \\
\nabla \boldsymbol{b^{(2)}} &= \left(\frac{\partial \boldsymbol{U^{(2)}}}{\partial \boldsymbol{b^{(2)}}}\right)^{\top} \nabla \boldsymbol{U^{(2)}} \\
\nabla \boldsymbol{H} &= \nabla \boldsymbol{U^{(2)}} \left(\frac{\partial \boldsymbol{U^{(2)}}}{\partial \boldsymbol{H}}\right)^{\top} \\
\nabla \boldsymbol{U^{(1)}} &= \left(\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{U^{(1)}}}\right)^{\top} \nabla \boldsymbol{H} \\
\nabla \boldsymbol{W^{(1)}} &= \left(\frac{\partial \boldsymbol{U^{(1)}}}{\partial \boldsymbol{W^{(1)}}}\right)^{\top} \nabla \boldsymbol{H} \\
\nabla \boldsymbol{b^{(1)}} &= \left(\frac{\partial \boldsymbol{U^{(1)}}}{\partial \boldsymbol{b^{(1)}}}\right)^{\top} \nabla \boldsymbol{H}
\end{aligned}
\tag{2.25}
$$

2.3.13.3 Learning Rate

Como dito anteriormente, o gradiente indica a melhor direção para maximização de uma função, entretanto ele não nos diz a quantidade a ser percorrida naquela direção. No caso das *neural networks*, essa distancia é um hiperparâmetro conhecido como ***Learning Rate*** (η) ou *step size*. A equação 2.26 mostra como os pesos de uma NN são atualizados após o calculo de seus gradientes.

$$\boldsymbol{W^{(L)}} = \boldsymbol{W^{(L)}} - \eta \nabla \boldsymbol{W^{(L)}} \tag{2.26}$$

A *learning rate* é um dos hiperparametros mais importantes a serem sintonizados para uma boa performance do modelo. A figura 2.27 demonstra os problemas relacionados aos valores desse parâmetro. Com learning rates pequenas (curva em azul) as melhorias serão próximas a lineares, demorando mais para atingirem um valor aceitável de função de custo. Com learning rates maiores (curva verde) as curvas se parecem mais exponenciais, causando um decaimento maior da função de custo, mas a curva acaba ficando presa em valores piores (ou até mesmo divergindo como demonstrado na curva amarela). Isto acontece devido ao fato que há muita energia na otimização e os parâmetros ficarão oscilando caoticamente, incapazes de se estabelecerem em um bom lugar. Logo, uma boa learning rate é encontrada em um valor intermediário, não tão grande e não tão pequeno como demonstrado na curva vermelha[18].

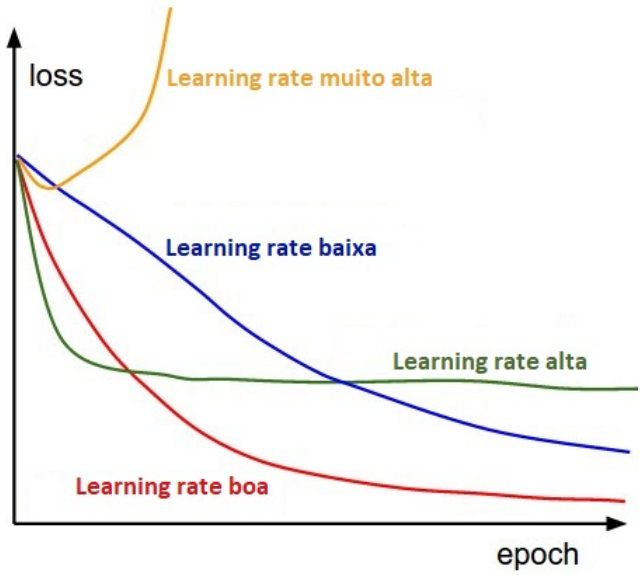

Figura 2.27: Curvas de aprendizagem para diferentes *learning rates*. Adaptado de: Karpathy [18]

2.3.13.4 Variantes do SGD

Outro problema no processo de otimização é que o mesmo pode ficar preso em um mínimo local (no qual em ambas suas extremidades a função de custo aumenta) caso esteja sendo treinado com SGD básico e com uma learning rate muito pequena. A figura 2.28 demonstra um exemplo ilustrativo no qual isso pode acontecer.

Este problema pode ser contornado com as variantes do SGD como o SGD com momentum, Adagrad, RSMProp, Adam, entre outros. Essas variantes são chamadas comumente de otimizadores ou métodos de otimização e levam em conta não apenas o gradiente

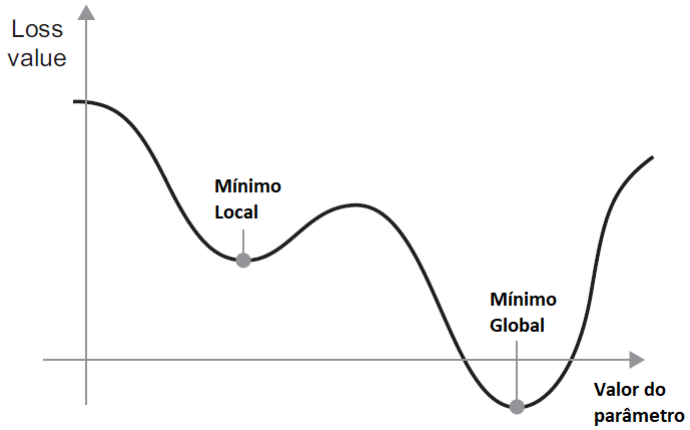

Figura 2.28: Curva com diferentes Mínimos Locais. Adaptado de: Chollet [21]

atual, mas também os anteriores. Muitos desses variantes utilizam um hiperparâmetro chamado **momentum** (inspirado no conceito físico de mesmo nome). Para visualização do funcionamento desse hiperparâmetro, imagine o algoritmo de otimização como sendo um bola rolando pela curva demonstrada pela figura 2.28, caso a bola possua momentum grande o suficiente ela não ficará presa no mínimo local, avançando assim para o mínimo global [21]. Para mais detalhes sobre os métodos de otimização consultar Karpathy [18] na sessão Neural Networks parte 3.

2.3.14 Pré-Processamento dos dados

Como os dados de entrada interagem de forma multiplicativa com os pesos da rede neural em forma de cascata, o pré-processamento dos dados de entrada é uma etapa importante para garantir um melhor condicionamento numérico e uma convergência mais rápida para o algoritmo de treinamento.

A forma mais comum de pré-processamento é a de **subtração da média** dos dados. Nessa abordagem, subtraímos a média de cada **característica** (chamamos de características informações que são importantes aos dados de entrada. Ex: pixels de uma imagem, tamanho de uma pessoa) dentro dos dados de entrada. Dessa forma, centramos a nuvem de dados em torno da origem em cada dimensão eliminando assim a translação dos dados o que ajuda a evitar a explosão ou o desaparecimento dos gradientes[18].

Caso os dados de entrada possuam uma diferença grande de escala entre suas características, o algoritmo de treinamento dará preferência para os de maior escala, o que dificultará a otimização dos demais parâmetros. Para contornar esse problema, devemos **normalizar** as entradas para que todas as características dos dados possuam as mesmas escalas. A normalização de escala acontece em um espaço dimensional muito alto, entretanto com uma visualização em duas dimensões podemos ter uma breve noção do

seus benefícios. A figura 2.29 demonstra os benefícios da normalização das entradas para o algoritmo de aprendizagem. Do lado esquerdo da imagem, vemos que o algoritmo de aprendizagem tem uma maior dificuldade para atingir o objetivo, dando preferência para a característica de maior escala. A distância necessária para o algoritmo atingir seu objetivo dependerá do ponto de início que por sua vez dependerá da inicialização dos parâmetros da NN, logo essa variação na escala dos parâmetros dificulta a otimização. Do lado direito vemos os dados normalizados, o algoritmo atualizará igualmente cada característica da entrada e a forma da superfície ficará mais circular, dessa forma se tornando mais robusta a inicialização dos parâmetros e convergindo mais rápido[23].

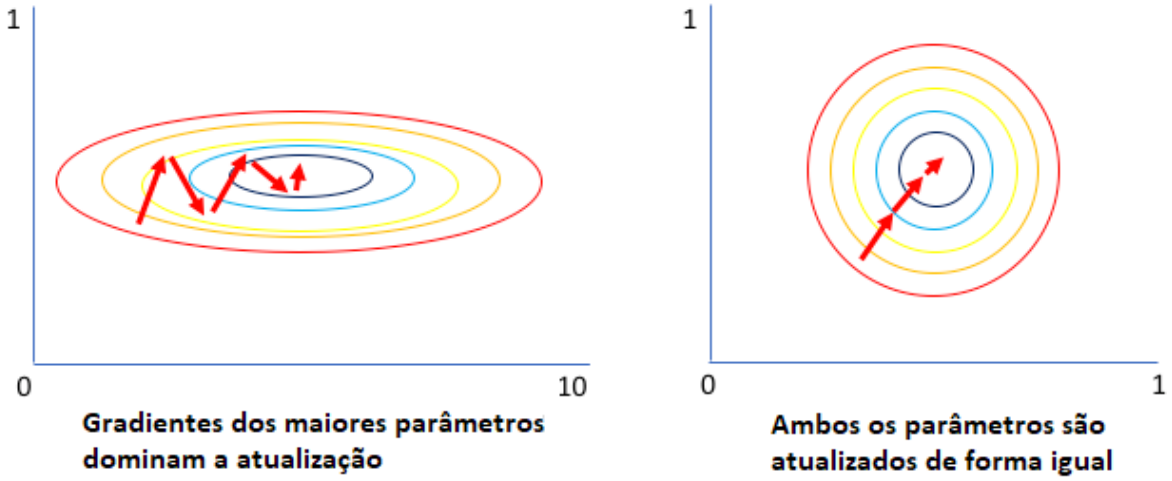

Figura 2.29: Diferença causada por diferentes escalas no SGD. Adaptado de: Jordan [23]

As duas formas mais comuns de normalização são: dividir cada dimensão por seu desvio padrão, uma vez que tenha sido centrada em zero (subtração de média) e a outra é normalizar cada dimensão para que o mínimo e máximo ao longo daquela dimensão seja -1 e 1 respectivamente. Quando se trata de imagens, a segunda é mais utilizada e a formula para realizá-lo em cada canal de cor é dada pela equação 2.27, no qual x representa o valor de cada pixel da imagem[18].

$$x = \frac{2x - 255}{255} \tag{2.27}$$

2.3.15 Métodos de Regularização

Métodos de regularização são métodos que ajudam a evitar o problema de *overfitting*. Este problema ocorre quando uma rede neural aprende apenas a representar o seus dados de teste e não é capaz de generalizar a outros dados não vistos.

2.3.15.1 Distância L2

Talvez o método regularização mais utilizado seja a distância euclidiana, ou distância L2. Este método penaliza o quadrado da magnitude de todos os parâmetros de uma camada de uma rede neural, sendo adicionado diretamente a função de custo. Em outras palavras, para cada peso $\boldsymbol{w}$ em uma camada da rede neural, nos adicionamentos o termo $\frac{1}{2}\lambda \boldsymbol{w}^2$ a função de custo, no qual λ é um hiperparâmetro chamado **força de regularização**[18]. A equação 2.28 demonstra como regularização L2 é somada a função de custo dos dados.

$$L = \underbrace{-\frac{1}{N}\sum_{i} L_i}_{\text{loss dos dados}} + \underbrace{\frac{1}{2}\lambda \sum_{k}\sum_{l} W_{k,l}^2}_{\text{loss da regularização L2}} \tag{2.28}$$

no qual N é número de exemplos, $i = \{1, \cdots, N\}$, L_i é a função de custo de cada exemplo. Já no segundo termo da equação temos k e l sendo iteradores para demonstrar que somamos todos os elementos da matriz de pesos em todas suas dimensões (na equação 2.28 temos uma matriz de 2 dimensões). Realizamos a operação mostrada neste ultimo termo para cada camada com a regularização, somando-os a função de custo total.

A regularização L2 pode ser interpretada intuitivamente como punindo severamente vetores de pesos com apenas alguns valores altos e demais baixos e preferindo vetores com valores mais difusos. Em outras palavras, devido a interação multiplicativa entre os pesos e as entradas, essa regularização tem a propriedade de encorajar a rede neural a usar todos os elementos das entradas ao invés de usar muito poucos elementos[18].

2.3.15.2 Dropout

Dropout é um método de regularização extremamente efetivo e simples que consiste na técnica de desativar neurônios de uma camada após a função de ativação com uma certa probabilidade p (um hiperparâmetro). A figura 2.30 demonstra a ideia por de trás do funcionamento do método de *dropout*.

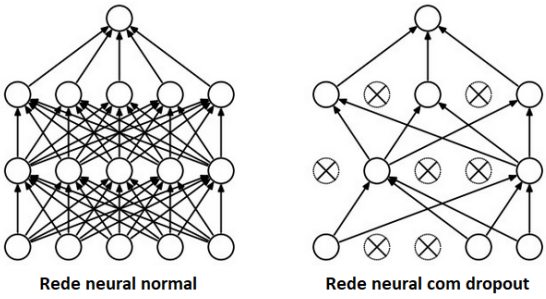

Figura 2.30: Ideia por de trás do funcionamento do método de *dropout*. Adaptado de: Karpathy [18]

Durante o treinamento, a técnica de *dropout* pode ser interpretada como se estivéssemos amostrando sub-arquiteturas de redes neurais dentro da rede neural total, e somente atualizando os parâmetros da rede amostrada. **Durante o teste o *dropout* não é aplicado**, pois durante o mesmo queremos uma predição média de todos os subconjuntos de redes neurais possíveis, ou seja, da rede neural total. Outra interpretação para o papel do *dropout* é que ao desativarmos um neurônio forçamos os neurônios de sua vizinhança a tomar o seu papel na predição, dessa forma generalizamos o papel de cada neurônio[18]. Para mais detalhes sobre o método de *dropout*, junto com uma implementação em linguagem python do mesmo, consultar Karpathy [18] na sessão Neural Networks parte 2.

2.3.15.3 Batch Normalization

O treinamento das redes neurais é complicado devido ao fato que a distribuição da entrada de cada camada muda durante o treinamento, pois os parâmetros da camada anterior mudam (a entrada de uma camada é a saída da camada anterior). Esse fato deixa o treinamento mais lento, requirindo baixas taxas de aprendizado[24]. *Batch normalization*, ou normalização dos *batches*, é uma técnica desenvolvida por Ioffe and Szegedy [24] que torna o treinamento das redes neurais menos problemática ao explicitamente forçar as ativações ao longo da rede a tomar uma distribuição gaussiana desde o começo do treinamento[18]. Para mais detalhes sobre o método de *batch normalization*, os cálculos, conceitos (como *internal covariate shift*), benefícios e implementação prática consultar Nayak [25] além de Ioffe and Szegedy [24].

2.3.16 Exemplos

Para assimilação de todo conteúdo mostrado até esse momento, além das referências já citadas ao decorrer do texto, sobre redes neurais e classificadores lineares são indicados os seguintes links:

- Um classificador linear dinâmico rodando no navegador para classificar pontos (x,y) e diferentes cores. Nesta plataforma podemos mexer em todos dos hiperparâmetros envolvidos no aprendizado de um classificador linear e visualizar seus efeitos na classificação dos dados. Além disso, podemos mudar os pesos, os dados de entrada e visualizar algoritmo aprendendo, através da otimização de gradiente descendente (também visível), a melhor separação linear para os 2 grupos de dados.

  http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

- Uma implementação em linguagem python (sem bibliotecas de *deep learning*) de um classificador linear (*Softmax*) e de uma rede neural para a classificação de pontos (x,y) distribuídos de forma espiral (Figura 2.17). Esta implementação em código

mostra passo a passo as multiplicações matriciais envolvidas, funcionamento do SGD, implementação do método de regularização L2 nesses algoritmos de aprendizagem.

http://cs231n.github.io/neural-networks-case-study/

2.3.17 Rede Neural de Convolução

Devido ao fato que todos os neurônios de uma rede neural densa devem se conectar com todos os neurônios da camada anterior criam uma desvantagem ao se trabalhar com imagens. Por exemplo uma imagem de 200x200x3 gera 120000 pesos ao se conectar a um neurônio, como uma camada é feita de vários neurônios, e uma rede neural feita de várias camadas o número de parâmetros torna-se impraticável. Além do mais, redes neurais densas aprendem padrões globais em seu espaço de características de entrada necessitando reaprender padrões casos os mesmos apareçam e outras localizações. As redes neurais de convolução, do inglês ***Convolutional Neural Networks*** (CNN) tomam vantagem do fato que suas entradas serão somente imagens e criam uma arquitetura mais eficiente. Os padrões que uma CNN aprende são invariantes a translação, logo ao aprender um padrão em uma parte da imagem, ela pode reconhecê-lo em qualquer outra posição. Além disso elas aprendem hierarquias de padrões espaciais, a primeira camada de convolução aprenderá pequenos padrões locais como bordas e borrões, a segunda camada de convolução aprenderá padrões mais amplos feitos das características da primeira camada, como contornos, e assim por diante[18][21]. Essas duas características citadas estão presentes em nosso mundo visual como demonstrado no exemplo ilustrativo dado pela imagem 2.31, o que torna a arquitetura das CNNs muito poderosa e eficiente.

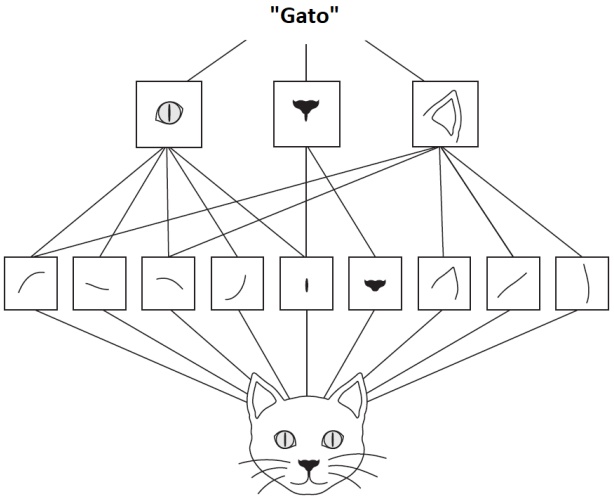

Figura 2.31: Invariança a translação de características e hierárquia de conceitos em uma imagem. Adaptado de: Chollet [21]

As CNN funcionam de um jeito similar as *neural networks* densas apresentadas anteriormente: sendo feitas de neurônios que aprendem pesos e bias e que passam por uma função de ativação, a função ainda é expressa como uma única função de score diferenciável, elas possuem uma função de custo em sua última camada (densa) e todas as dicas e o treinamento ocorrem de maneira similar. Entretanto, como dito anteriormente a entrada das redes neurais de convolução são somente imagens. A arquitetura de uma rede neural de convolução constitui-se principalmente de 3 tipos de camadas: Camada de Convolução, Camada de Pooling e Camada totalmente conectada ou densa (exatamente como visto nas redes neurais densas)[18]. A imagem 2.32 demonstra os tipos de camada interagindo entre si em uma CNN para a classificação de objetos.

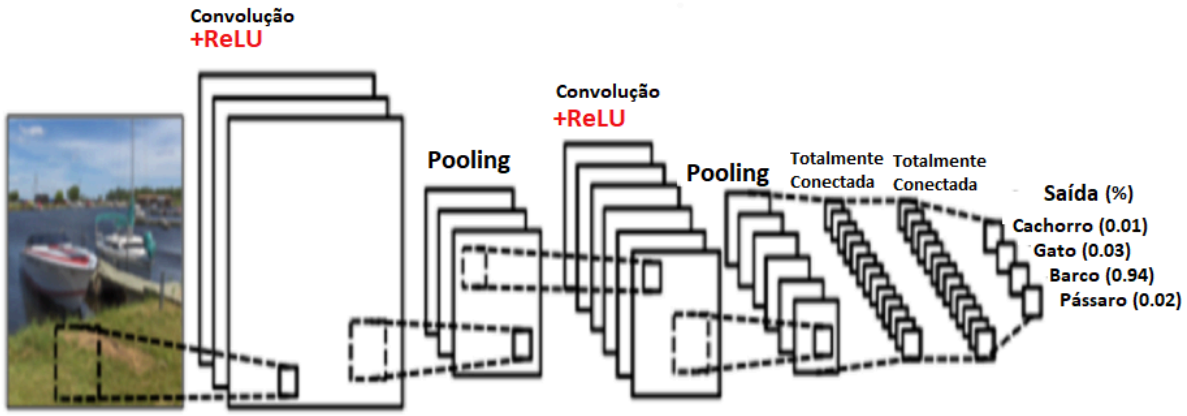

Figura 2.32: Exemplo da arquitetura de uma CNN. Adaptado de: Ujjwalkarn [26]

2.3.17.1 Operação de Convolução

Podemos definir as redes neurais de convolução como simples redes neurais que usam a operação de convolução no lugar da multiplicação geral de matrizes em ao menos uma das camadas. A forma mais geral a operação de convolução é uma operação entre duas funções de valores reais, como mostrada pela equação 2.29.

$$s(t) = \int x(a)w(t-a)da \tag{2.29}$$

A operação de convolução é tipicamente denotada com um asterisco, como mostrado na equação 2.30.

$$s(t) = (x * w)(t) \tag{2.30}$$

Na terminologia das redes neurais de convolução, o primeiro argumento (x) na convolução é referido como entrada (**input**) e o segundo argumento (w) é referido como **kernel** ou **filtro**. Como imagens são sinais discretos no tempo, a operação de convolução

é realizada pela equação 2.31, chamada de ***cross-correlation***. Como essa equação é a utilizada nas CNN, adotaremos como na literatura, seguindo a convenção de chamar essa equação apenas de convolução[5].

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.31}$$

A convolução discreta pode ser vista com um 'produto escalar' entre matrizes (entrada e kernel), no qual o kernel desliza em cima da entrada. Ou seja, realizamos a multiplicação da imagem pelo kernel em uma posição (Ex: i e j igual a zero na equação 2.31) e conseguimos um valor de saída, após deslizamos o kernel por uma distância em pixels (incrementamos i ou j), por exemplo 1 pixel a direita ou abaixo, e realizamos a mesma operação. Começando no canto esquerdo superior da imagem, realizamos essa operação até chegarmos ao canto direito inferior da imagem. Dessa forma, para cada posição do kernel, conseguimos uma saída, no qual consiste um pixel de uma nova imagem. A saída final da operação de convolução entre a entrada e o kernel é chamada na terminologia das CNN de ***feature map***, ou mapa de características. A imagem 2.33 demonstra como ocorre a convolução em imagens.

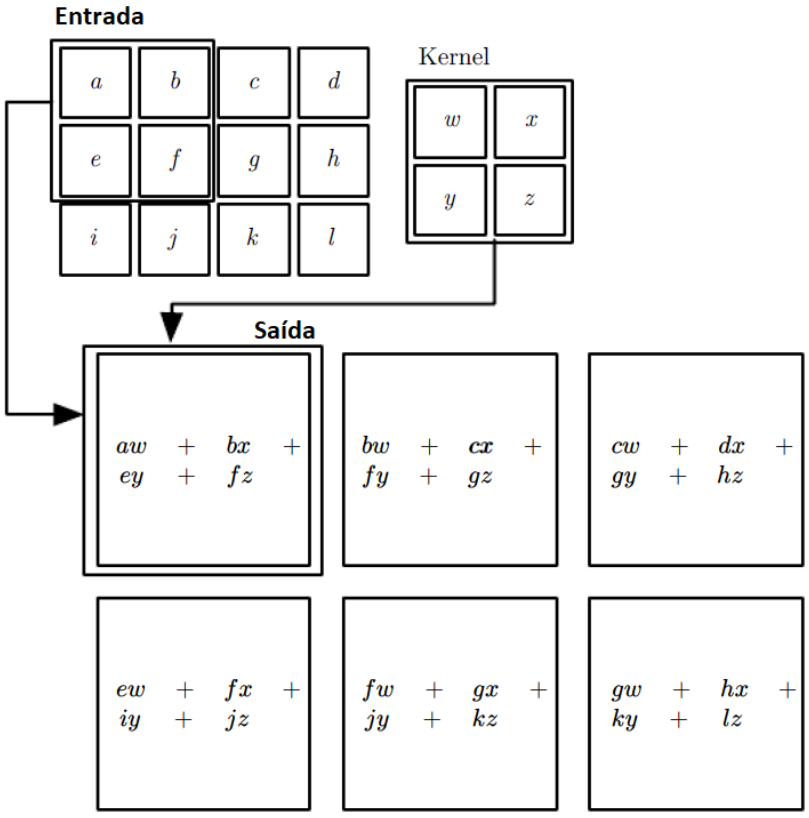

Figura 2.33: Exemplificação da multiplicação de uma kernel por uma imagem. Adaptado de: Goodfellow et al. [5]

Como mostrado na figura 2.33, a imagem de saída possui uma dimensão de saída menor (no exemplo 2 x 3) que a dimensão da imagem de entrada (3 x 4). Esse problema pode ser resolvido com a adição de zeros nas bordas da imagem de entrada, esse método é chamado de ***zero padding***. A figura 2.34 demonstra como ocorre a convolução em uma imagem com *zero padding*.

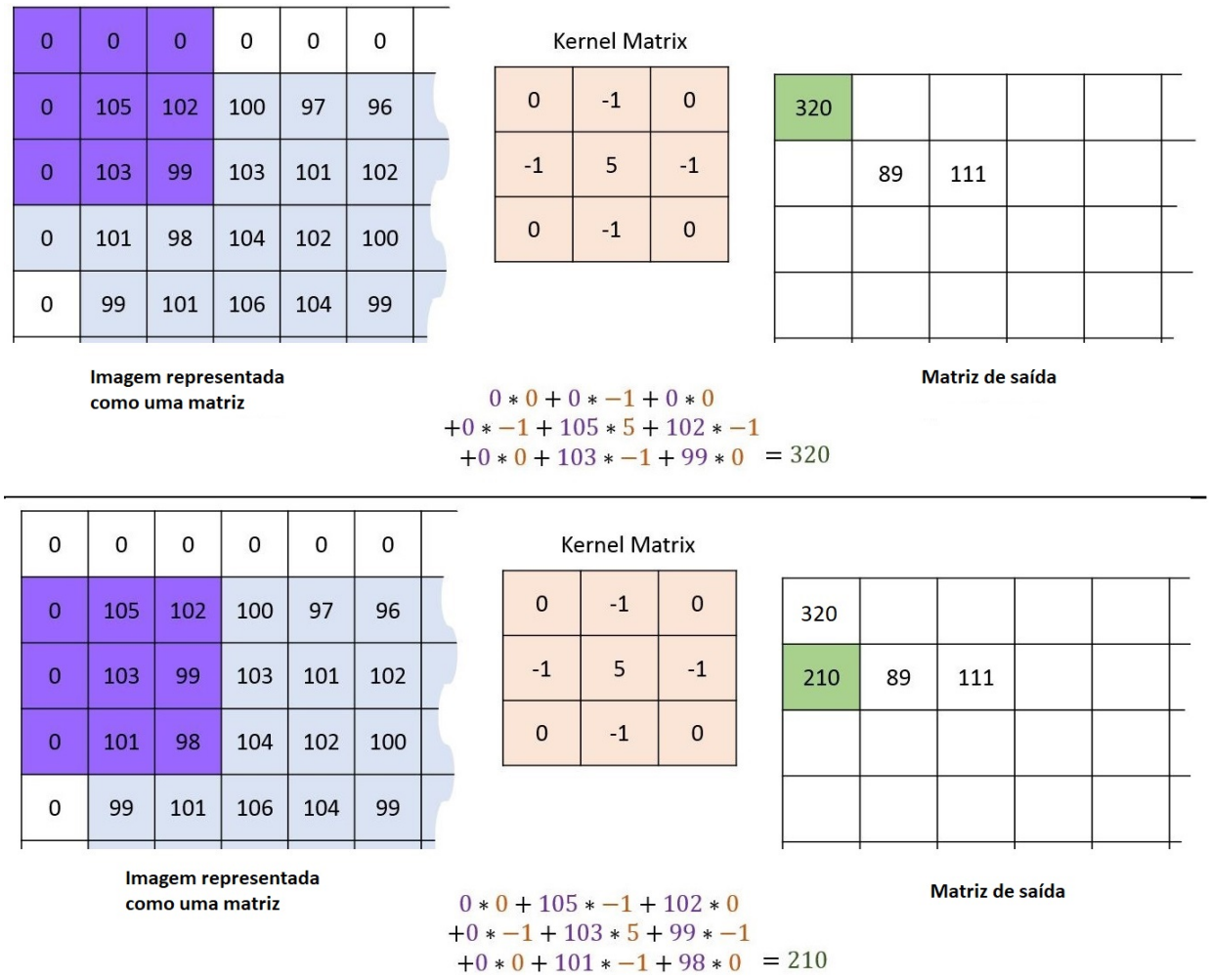

Figura 2.34: Convolução entre um kernel e uma imagem com zero padding. Adaptado de: Machinelearninguru.com [27]

Como citado anteriormente os kernels também são chamados de filtros, pois ao fazer a convolução com a imagem de entrada podemos filtrar ou destacar características úteis da imagem, como por exemplo a detecção de bordas, borrar a imagem para eliminar ruídos, ou evidenciar uma característica. A imagem 2.35 demonstra algumas operações utilizadas em visão computacional utilizando a convolução com kernels.

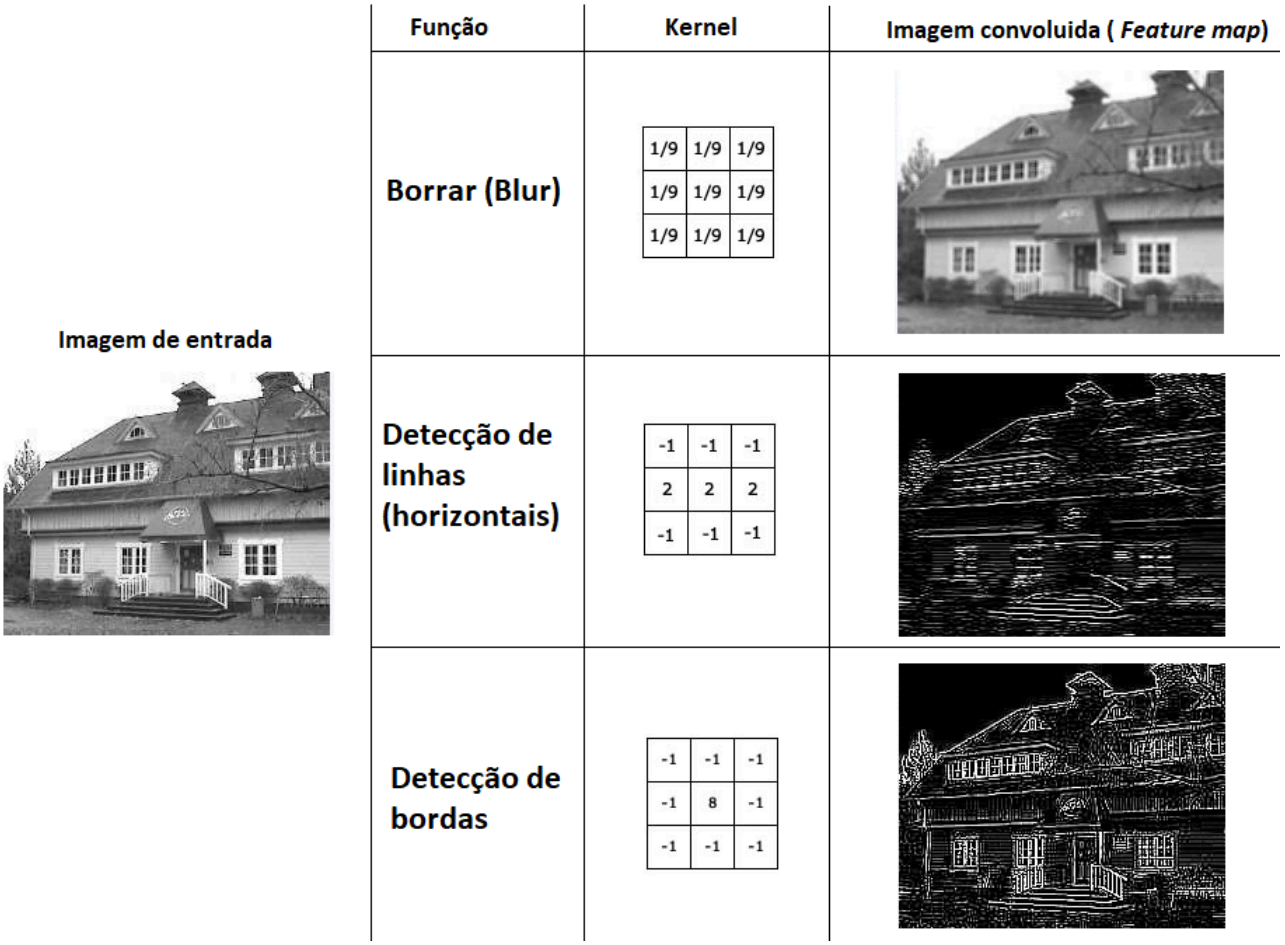

Figura 2.35: Exemplo de filtros (kernels) aplicado em imagens. Adaptado de: Sinha [28]

2.3.17.2 Camada de convolução

Diferentemente das redes neurais densas que possuem suas camadas formadas por neurônios alinhados em uma linha vertical, as CNNs possuem suas camadas, também chamadas de volumes de ativação, arranjadas em volumes tridimensionais formados por largura, altura e profundidade (profundidade aqui condiz com a terceira dimensão da camada de convolução, e não a profundidade da *network*). Além disso, como dito anteriormente, é impraticável conectar todos os neurônios de uma camada a todos pixels de uma imagem, logo os neurônios de uma camada de convolução são conectados apenas a uma pequena região da imagem. A extensão espacial dessa pequena região é um hiperparâmetro chamado ***receptive field*** e pode ser pensado como sendo o tamanho do kernel da operação de convolução. A figura 2.36 mostra a estrutura de uma camada convolutiva. Nesse exemplo, vemos que a imagem de entrada é representada como um volume com dimensões de [32x32x3] e vemos também *receptive field* de um neurônio sobre a imagem de entrada.

O paralelepípedo azul cheio de neurônios dentro do cubo azul maior é o que chamamos

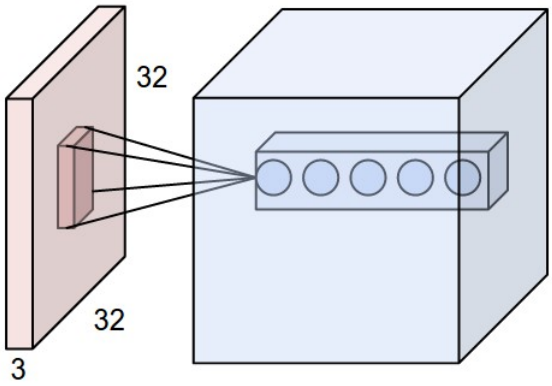

Figura 2.36: Exemplo de uma camada de convolução. Retirado de: Karpathy [18]

de ***depth column***. Todos os neurônios em uma *depth column* possuem a mesma posição em relação as dimensões de altura e largura, variando apenas sua profundidade. Todos os neurônios em uma *depth columnn* possuem o mesmo *receptive field* do volume de entrada variando apenas seus parâmetros. A figura 2.37 mostra outro conceito importante nas CNNs. Na imagem podemos ver as fatias do nosso volume de ativação, chamadas ***depth slice***, ou seja, um recorde bidimensional da camada de convolução.

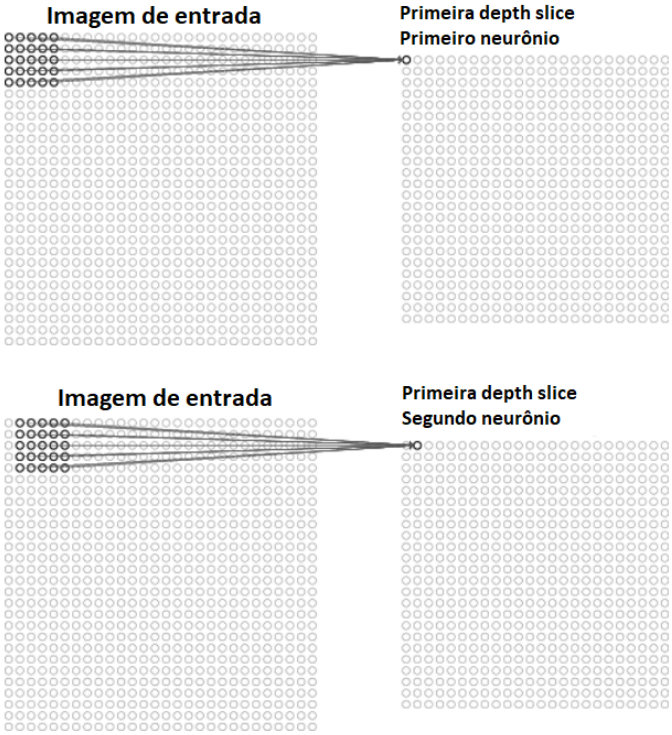

Figura 2.37: Exemplo de uma camada de convolução. Adaptado de: Nielsen [17]

Em uma CNN os parâmetros de todos os neurônios em uma *depth slice* são iguais, ou seja, ao invés de cada neurônio com seu *receptive field* aprender um kernel, todos aprendem o mesmos parâmetros. Dessa forma, diminuímos tremendamente o número

de parâmetros de uma rede neural. Assim, cada depth slice aprenderá um único filtro, sendo cada neurônio da fatia responsável pelo resultado da operação de convolução do kernel sobre uma posição determinada da entrada. A figura 2.37 demonstra como ocorre a convolução entre a imagem e uma *depth slice* com cada neurônio sendo o responsável por armazenar uma posição distinta do filtro sobre a imagem. Embora a *depth slice* seja um fatia do volume de ativação com profundidade 1, ela deve atuar sobre toda a profundidade do volume de entrada, por exemplo, um kernel para uma imagem com 3 canais de cores, possuirá uma profundidade (do kernel) de 3. Assim, a profundidade de uma *depth slice* será sempre igual a profundidade do volume de entrada e profundidade do volume de ativação será igual ao número de filtros da camada. Para visualizar melhor o que foi dito nesse parágrafo, ver a animação em Karpathy [18] na sessão Convolution Neural Networks: Architectures, Convolution/ Pooling Layers.

A distância em pixels que o kernel se deslocará, ou seja, a distância de um *receptive field* de um neurônio para o próximo é um hiperparâmetro conhecido como ***stride***. Em adição, como já mencionado anteriormente, o resultado de uma convolução gera uma imagem com dimensões menores que a dimensão do volume de entrada e para resolver esse problema devemos adicionar os *zero-padding*. A quantidade zero-padding é outro hiperparâmetro na camada de convolução. A figura 2.38 mostra dois exemplos com hiperparâmetros distintos. O lado esquerdo mostra 5 neurônios, cada um com um *receptive field* de 3, um *stride* de 1, e imagem de entrada possui *zero-padding* de 1 (1 em cada borda). Já o lado direito apresenta 3 neurônios com os mesmos *receptive field* e *zero-padding* do lado esquerdo, entretanto, possui um *stride* de 2 (note como o receptive field andou 2 pixels).

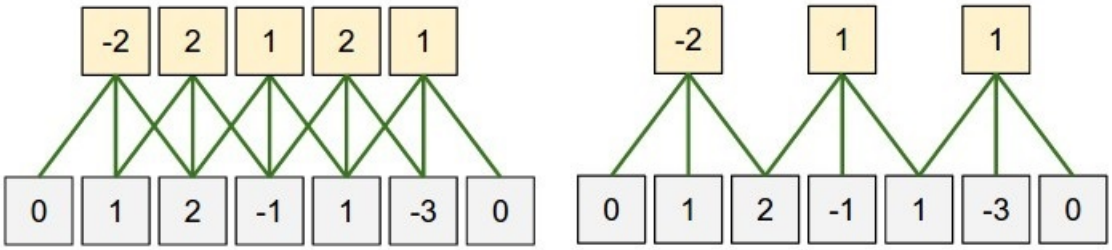

Figura 2.38: Exemplo do uso dos hiperparâmetros de uma CNN. Adaptado de: Karpathy [18]

Como mencionado anteriormente, a profundidade de um volume de ativação é dado pelo número de filtros desejados para aquela camada, entretanto nada foi dito sobre como obtemos sua largura e altura. Como visto na figura 2.38 mudando-se o número do *stride* alteramos o número de neurônios. De fato, o número de neurônios e, consequentemente, o tamanho da altura e da largura da camada de convolução são dependentes de seus hiperparâmetros. A equação 2.32 demonstra como é calculado as dimensões de largura e altura de um volume de ativação.

$$W_2 = \frac{(W_1 - F + 2P + S)}{S} \tag{2.32}$$

Na equação 2.32, W_2 é a dimensão de interesse do volume de ativação (largura ou altura), W_1 é a dimensão de interesse do volume de entrada, F é o *receptive field*, P é quantidade de *zero-padding* e S é o *stride*. Utilizando o exemplo mostrado a direita na figura 2.38 temos $W_1 = 5$, $F = 3$, $P = 1$ e $S = 2$, logo temos uma largura W_2 com 3 neurônios.

Logo, a função de camada de convolução é aprender os filtros que são importantes para identificar seu objetivo. A figura 2.39 mostra um exemplo de um filtro aprendido de [3x3] para identificar dígitos escritos a mão. A direita da imagem, vemos o *feature map* ou *activation map* do filtro, quanto mais claro, maior será o resultado gerado pela convolução. Ou seja, quanto mais claro uma região no *feature map* maior foi a presença do filtro naquela posição.

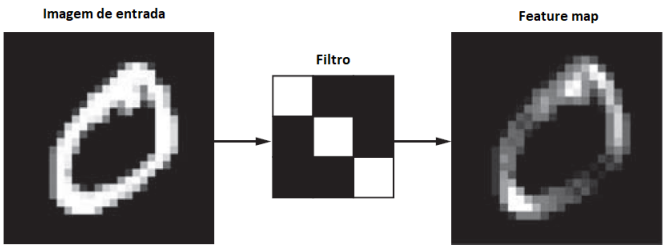

Figura 2.39: Exemplo de um *Feature map* gerado por uma convolução. Adaptado de: Chollet [21]

A figura 2.40 mostra os 96 filtros aprendidos pela primeira camada de convolução da CNN Alexnet desenvolvida por Krizhevsky et al. [29] para competição ImageNet. Como demonstrado, a primeira camada é responsável por identificar conceitos mais genéricos como bordas e borrões.

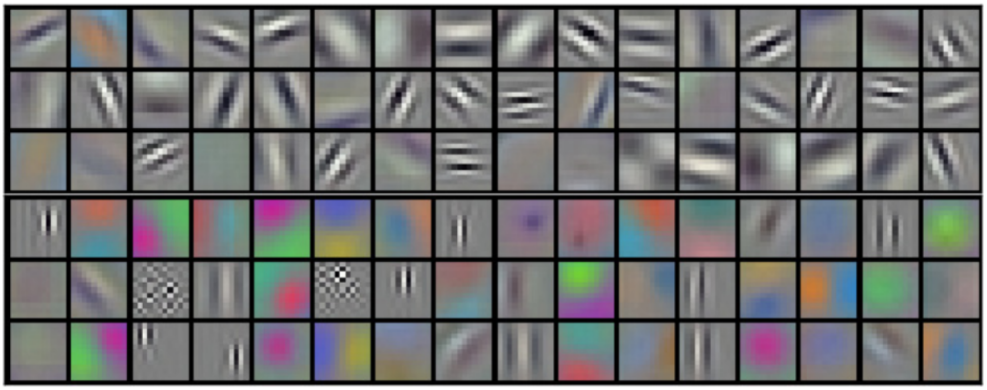

Figura 2.40: Exemplo de filtros aprendidos pela primeira camada de convolução da AlexNet. Retirado de: Krizhevsky et al. [29]

Cada um dos filtros mostrados na imagem 2.40 possui tamanho de [11x11x3], ou seja a primeira camada possui F=11 e avalia um volume de entrada com 3 canais (RGB). Cada *depth slice* dessa arquitetura possui 55x55 neurônios. Note que compartilhamento de parâmetros feito pela *depth slice* é perfeitamente razoável: se a detecção de uma borda vertical é importante em alguma localização da imagem, ela deve ser intuitivamente útil em outra localização devido a estrutura invariante a translações de imagens. Logo não é necessário reaprender a detectar uma borda vertical em cada um das 55x55 localizações distintas no volume de entrada [18].

Assim, resumindo o que foi apresentado nessa sessão, a arquitetura de uma camada de convolução:

- Aceita um volume de tamanho $W_1 \times H_1 \times D_1$
- Requer quatro hiperparâmetros:
  - Número de filtros: K,
  - Sua extensão espacial conhecida como *receptive field*: F,
  - A distância de deslocamento de F, o stride: S,
  - quantidade de zero-padding: P,
- Produz um volume de tamanho $W_2 \times H_2 \times D_2$
  - No qual W_2 e H_2 são dados pela equação 2.32
  - $D_2 = K$
- Com os parâmetros compartilhados entre uma *depth slice*, introduzimos $(F \times F \times D_1)$ pesos por filtro, com um total de $(F \times F \times D_1) \times K$ pesos e K bias.
- No volume de saída, a d-ézima *depth slice* (de tamanho $W_2 \times H_2$) é o resultado da convolução do d-ézimo filtro sobre o volme de entrada com um *stride* de S, e com um offset dado pelo d-ézimo bias.

2.3.17.3 Camada de Pooling

A função de uma camada de pooling, também conhecida como **max pooling**, é diminuir o tamanho espacial das representações para reduzir o número de parâmetros e o cálculo computacional dentro da *network*. Esse tipo de camada, normalmente é inserida entre sucessivas camadas de convolução na arquitetura da CNN e não adiciona nenhum parâmetro a mais como calcula a função fixa da entrada.

A camada de pooling opera independentemente em cada *depth slice* da entrada e a redimensiona espacialmente usando a operação de $max(x,y)$. A forma mais comum de

uma camada de pooling é com um filtro de tamanho F = 2 (2x2) aplicado com um *stride* S=2, diminuindo assim a dimensão de cada *depth slice* da entrada por 2 na largura e na altura, descartando assim 75% das ativações. A dimensão de profundidade se mantém intocável. A figura 2.41 demonstra como a camada de pooling funciona, cada função max toma 4 números mantendo o apenas o maior.

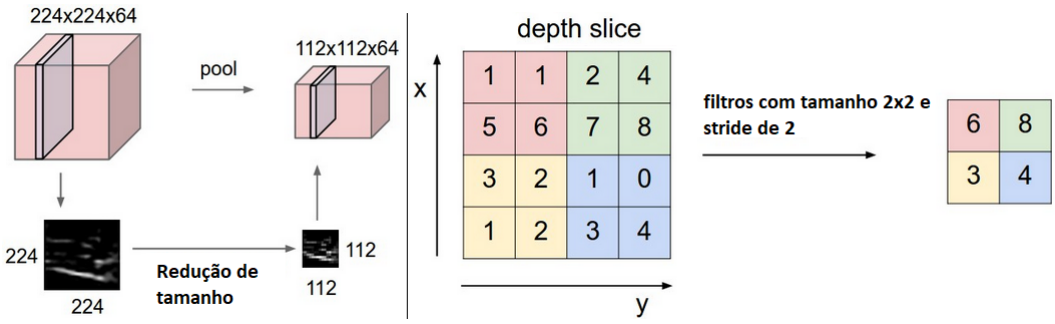

Figura 2.41: Funcionamento de camada de pooling. Adaptado de: Karpathy [18]

2.3.18 Deep Q-Network

Como já dito anteriormente, caso a MDP possua um número muito grande de estados é inviável utilizar as abordagens clássicas RL, como o Q-learning aplicado no exemplo acima como uma tabela. Considere a imagem de um jogo de atari mostrada na figura 2.42, a imagem será redimensionada para um tamanho de 84x84, colocada em escala cinzar (256 níveis de cores - 0:255). Se considerarmos cada pixel como sendo um estado da nossa MDP, teríamos $256^{84x84} = 2^{56448}$ estados possíveis para um simples jogo de atari.

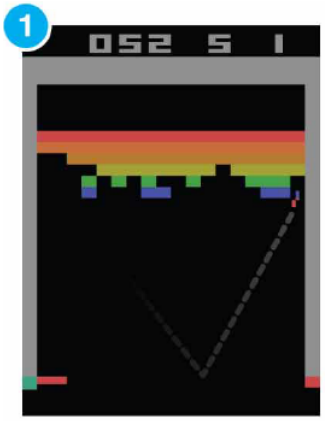

Figura 2.42: Imagem de um jogo de atari. Adaptado de: Mnih et al. [1]

Para esses casos, devemos utilizar uma função parametrizada (Softmax, Neural Networks) para aproximar o valor da *Q-function*, obtendo assim uma generalização para os estados na hora de tomar uma decisão. A figura 2.43 mostra a diferença no cálculo da *Q-function*

do *Q-learning* clássico (*Q-table*) para o uso de uma NN, no caso da *Deep Q Networks* (mais detalhes abaixo) entraremos apenas com o estado e em um *forward pass* teremos os valores de q para todas ações naquele estado.

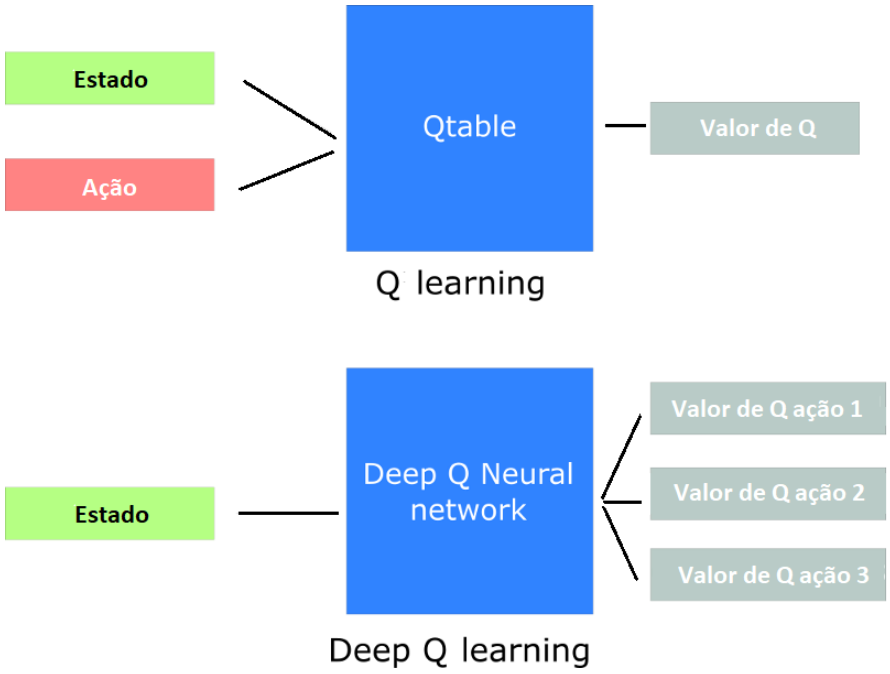

Figura 2.43: Q-table vs Q-network. Adaptado de: Simonini [15]

Neste tópico adotaremos as notações utilizadas na literatura de RL sobre DQN, logo, as funções parametrizadas, terão seus pesos $\boldsymbol{W}$ e bias $\boldsymbol{b}$ representado pela letra grega θ. Neste tipo de abordagem, como já dito, não possuímos mais uma representação da *Q-function* como uma tabela, teremos uma função para aproximar esses valores. Logo, atualizaremos os parâmetros θ da função parametrizada ao contrário dos valores individuais de $Q(s,a)$ como no algoritmo *Q-learning* clássico[10]. Assim sendo, nosso objetivo a ser otimizado será dado pela *loss function* mostrada na equação 2.33 e os pesos e bias serão atualizados como demonstrado na equação 2.34.

$$L(s,a|\theta_i) = (R + \gamma \max_{a'} Q(s',a';\theta_i) - Q(s,a;\theta_i))^2 \tag{2.33}$$

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta L(\theta_i) \tag{2.34}$$

no qual i (de iterações) representa o time step t das notações anteriores. Vale notar que a equação 2.34 é a mesma equação 2.26 apresentada na sessão 2.3.13.3 com uma notação diferente, agora sim, a variável α será a mesma *learning rate* das redes neurais.

Como já apresentado nas sessões anteriores, as redes neurais possuem diversas vantagens em relação ao seu poder representacional em relação ao mapeamento linear, entretanto ao aproximar os valores da *Q-function* por uma função não linear, o processo de aprendizado fica instável (oscilando ou mesmo divergindo)[1]. Isso é devido ao fato que a

mesma NN que gera o próximo valor de Q ($Q(s',a';\theta_i)$), que é usado para atualizar o valor atual de Q ($Q(s,a;\theta_i)$), também gera o próprio Q, esse fato pode leva a instabilidade[10].

Em 2015, fomos apresentados ao algoritmo *Deep-Q-Networks* (DQN) que foi capaz de alcançar as pontuações do estado da arte em vários jogos de Atari 2600 tendo como entrada somente o pixels da tela. Esse algoritmo utilizou uma rede neural convolucional para aproximar a *Q-function* em relações aos estados (pixels da imagem jogo). Já foi discutido nas sessões anteriores (sessão 2.3.17) como as CNN possuem uma enorme vantagem em relação as NN clássicas ao se trabalhar com imagens, entretanto CNN ainda são um tipo de NN e possuem os mesmos problemas abordados anteriormente.

O DQN usou de três técnicas para restaurar a estabilidade do aprendizado. A primeira, experiências do agente, $e_t = (s_t,a_t,r_{t+1},s_{t+1})$ são armazenadas no que chamamos de ***replay memory*** $\mathcal{D}$ e são amostradas uniformemente aleatórias durante o treinamento. Dessa forma, removemos a correlação de sequências de observações suavizando a mudança na distribuição dos dados[1]. A segunda técnica foi utilizar uma rede neural separada que gera os valores de $Q(s',a')$, chamada de *target network* ($\hat{Q}$), desacoplando assim o *feedback* que era gerado com a mesma *network* atualizando ambos os valores de Q. A *target network* é uma copia idêntica da CNN principal exceto pelo fato que seus parâmetros θ^- são atualizados de forma a ficarem iguais a θ a cada 10000 iterações. E por último, foi usado um método de otimização com *learning rate* adaptativa como o RMSProp (um pouco mais detalhes na sessão 2.3.13.4) que mantém um ajuste de *learning rate* por parâmetro, e atualiza α de acordo com o histórico do gradiente (*momentum*). Esse passo serve para compensar a falta de um conjunto de treinamento fixado, a mudança da natureza de $\mathcal{D}$ pode requerer que certos parâmetros sejam mudados novamente após atingirem um certo ponto fixo[10].

A DQN a cada iteração de treinamento i, armazena as experiências $e_t = (s_t,a_t,r_{t+1},s_{t+1})$, obtidas com a interação entre o agente e o ambiente naquela iteração, na *replay memory* $\mathcal{D}$ e depois amostra uniformemente da mesma atualizando a *loss function*. Assim, a função de custo da network no DQN é determinada com a equação 2.35.

$$L_i(\theta_i) = \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1})\sim\mathcal{D}}\Big[\big(R + \gamma \max_{a'} \hat{Q}(s',a';\theta^-) - Q(s,a;\theta_i)\big)^2\Big] \tag{2.35}$$

A arquitetura do modelo utilizada no DQN de 2015, foi uma CNN com 3 camadas de convolução e 2 camadas totalmente conectadas e a entrada foi um volume de 84 x 84 x 4 (4 imagens de 84 x 84 monocromáticas). A primeira camada de convolução tinha 32 filtros (profundidade) de 8 x 8 (*receptive field* de 8) com *stride* de 4. A segunda camada de convolução possuía profundidade de 64, com *receptive field* de 4, com stride de 2. Já, a última camada convolutiva possuía 64 filtros de 3x3 com stride de 1. A primeira camada densa tinha 512 neurônios. Cada uma dessas camadas teve como função de ativação a

ReLU. A última camada densa era uma camada linear com uma saída para cada ação possível dentro dos jogos. A arquitetura utilizada pode ser vista na figura 2.44.

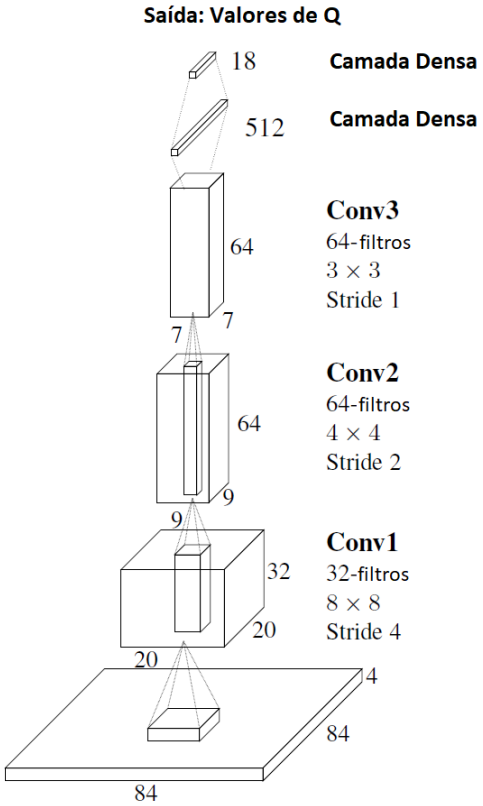

Figura 2.44: [Arquitetura de rede neural utilizada pelo DQN. Adaptado de: Hausknecht and Stone [10]

2.3.19 Observabilidade Parcial

No mundo real, são raros os casos em que o estado total do ambiente/sistema pode ser provido ao agente ou mesmo determinado. Em outras palavras, a propriedade de *Markov* é raramente encontrada em ambientes do mundo real, assim uma MDP não é capaz de modela-los. Para esse tipo de problema, utilizamos uma extensão das *MDP*s chamada de *Partially Observable Markov Decision Process* (POMDP), traduzida como Processo de decisão de Markov para observabilidade parcial. A POMDP captura a dinâmica desse tipo de sistemas ao tratar explicitamente que "estados" recebidos pelo agente são apenas uma pequena parte do estado do ambiente e o denomina de observação[10]. Nas POMDP a interação com o ambiente é descrito pela equação 2.36.

$$O_0, A_0, R_1, O_1, A_1, R_2, O_2, A_2, R_3, \cdots, O_{T-1}, A_{T-1}, R_T, O_T \tag{2.36}$$

no qual, $R \in \mathcal{R} \subset \mathbb{R}$, $A \in \mathcal{A}$ e $O \in \mathcal{O}$.

Algoritmos de RL que usam funções paramétricas para aproximação das suas *values functions* conseguem ser utilizados para os casos de observabilidade parcial sem nenhuma mudança[2].

2.3.20 Deep Recurrent Q-Network

Embora as funções paramétricas possam ser aplicadas sem mudanças aos casos de observabilidade parcial, suas aproximações podem ser arbitrariamente ruins devido ao fato que $Q(o,a|\theta) \neq Q(s,a|\theta)$. Para lidar com o problema de observabilidade parcial, o algoritmo DQN amontoa uma sequência de 4 *frames* e os utiliza como entrada para sua rede neural. Desta forma, o volume de entrada criado, possui as informações de velocidade, aceleração e direção dos objetos dentro da observação. Utilizando essa abordagem conseguimos tratar o ambiente de Atari 2600, como uma MDP[10].

Entretanto, em ambientes que possuem observações com informações incompletas do estado do ambiente, como por exemplo a tela piscando em um jogo de atari 2600, a performance de aprendizado do DQN é comprometida. Para lidar com esse problema, Hausknecht and Stone [10] desenvolveram a arquitetura do *Deep Recurrent Q-Network* (DRQN). Essa arquitetura substitui a primeira camada densa do DQN por uma camada do tipo ***Long Short-Term Memory*** **(LSTM)**. LSTM é um tipo de rede neural recorrente (*recurrent neural network* (RNN)) desenvolvida por Hochreiter and Schmidhuber [30] em 1997 para lidar com o problema de desaparecimento de gradiente das RNNs tradicionais ao processar longas sequência de dados.

Redes neurais do tipo recorrente são próprias para o tratamento de sequência de dados, elas realizam um *loop* iterando sobre uma sequência de entradas, um elemento por vez e decidindo quais informações dentro dos dados já avaliados são uteis para o próximo elemento da sequência. A figura 2.45 demonstra como ocorre essa iteração sobre a sequência x_t. Uma rede neural recurrente pode ser pensada como várias cópias da mesma rede neural, com cada uma passando uma mensagem para sua sucessora[31].

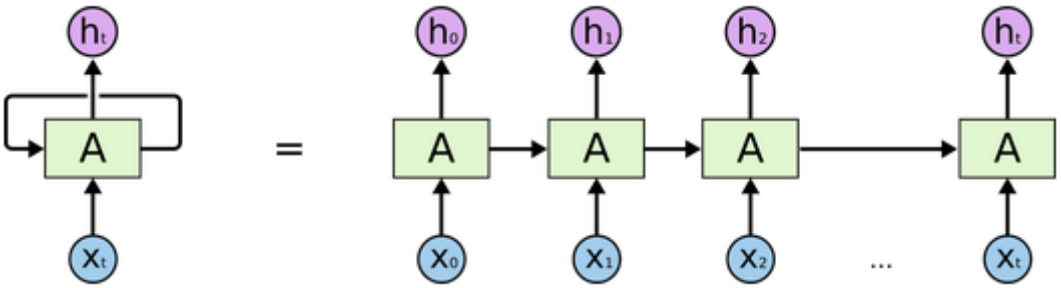

Figura 2.45: Princípio de funcionamento de uma rede neural recurrente ao processar uma sequência x_t. Retirado de: Olah [31]

Assim, redes neurais do tipo recorrente possuem uma capacidade de memória em relação as suas entradas anteriores, logo, elas conseguem aproximar melhor os valores de Q dado uma sequência de observações, levando a um melhor aprendizado em ambientes parcialmente observáveis. A arquitetura total do DRQN é demonstrada na figura 2.46. Na imagem são mostrados os dois últimos *frames* de uma sequência de observações sendo processados pela NN. No caso do DRQN, a saída (valores de Q) é dada apenas quando o último *frame* da sequência é processado.

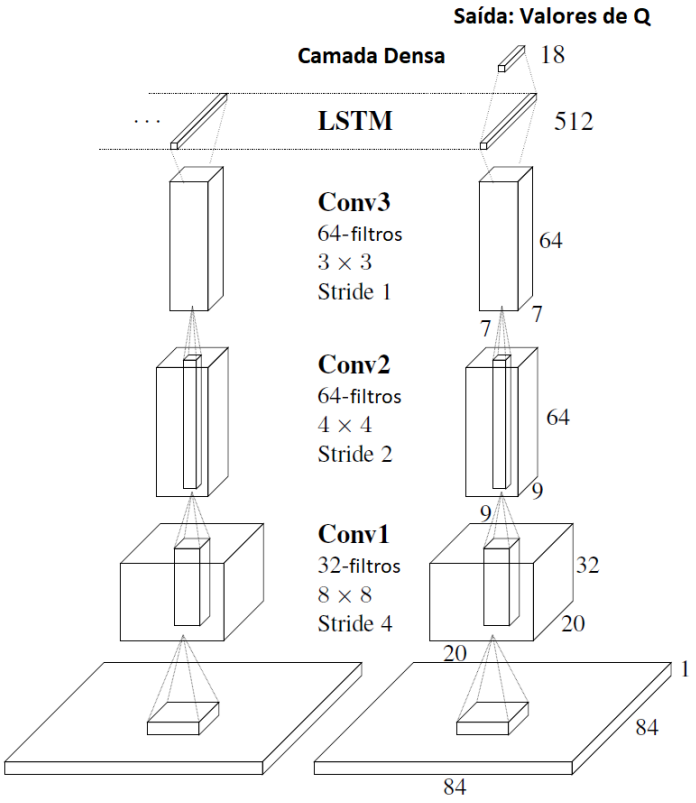

Figura 2.46: Arquitetura de rede neural utilizada pelo DRQN. Na imagem são mostrados os dois últimos *frames* de uma sequência de observações sendo processados pela NN. No caso do DRQN, a saída (valores de Q) é dada apenas quando o último *frame* da sequência é processado. Adaptado de: Hausknecht and Stone [10]

Para mais detalhes sobre os princípios de funcionamento das redes neurais recorrentes básicas e as do tipo LSTM são indicadas as seguintes fontes: Karpathy [32], Olah [31] e capítulo 6 a partir da sessão 2 do livro de Chollet [21] (possui diversas implementações praticas em Python com a API Keras).

Capítulo 3

Desenvolvimento

Neste capítulo são mostrados o ambiente tridimensional usado pelo agente, a API escolhida para o uso do *deep learning*, o desenvolvimento dos algoritmos de RL usados, as etapas de pré-processamento das imagens, a criação dos mapas tridimensionais.

3.1 Ambiente tridimensional

O ambiente tridimensional escolhido para o treinamento do agente foi o ambiente ViZDoom, disponível para download gratuitamente em http://vizdoom.cs.put.edu.pl/. ViZDoom é uma plataforma, baseado no jogo de FPS (*First Person Shooter*) Doom, para pesquisa em *visual reinforcement learning*, introduzido pelo trabalho de Kempka et al. [33]. A plataforma da acesso a um ambiente 3D, semi realístico, com visão em primeira pessoa. Além disso, o usuário tem total controle sobre o ambiente com as ferramentas disponibilizadas pela API (*Application Programming Interface*). As principais características da plataforma ViZDoom são:

- API de fácil uso para C++, Python e Java
- Facilidade de criar cenários customizados
- Modos Single-player (sync e async), multi-player (async), e modo espectador para AI.
- Resolução e parâmetros de renderização customizáveis
- Renderização off-screen

- Velocidade (até 7000 frames por segundo)
- Acesso ao buffer de profundidade (visão 3D)
- Uma plataforma leve (alguns MBs)
- Suporte a Linux e Windows

3.2 Deep Learning API

Para o desenvolvimento das redes neurais utilizadas neste projeto foi utilizado a API chamada Keras em conjunto do framework conhecido como Tensorflow (versão para GPU).

3.2.1 TensorFlow

TensorFlow é uma biblioteca open source para o cálculo numérico de alta performance e atualmente a biblioteca mais utilizada para o desenvolvimento de *machine learning*. Sua arquitetura flexível permite o desenvolvimento de cálculos entre uma variedade de plataformas (CPUs, GPUs) e dispositivos (desktops, clusters, dispositivos móveis). O TensorFlow foi desenvolvido pelos pesquisadores e engenheiros do Google Brain (organização responsável por parte da pesquisa em inteligência artificial dentro do Google) com propósitos de conduzir pesquisas em machine learning e *deep neural networks*.

Na arquitetura do TensorFlow, criamos grafos computacionais nos quais os dados de entrada (tensores) irão fluir. Essa abordagem nos permite maior flexibilidade na criação de modelos, mantendo por exemplo, toda estrutura de cálculo alterando apenas o tipo do tensor de entrada. Logo, com o TensorFlow neste ano de 2018, primeiramente, precisamos declarar a variáveis e depois especificar as operações a serem realizadas. Assim, quando estamos trabalhando com projetos grandes como redes neurais, demandamos mais linhas de código, o que cria um fator dificultante para prototipagem rápida e interpretação do código.

3.2.2 Keras

Keras é uma API de alto nível para a criação de redes neurais, escrita em Python e capaz de rodar no topo das bibliotecas TensorFlow,CNTK, ou Theano. O Keras segue as melhores práticas para reduzir a carga cognitiva (facilitando assim a programação em relação ao TensorFlow). O Keras oferece APIs simples e consistentes, minimiza o número de ações requeridas do usuário para a criação das NN, além de prover um excelente feedback de erro ao usuário.

3.3 Hardware utilizado

Para os treinamentos dos agentes, foi utilizado o computador de mesa do aluno, que possui as seguintes especificações:

- Processador: Core i7 4790k de 4.0 GHz
- Memória Ram: 32GB DDR3
- Disco rígido: SSD de 250GB + 1 HD de 1TB
- Placa gráfica: 2 NVIDIA GeForce GTX 970 com memória de 4GB

Logo todos os índices de desempenho de processamento foram feitos com essas especificações.

3.4 Desenvolvimento dos algoritmos

Para uma maior velocidade no desenvolvimento inicial dos algoritmos de RL foram utilizados ambientes bidimensionais relativamente simples. Devido a sua simplicidade, esses ambientes propiciam uma convergência de aprendizado mais rápida, facilitando assim a prototipagem e a localização de erros dentro dos códigos. Os ambientes bidimensionais escolhidos fazem parte da biblioteca chamada Gym. Gym é um *toolkit* para o desenvolvimento e teste de algoritmos de *reinforcement learning*. Essa biblioteca possui diversos sistemas clássicos de controle, problemas de robótica e jogos de Atari.

Além da utilização desses ambientes para o desenvolvimento dos códigos finais, foram realizados diversos testes em cada um dos ambientes, variando os hiperparâmetros na aprendizagem e vendo suas consequências no processo de aprendizado.

3.5 Desenvolvimento do algoritmo DQN

Para o desenvolvimento do algoritmo DQN/DRQN foi escolhido o jogo Pong de Atari 2600. Um agente consegue aprender a jogar o jogo Pong com maestria com média de 2 horas de treinamento (no computador do aluno) utilizando os hiperparâmetros especificados por [34]. Essas características o fizeram ser escolhido para o desenvolvimento inicial do código. Uma vez que código do DQN encontrava-se funcional com o Pong, o mesmo foi expandido para o problema de aprendizagem de navegação em um ambiente tridimensional. Para essa adequação, a estrutura básica do algoritmo DQN não precisou ser alterada, demonstrando assim a sua capacidade de generalização para inúmeros ambientes de aprendizagem.

Sobre o jogo de Pong: é um jogo de Atari 2600, no qual objetivo é marcar pontos sobre o adversário e evitar sofrer pontos do mesmo, similar ao esporte de tênis de mesa. Neste ambiente ganha quem conseguir marcar 21 pontos primeiro. A figura 3.1 mostra o ambiente do Pong dentro da biblioteca Gym.

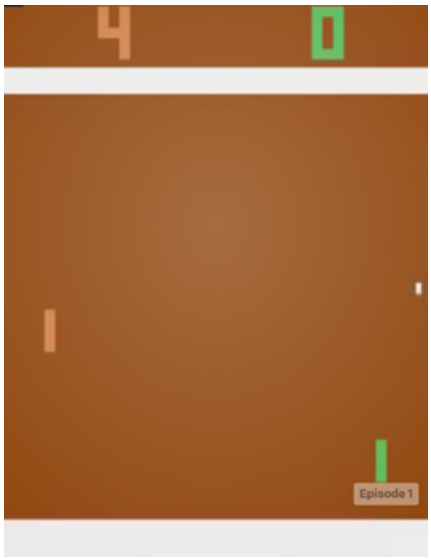

Figura 3.1: Imagem do jogo Pong de Atari 2600 dentro da biblioteca Gym.

3.6 Pré-Processamento das imagens de entrada

Antes de entrar na rede neural do DQN, cada *frame*(imagem do jogo) necessita ser redimensionado para um tamanho pré-definido pelo usuário na inicialização do agente. A figura 3.2 demonstra o redimensionamento em tamanho real de uma imagem obtida pela plataforma VizDoom.

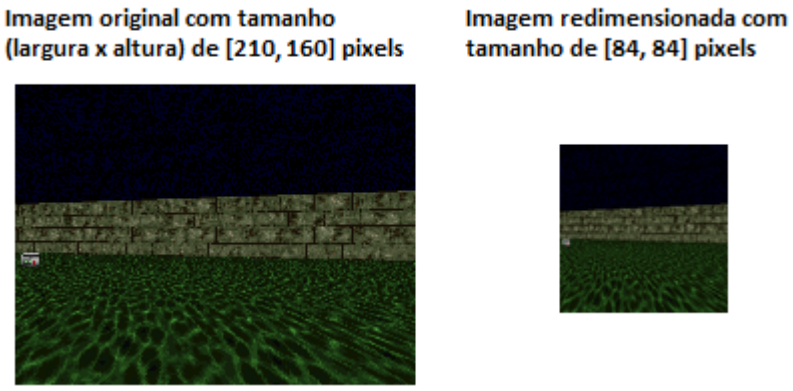

Figura 3.2: Diferença entres a imagens do ambiente ViZDoom original e redimensionada.

Além de redimensionadas, as imagens precisam ser amontadas (concatenadas), para a criação do volume de entrada/estado que será enviado a rede neural. A quantidade de

imagens a serem concatenadas em um volume é um hiperparâmetro chamado de tamanho de histórico. O volume criado terá dimensões de acordo com o tipo de arquitetura de rede neural a ser utilizada. Caso a arquitetura utilizada seja a do DRQN, o volume de entrada terá as seguintes dimensões [tamanho do histórico, largura, altura, canais de cores], pois nesse tipo de arquitetura a rede neural itera sobre cada um dos elementos de uma sequência (neste caso, cada um dos frames), decidindo quais informações são úteis para serem agregadas no processamento no próximo elemento. Já no caso da arquitetura do DQN, devemos concatenar os frames da seguinte forma [largura, altura, canais de cores × tamanho de histórico]. Como o DQN não possui ”memória”, devemos passar todas as imagens como se fosse apenas uma única imagem com N canais (N = canais de cores × tamanho de histórico). A figura 3.3 demonstra a criação do volume de entrada para um algoritmo com histórico de 4 frames, a imagem sobreposta é uma ilustração sobre como o agente ”vê” o volume de entrada, sendo o frame mais recente mais claro/sólido . É possível observar que ao concatenar uma sequência de frames, ganhamos informações sobre a direção, velocidade e aceleração da bola, do agente e do adversário. Todos os estados/volumes de entrada serão representados como imagens sobrepostas para fins de visualização.

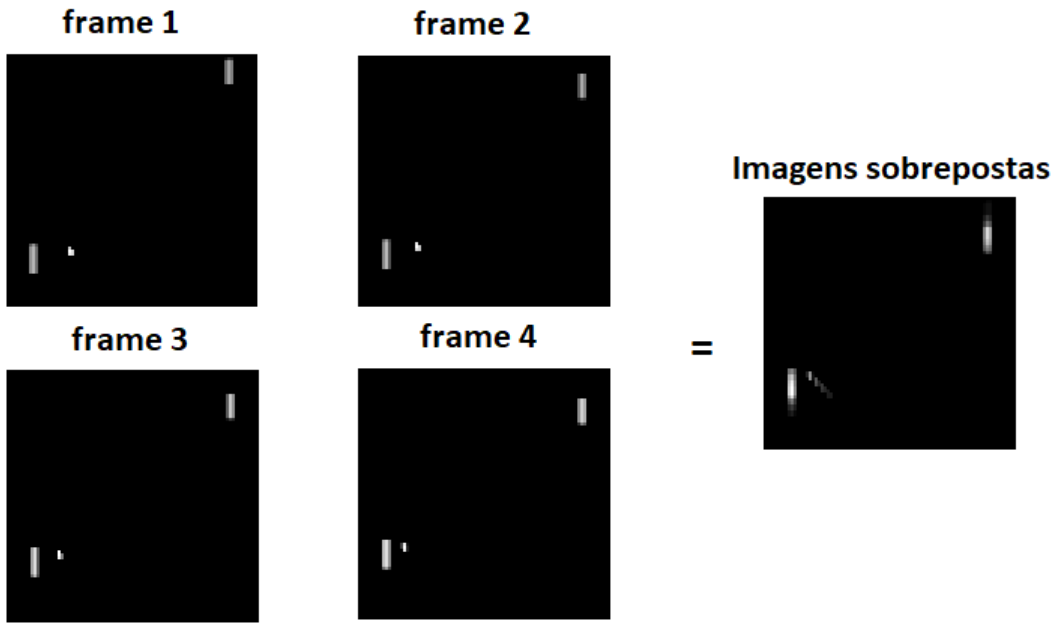

Figura 3.3: Demonstração da criação de um volume de entrada com tamanho de histórico 4.

Como última etapa antes de entrar na primeira camada de convolução, cada pixel do volume de entrada é normalizado utilizando a formula 2.27 introduzida na sessão 2.3.14. A seguir é demonstrado a parte do código que realiza essa operação. No código, Lambda se refere a uma camada genérica que realiza uma função anônima, neste caso é a aplicação

da equação 2.27.

```
lamb = Lambda(lambda x: (2 * x - 255) / 255.0, )(input)
```

3.7 Desempenho de Processamento

Durante o desenvolvimento do algoritmo foram buscadas as melhores maneiras de aumentar o processamento de frames/segundo. A parte que demanda mais tempo de processamento é a utilização das redes neurais. Durante o cálculo do erro de treinamento das redes neurais, necessitamos dos resultados de ambas as redes neurais Q e $\hat{Q}$ para todas as N amostras colhidas da *Replay Memory*. Logo, essa parte do código foi pensada de forma a aproveitar o máximo da computação vetorizada, assim *for loops* em python nativo foram substituídos pela vetorização em Numpy (principal biblioteca matemática do python) e posteriormente foram mudados para vetorização em Tensorflow e seu processamento mudado da CPU para a GPU (para tomar vantagem do paralelismo massivo das GPUs). Depois do uso da redes neurais, a parte que mais utiliza recursos de processamento é a amostragem das experiências a medida que *Replay Memory* aumenta.

Com esses dois problemas em mente foi pensado em uma abordagem de processamento em paralelo (*multi-threading*) do algoritmo DQN/DRQN. A figura 3.4 demonstra de forma simplificada o núcleo do algoritmo de DQN padrão. Em amarelo e vermelho estão sinalizados as duas etapas que mais demandam tempo de processamento. Como é possível notar, ambas as etapas encontram-se em sequência.

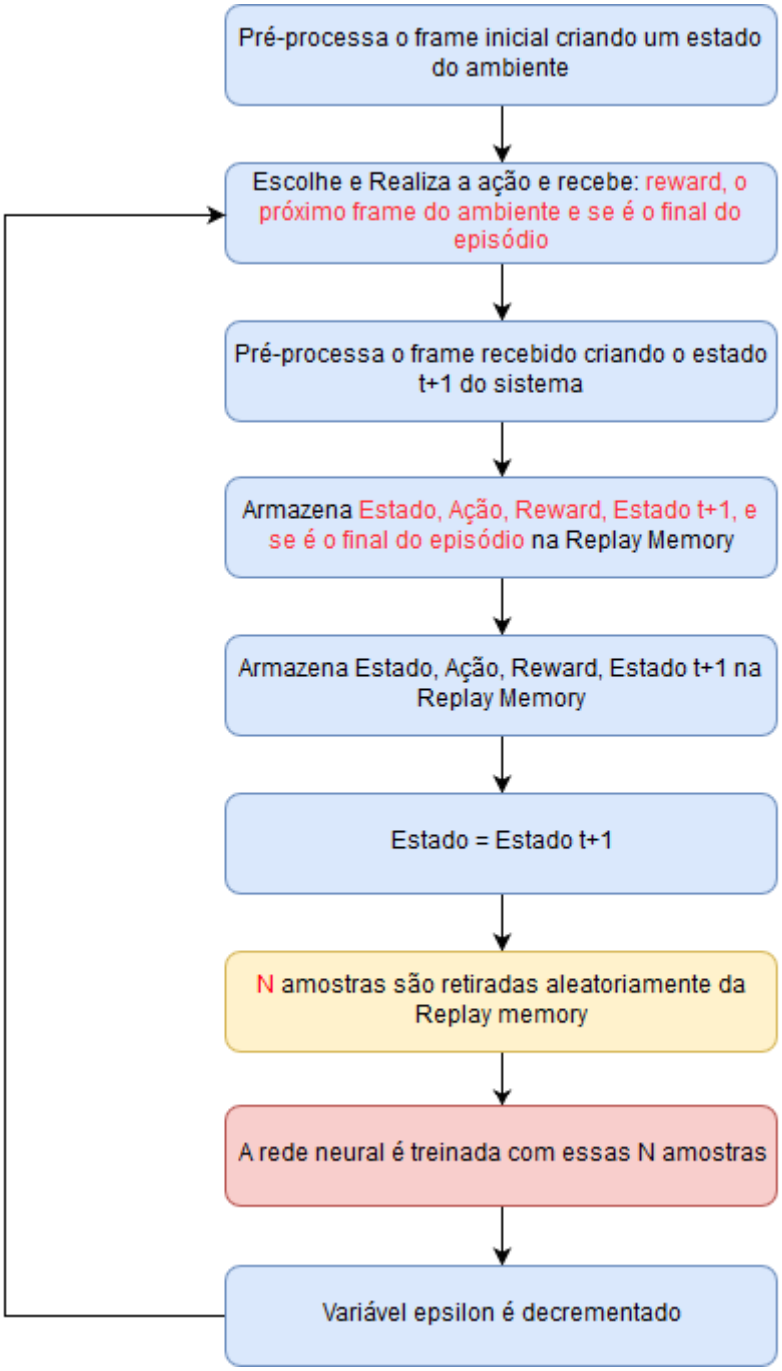

Figura 3.4: Funcionamento simplificado da versão padrão do algoritmo DQN/DRQN.

A parte da amostragem das experiências em nada influencia o restante do algoritmo. Assim, foi desenvolvida uma arquitetura para rodar a parte de amostragem em paralelo com o restante do código de aprendizagem, deixando as amostras colhidas a priori para o uso no treinamento das redes neurais. Em ambas as arquiteturas, a *Replay Memory* deve possuir uma quantidade de experiências armazenadas em quantidade maior ou igual ao *batch size* (N amostras colhidas). A versão *multi-threading* do algoritmo é mostrado na figura 3.5.

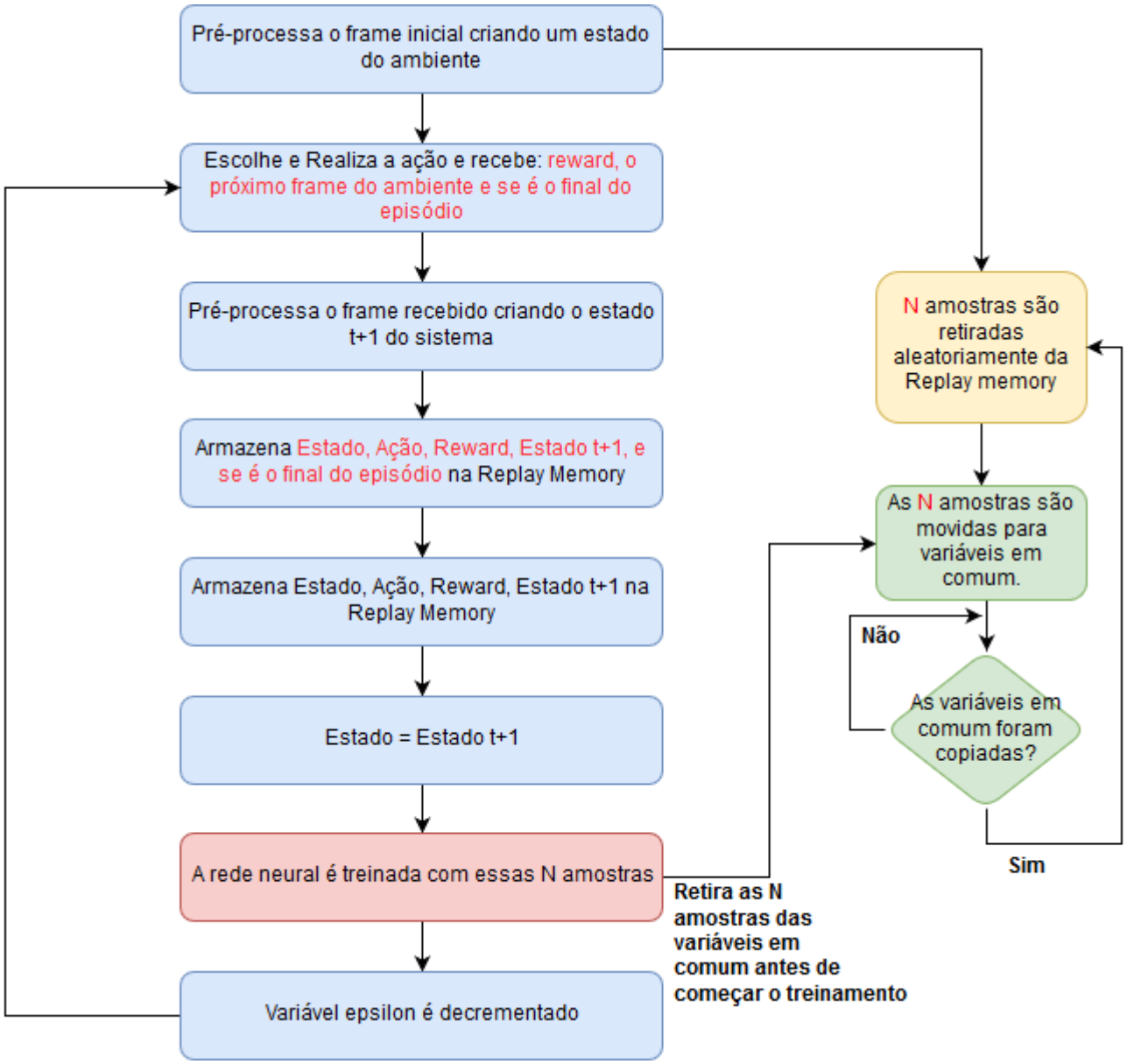

Figura 3.5: Funcionamento simplificado da versão paralelizada do algoritmo DQN/DRQN.

A figura 3.6 mostra os frames/segundo durante o aprendizado de um agente no jogo Pong com os mesmos hiperpâmetros durante 500000 frames. As curvas mais claras são os valores ao decorrer da simulação e as curvas mais escuras são as médias móveis a cada 20 amostragens. Os frames processados por segundo foram em média de 103.78 para a versão paralelo e 69.17 para a versão padrão.

É importante notar o decremento dos frames/segundos em ambas simulações na figura 3.6 nos primeiros 100000 frames. Esse decremento é justamente pelo aumento do uso das redes neurais para escolher as ações com os melhores valores de Q, a medida que a variável de exploração ϵ decaí. Nestas simulações o valor mínimo de ϵ é atingido em 100000 frames. Também é possível observar que o desempenho da versão paralelizada oscila com mais frequência, entretanto essa versão processa 34 frames/segundo a mais que

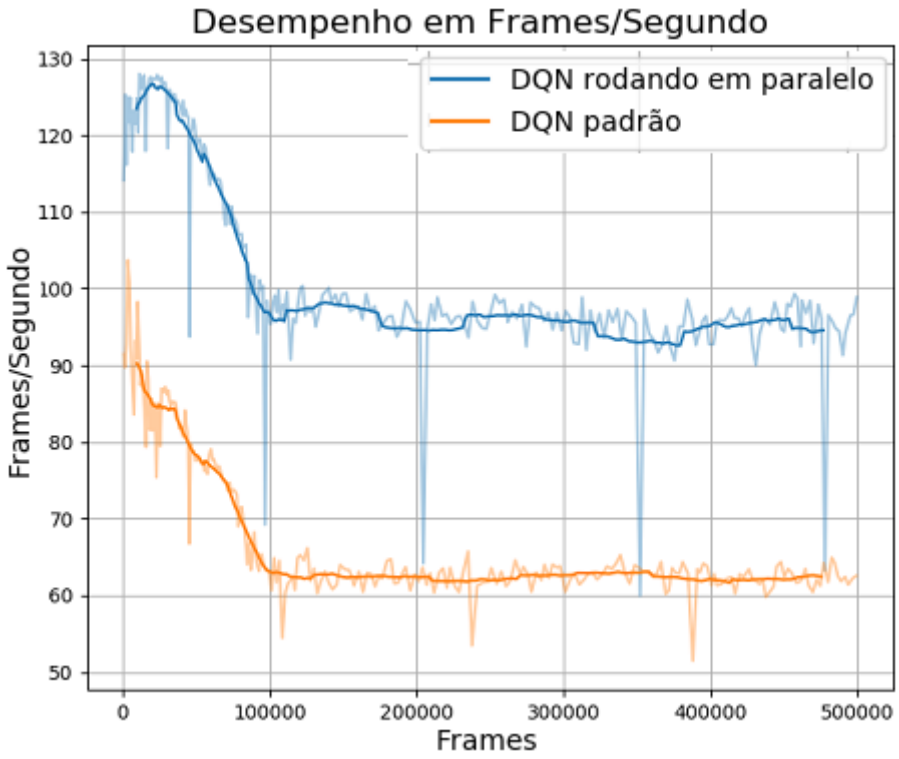

Figura 3.6: Exemplo de diferença de desempenho de processamento entre a arquitetura padrão e paralelizada do DQN ao longo do aprendizado.

a versão padrão. A versão padrão demorou 2 horas e 10 minutos para rodar os 500000 frames e versão paralelizada demorou 1 hora e 25 minutos. Logo, neste caso houve uma redução de 35% no tempo de processamento.

Como a amostragem das experiências começa antes que a experiência daquele time step seja adicionada a *Replay Memory*, isso impossibilita que essa experiência tenha a possibilidade de ser amostrada naquele momento. Em outras palavras, usando a terminologia de teoria de controle digital é como se tivéssemos introduzido um atraso de uma amostragem no algoritmo. Entretanto, como mostrado pela figura 3.7 o aprendizado em ambas as arquiteturas deu-se de forma "similar". As variações mostradas são devido ao sistema como um todo (ambiente + agente) ser estocástico, por mais que as sementes dos métodos aleatórios tenham sido fixadas, o fato de rodar os algoritmos na GPU introduz outras fontes de aleatoriedade. Esse fato é causado pelos algoritmos de otimização necessários para rodar os códigos na GPU.

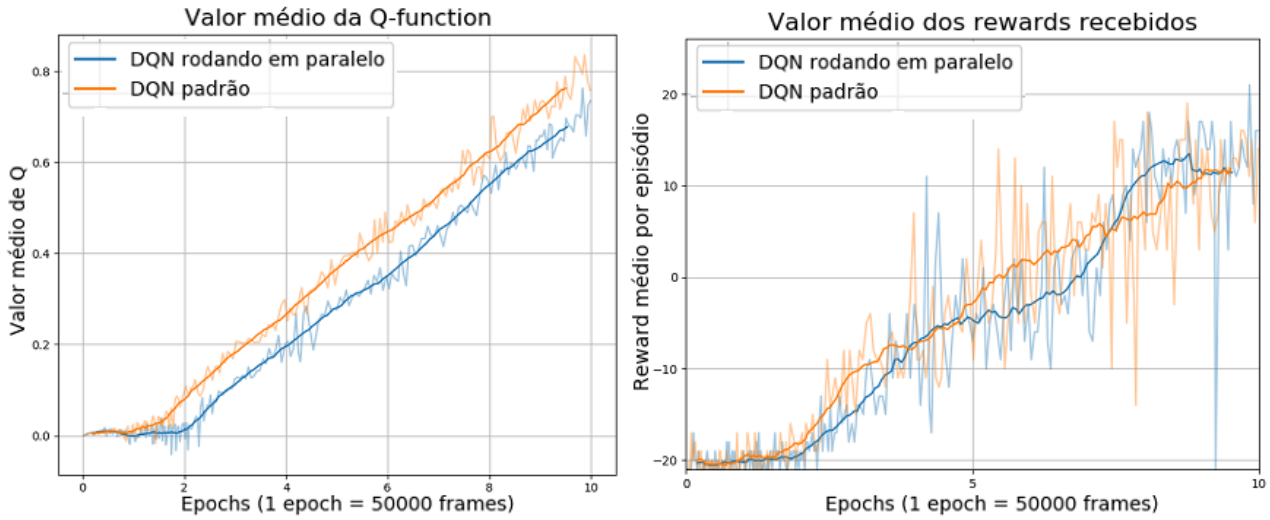

Figura 3.7: Valores médio da função Q e dos rewards obtidos durante o aprendizado na arquitetura padrão e paralelizada do DQN.

3.8 Criação dos mapas tridimensionais

Após a finalização do algoritmo a ser utilizado no aprendizado da navegação tridimensional foram criados os ambientes tridimensionais. Foram criados dois mapas para serem rodados na plataforma VizDoom. Ambos os mapas foram criados utilizando o software DoomBuilder disponível para download gratuitamente em: http://www.doombuilder.com/. Além do download, o site mencionado conta com uma série de tutoriais em vídeo para a criação de mapas. A textura que foi utilizada para a criação dos mapas foi a "freedoom2.wad"que vem em conjunto com a biblioteca VizDoom (encontra-se no diretório da biblioteca).

Para a criação dos mapas foi pensando um ambiente que imitasse algum problema de robótica móvel em relação a parte de navegação. O problema escolhido foi o de um robô móvel navegando de volta para sua plataforma de recarga de bateria. Assim sendo, foi criado um mapa no qual o agente tem como objetivo encontrar sua plataforma de recarga de bateria o mais rápido possível. Logo, o agente é incentivado a não perder tempo realizando ações em um mesmo lugar, a evitar colisões de qualquer tipo com o ambiente.

A figura 3.8 mostra a visão superior do primeiro mapa criado. Os círculos em vermelho sinalizam locais no qual o agente pode começar um episódio. O mapa é constituído de 2 salas separadas por um corredor "curvo" e em cada episódio o agente é colocado aleatoriamente em algum dos círculos olhando para alguma direção também aleatória. O quadro em verde no canto superior direito demonstra a posição da plataforma de recarga que o agente tem que alcançar. Já a figura 3.9 demonstra o mapa pela visão do agente. Durante o desenvolvimento do mapa, o teto foi colocado bem acima de tal forma que não

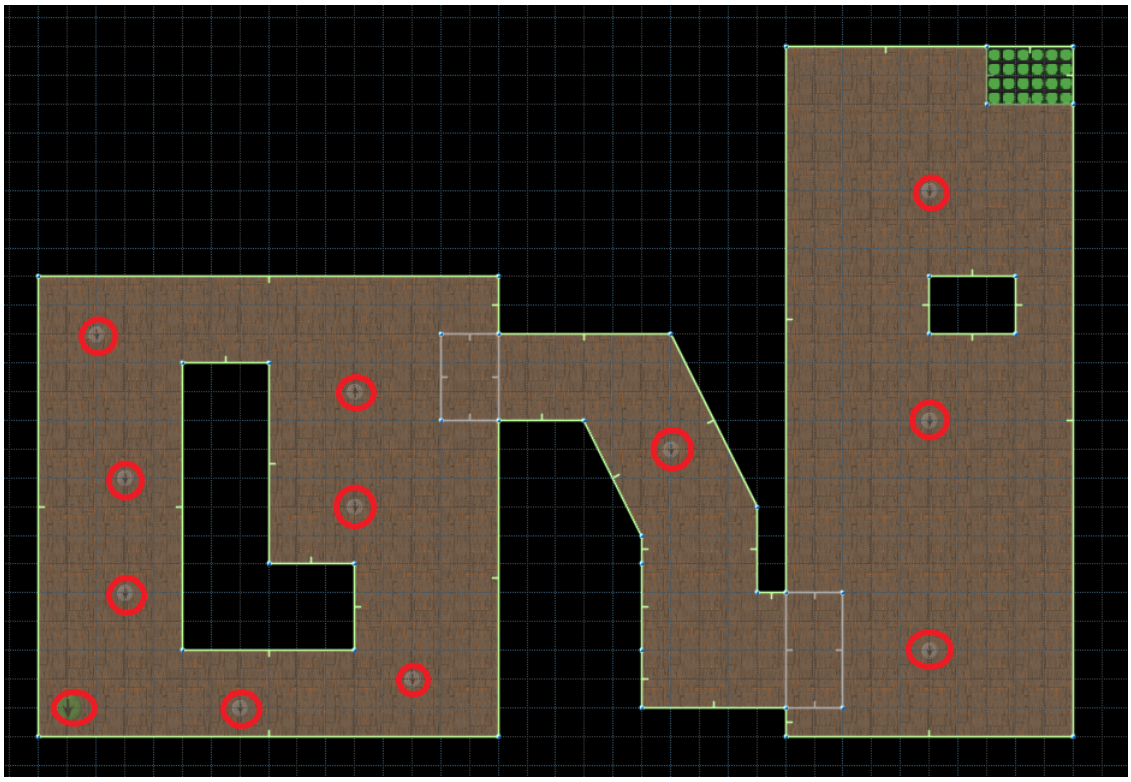

Figura 3.8: Visão superior do primeiro mapa tridimensional. Os círculos em vermelho sinalizam locais no qual o agente pode começar um episódio. O quadro em verde no canto superior direito demonstra a posição da plataforma de recarga que o agente tem que alcançar.

aparecesse na visão do agente. Isto foi feito para tentar aproximar a visão de um pequeno robô móvel navegando em uma sala. Além disso, cada sala possui texturas diferentes em suas paredes para que o agente possa identificar em que local ele se encontra.

Como mostrado na figura 3.10, o segundo mapa pode ser considerado o primeiro mapa rotacionado e com paredes a mais na sala que encontra-se o objetivo do agente. Entretanto nesse mapa, o agente não pode começar em uma posição na qual ele consiga ver de imediato a plataforma de recarga. O segundo mapa foi criado para testar as capacidade de generalização de aprendizado de um agente treinado no primeiro mapa.

Os mapas e todos os recursos utilizados nesse trabalho de conclusão de curso encontram-se no github do aluno, no qual o link encontra-se no anexo A.1.

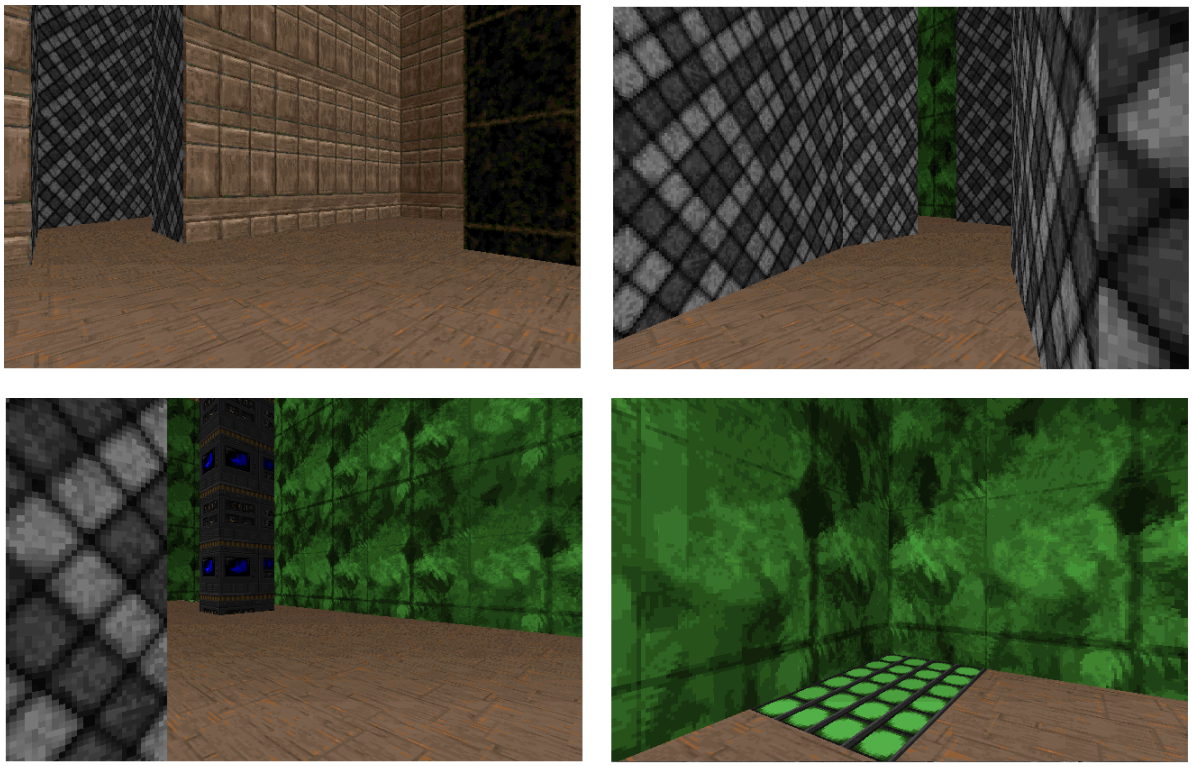

Figura 3.9: Imagens do ponto de vista do agente no primeiro mapa tridimensional. A imagem do canto inferior direito mostra a plataforma de recarga de baterias, objetivo do agente

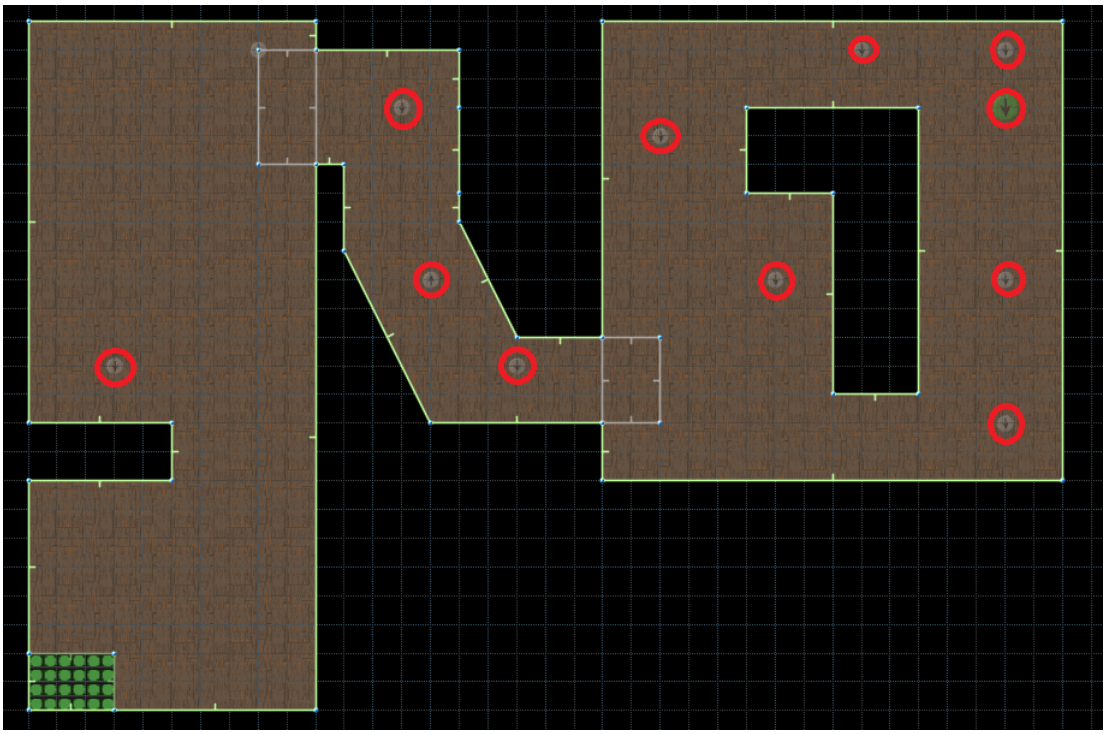

Figura 3.10: Visão superior do segundo mapa tridimensional. Os círculos em vermelho sinalizam locais no qual o agente pode começar um episódio. O quadro em verde no canto superior direito demonstra a posição da plataforma de recarga que o agente tem que alcançar.

Capítulo 4

Processo de aprendizagem

Neste capítulo são mostrados os resultados e suas devidas análises no processo de aprendizagem do agente nos ambientes: Pong e nos dois mapas tridimensionais criados na plataforma VizDoom.

4.1 Pong

O primeiro passo no processo de aprendizagem do jogo Pong foi a modelagem do ambiente como uma MDP que pode ser vista abaixo.

- Estados/Observações: Sequência de imagens obtidas da plataforma Gym.
- Ações:
  - Deslocar o pad para cima, para baixo e ficar parado (no-op).
- Rewards:
  - -1 por cada ponto marcado pelo adversário.
  - +1 por cada ponto marcado pelo agente.
- Fim de episódio: Quando todas as vidas se esgotam (uma vida acaba quando o adversário ou o agente marcam 21 pontos)

Para o aprendizado do jogo Pong foi utilizado o algoritmo DQN e foram fixados os hiperparâmetros mostrados pela figura 4.1. Esses hiperparâmetros foram baseado no trabalho de Lapan [34].

A arquitetura de rede neural utilizada para o treinamento do Pong foi a mesma utilizada por Mnih et al. [1] e no qual o número total de parâmetros é mostrado pela figura 4.2. Essa arquitetura apresenta 1685667 parâmetros a serem treinados.

Parâmetros	Valores
Frame skip	4 frames
Tamanho do histórico do agente	4 frames
Formato da imagem de entrada (altura, largura, canais de cores)	[84, 84, 1]
Discount factor (γ)	0.99
ε	[1.0, 0.02]
Decaimento linear	100000 frames
Atualiza Q-target	1000 frames
Tamanho da replay memory	100000 (experiências)
Número de frames no qual o agente toma somente ações aleatórias (preencher a replay memory antes do começo do treinamento)	10000 frames
Batch-size	32
Optimizer	Adam
Random seed	1

Figura 4.1: Hiperparâmetros do DQN fixos durantes os testes de aprendizado do jogo Pong.

```
Total params: 1,685,667
Trainable params: 1,685,667
Non-trainable params: 0
```

Figura 4.2: Número de parâmetros por camada da arquitetura do DQN proposta por Mnih et al. [1].

O único teste de aprendizado realizado com o jogo Pong foi a comparação de diferentes *learning rates* com o método de otimização (*optimizer*) ADAM (*Adaptive Moment Estimation*). O método de otimização Adam é uma variante do SGD proposta por Kingma and Ba [35] em 2014. É um método que combina as vantagens de duas outras variantes do SGD, AdaGrad e RMSProp, ao calcular *learning rates* adaptativas individualmente para cada parâmetro diferente utilizando estimativas dos momentos de primeira e segunda ordem dos gradientes. Em outras palavras, uma lr é mantida para cada peso da rede neural e é separadamente adaptado a medida que o aprendizado progride.

As figuras 4.3 e 4.4 mostram o resultado do treinamento com as lr de 0.00010 e 0.00025 com uma média móvel com janela de 20 amostragens. A imagem da direita na figura 4.3 demonstra os *rewards* por episódio ao longo do treinamento. Repare que o progresso da lr de 0.00025 é mais rápido nas primeiras 6 *epochs* (300 mil frames), entretanto, após esse ponto tem seu progresso diminuído e é ultrapassado pela lr de 0.00010 na *epoch* de número 10. Uma *learning rate* muito alta possui dificuldades em se estabilizar em um bom ponto no espaço de otimização. Mesmo que o algoritmo de otimização possua uma lr adaptativa, demorará mais para encontrar um bom ponto, se o sistema possui muita ”energia”.

A imagem da esquerda na figura 4.3 demonstra o valor médio da função de Q para

cada episódio. Podemos observar novamente o efeito descrito anteriormente para a lr de 0.00025. Usando uma analogia à teoria de controle seria como se houvesse um *overshoot* nos valores de Q para a lr mais alta.

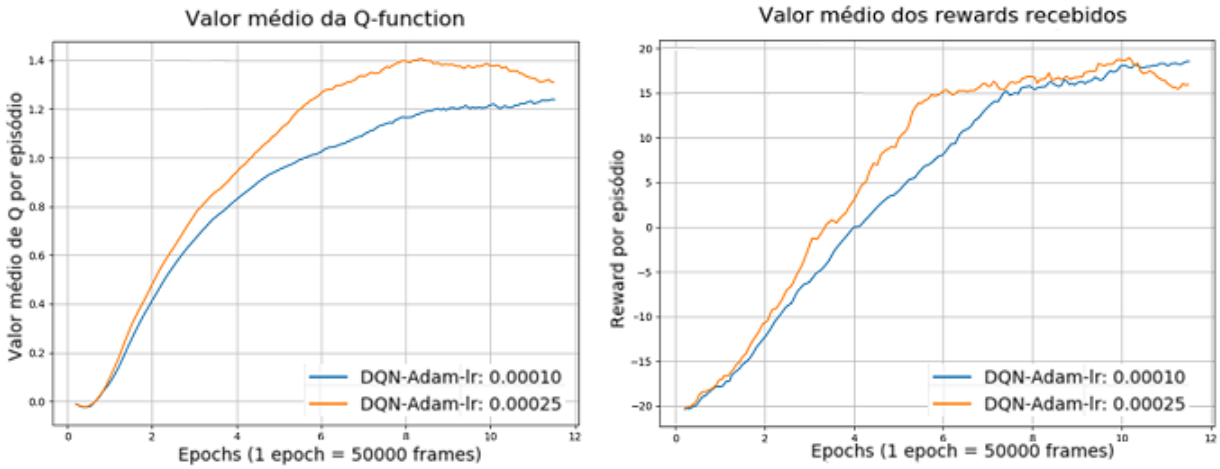

Figura 4.3: Valor médio da *Q-function* por episódio durante o aprendizado com duas learning rates diferentes do jogo Pong com *optimizer* Adam.

A figura 4.4 demonstra os valores médio da *loss* por episódio ao longo do treinamento. O valor da *loss* (função de custo) deve decrescer a medida que o agente fica mais confiante sobre quais as melhores ações a serem tomadas.

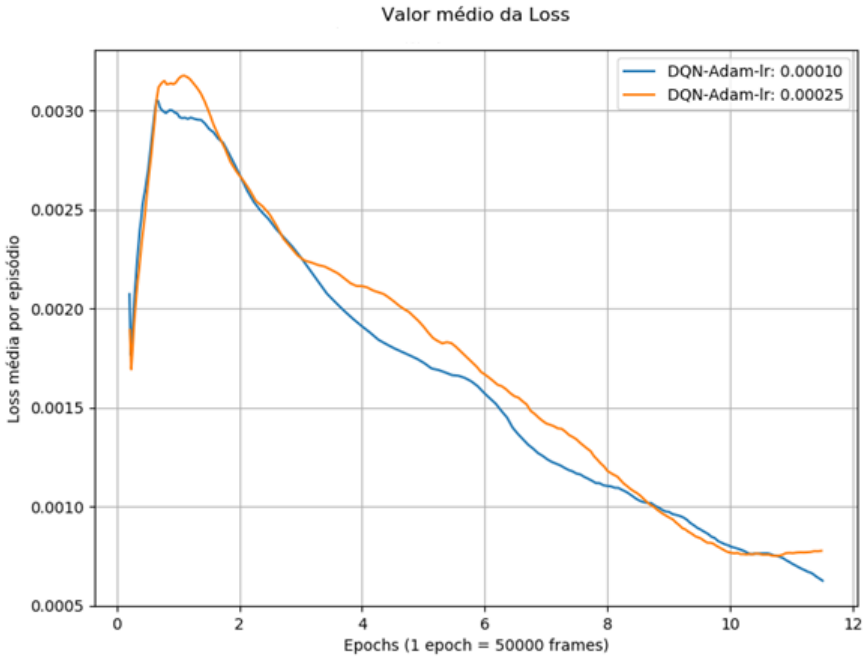

Figura 4.4: Loss média durante o aprendizado com duas learning rates do jogo Pong com *optimizer* Adam.

A figura 4.5 mostra o teste do agente jogando Pong com a *Q-function* aprendida ao longo de três momentos durante o treinamento com Adam e uma lr de 0.0001. A lr de 0.0001 foi escolhida devido ao seu rendimento em comparação a lr de 0.00025 mostrado nos parágrafos anteriores. Com a *Q-function* inicial o agente não consegue ir bem, de fato, essa *Q-function* é o mesmo que escolher as ações aleatoriamente. Com a *Q-function* do meio, o agente já aprendeu a rebater a bola mandada pelo adversário e consegue marcar alguns pontos. E com a última, o agente aprendeu com maestria como ganhar do adversário sem sofrer nenhum ponto. Fato curioso sobre a última *Q-function*, devido ao fato do jogo Pong ser simples e as ações do adversário fixas dentro do código do jogo, o agente aprendeu a explorar uma estratégia na qual ele sempre rebate a bola no mesmo local de forma a mesma ser refletida pela parede (sempre no mesmo ponto) e "enganar" o adversário.

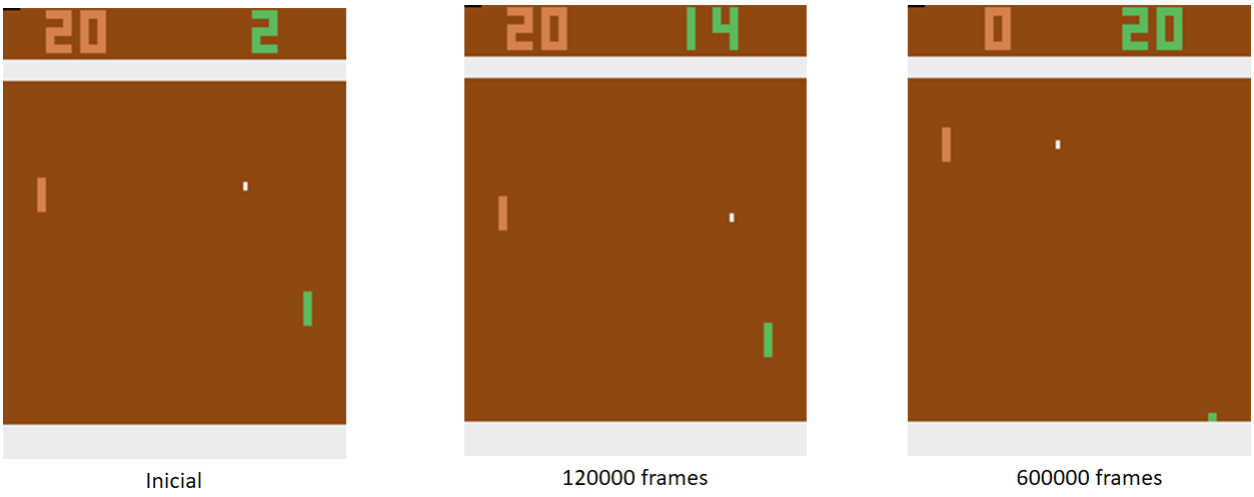

Figura 4.5: Resultado do agente jogando Pong em três momentos do treinamento.

4.1.1 Visualização do aprendizado das redes neurais

Para a visualização de aprendizado das redes neurais considere o estado mostrado pela figura 4.6. Esse estado é um volume de entrada no formato [84, 84, 4], ou seja, uma sequência de 4 frames de formato [84, 84, 1] concatenados.

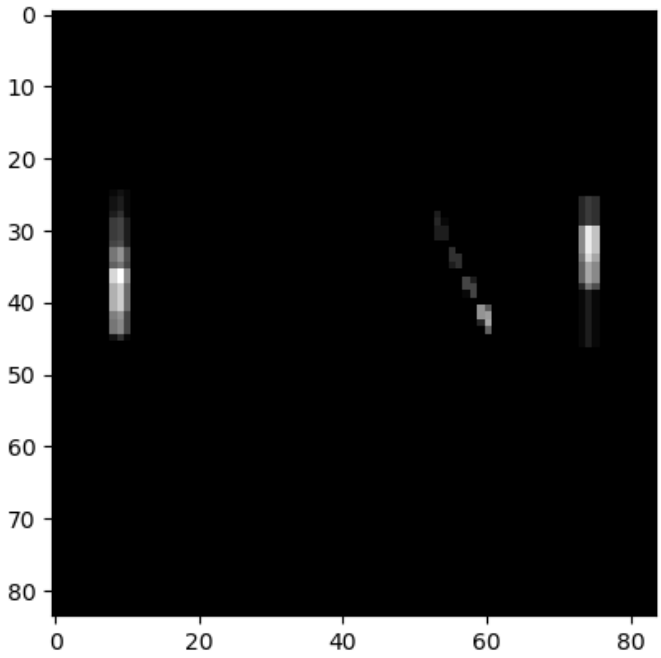

Figura 4.6: Um estado do jogo Pong que será enviado a rede neural para observação dos atributos aprendidos pela mesma.

Ao enviarmos o estado a uma rede neural com o pesos aprendidos depois de 600000 frames com o método de otimização Adam e uma *learning rate* de 0.0001, a mesma produz o *activation map* em sua primeira camada de convolução demonstrado pela figura 4.7.

A figura 4.7 mostra todos os 32 *activations maps*(tamanho de cada *activation map*: [20, 20]) gerados pelos 32 filtros da primeira camada convolutiva. Os pontos em branco são os locais dentro do *activation map* que possuem os maiores valores de ativação e os valores em preto os menores. Os valores em cinza claro demonstram valores medianos (podem ser considerado pontos sem ativação). Ao observar a imagem, vemos que vários dos filtros não são ativados com esse estado de entrada.

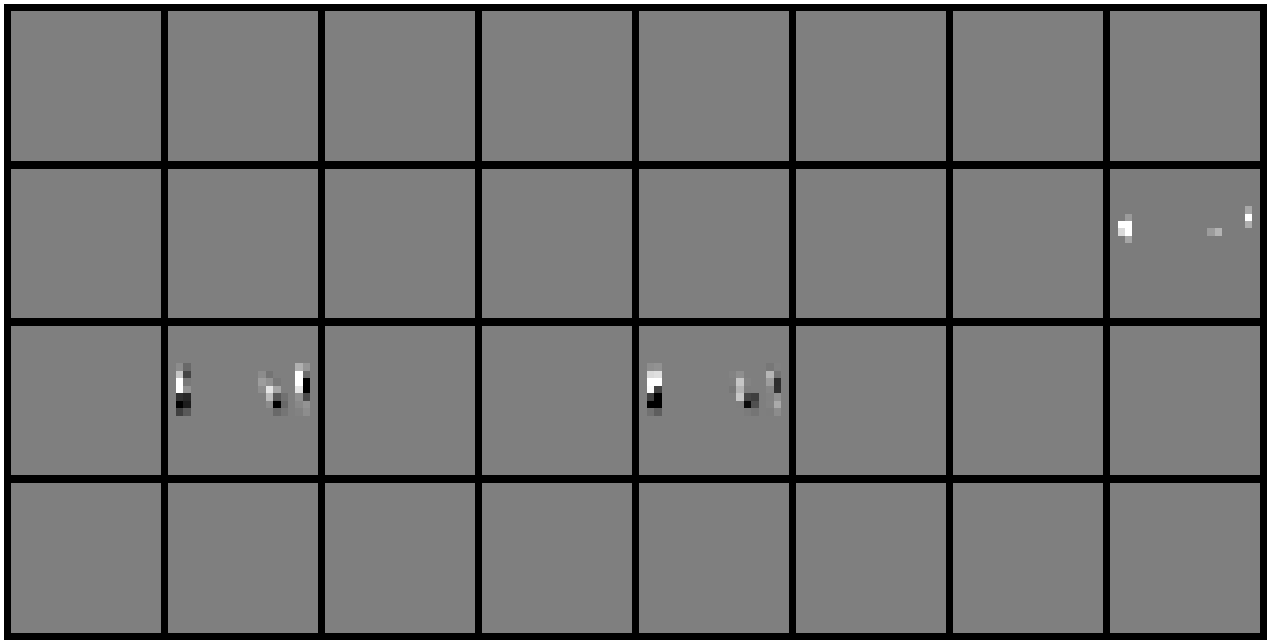

Figura 4.7: *Activation maps* da primeira camada convolutiva com o Jogo Pong.

Vemos que em cada um dos *activation maps* que vieram de filtros ativos "olham" para as extremidades do estado de entrada, locais no qual encontram-se a bola, o agente e seu adversário. Entretanto, podemos olhar de forma exata quais partes do volume de entrada estão gerando as maiores ativações dos filtros aprendidos em cada camada, dessa forma ganhamos uma intuição maior sobre o que esses filtros estão procurando na imagem. A figura 4.8 mostra os locais do volume de entrada que geraram as maiores ativações dos filtros de cada camada convolutiva. No canto superior esquerdo são mostrado os locais que tiveram as 20 maiores ativações dentre os filtros da primeira camada convolutiva. Já no canto superior direito, temos os locais que geraram as 3 maiores ativações dentre os filtros da segunda camada de convolução. E por último no canto inferior temos os locais que geraram as 3 maiores ativações da terceira camada convolutiva. Ao analisar a figura como um todo, é possível observar que os filtros aprenderam a detectar a posição do inimigo, da bola e do agente, sendo que cada camada olha uma região maior no volume de entrada.

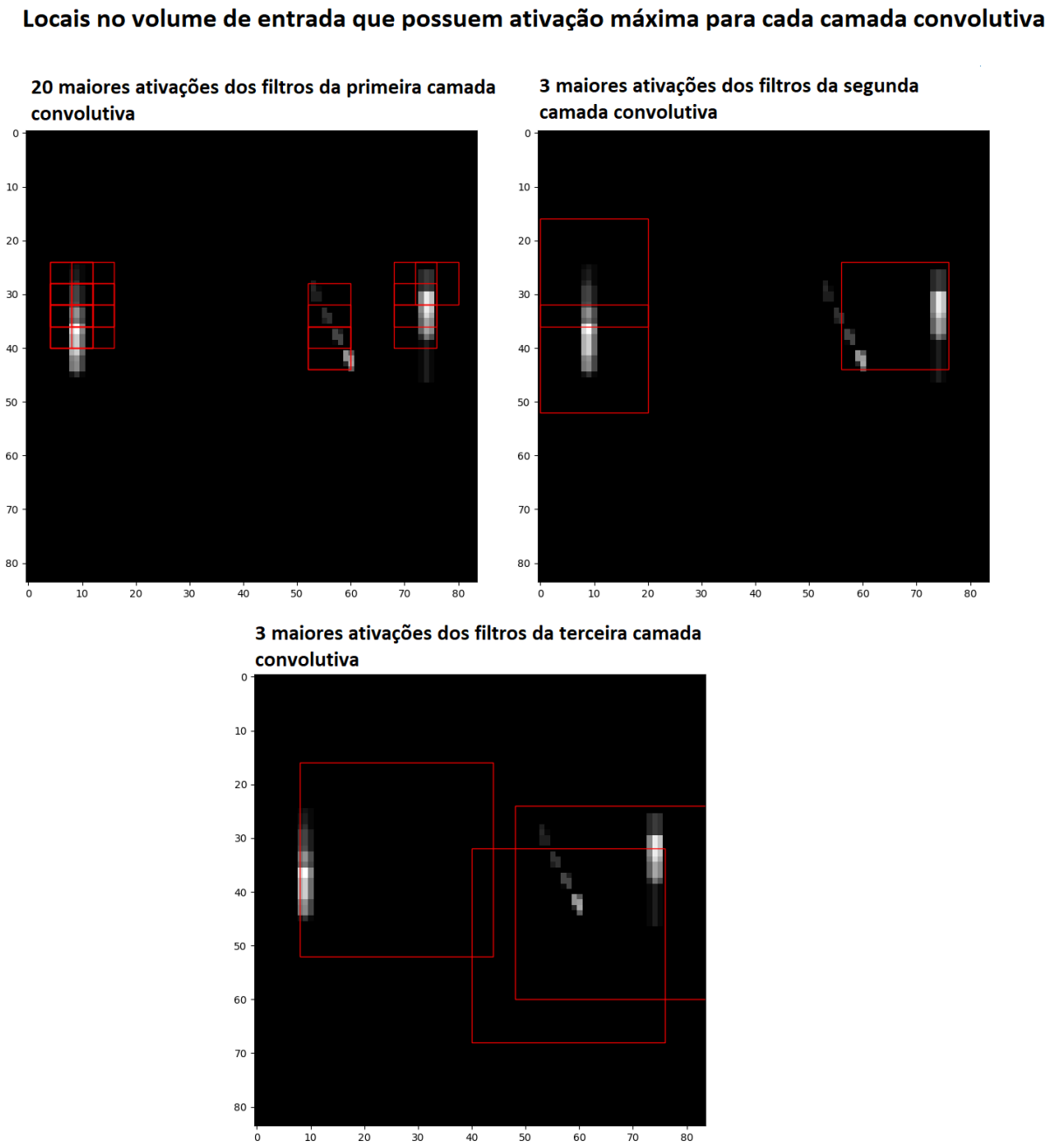

Figura 4.8: Locais do volume de entrada que geram as maiores ativações do filtros de cada camada de convolução no jogo Pong.

Cada filtro de uma camada de convolução possui a mesma profundidade do volume de entrada da camada, ou seja, no caso da primeira camada, cada filtro possui uma profundidade de 4. Como a primeira camada gera 32 *activation maps*, o volume de entrada da segunda camada convolutiva terá profundidade de 32, assim cada filtro dessa camada convolutiva terá uma profundidade de 32. Por ultimo, na terceira camada convolutiva, como temos um volume de entrada de 64 *activation maps*, cada filtro terá uma profundidade de 64. Logo, fica inviável a visualização dos filtros aprendidos, exceto pela primeira camada, no qual os filtros são mostrados pela figura 4.9. Cada quadrado subdividido em 4 é um filtro, cada subdivisão é uma *depth slice* do filtro. Como é possível observar, não conseguimos extrair nenhuma informação relevante sobre o que de fato o filtro detecta.

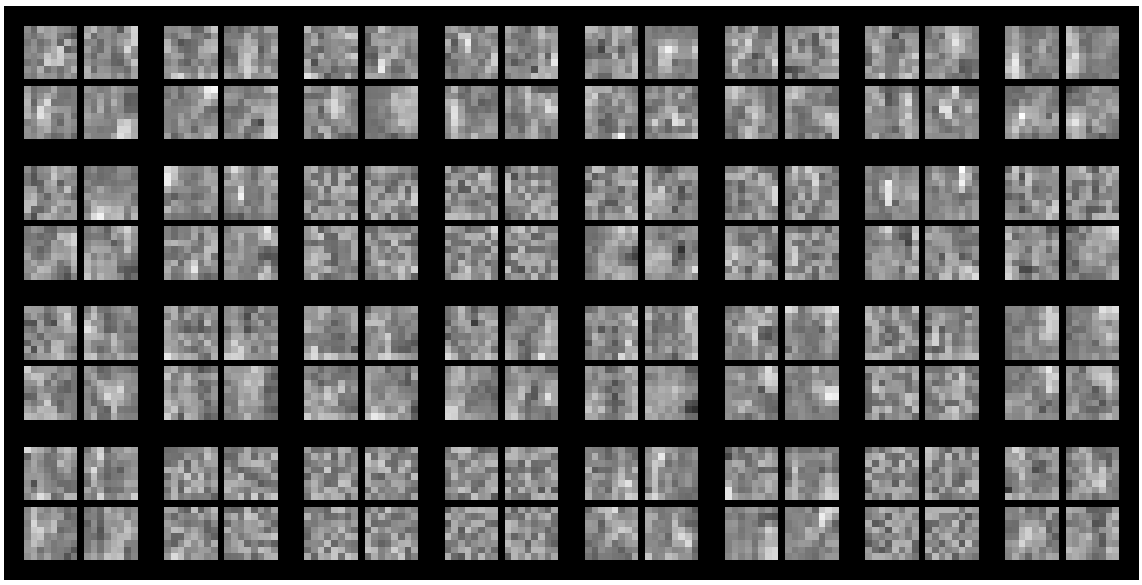

Figura 4.9: 32 Filtros aprendidos pela primeira camada de convolução ao jogar o jogo Pong

Entretanto, podemos utilizar um método de otimização para encontrar qual é o volume de entrada da rede neural que maximiza cada um dos filtros de uma camada de convolução. Esse método consiste em criar um volume de entrada formado de pixels escolhidos em uma distribuição normal e colocar uma função de custo para o filtro desejado. Assim, alimentamos a rede neural com o volume criado, calculamos a função de custo daquele filtro e realizamos o processo de gradiente ascendente em relação a imagem de entrada para maximizar aquela função de custo. Em sequência atualizamos os pixels do volume de entrada na direção do gradiente recebido. O link a seguir mostra em detalhes a implementação dessa abordagem em Keras e demonstra vários filtros de redes neurais conhecidas: https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html.

Utilizando um algoritmo explicado no link acima foram calculados quais os volumes de entrada que maximizam cada um dos filtros das camadas de convolução. A figura 4.10

demonstra os 32 volumes de ativação que conseguem a máxima ativação de cada filtro na primeira camada de convolução. Cada filtro está subdividido em 4, sendo cada subdivisão uma *depth slice* do volume de entrada. É possível observar que cada filtro busca uma textura diferente no volume de entrada, texturas que muito lembram o formato da bola do jogo.

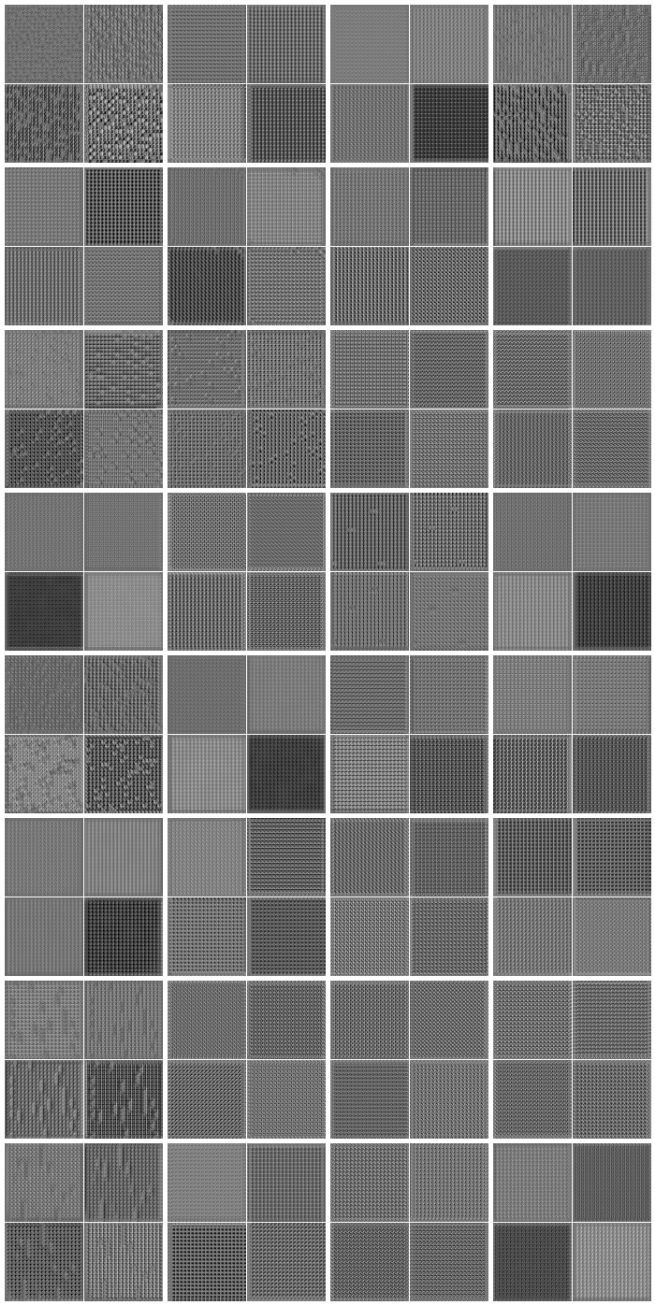

Figura 4.10: Os 32 volumes de entrada que maximizam os filtros da primeira camada convolutiva.

A figura 4.11 mostra com um zoom, o volume de entrada que maximiza o primeiro filtro da primeira camada convolutiva.

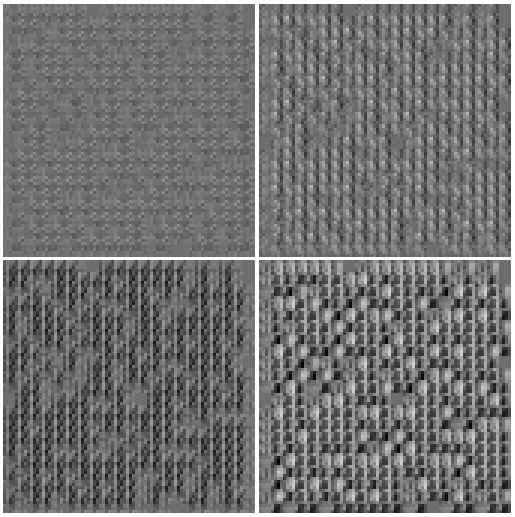

Figura 4.11: Zoom no volume de entrada que maximiza a ativação do primeiro filtro da primeira camada de convolução.

4.2 Navegação tridimensional

O primeiro passo para o aprendizado da navegação nos ambientes tridimensionais foi a modelagem do ambiente como a MDP que é mostrada abaixo.

- Estados/Observações: Sequência de imagens obtidas da plataforma ViZDoom concatenas em volumes.
- Ações:
  - Andar para Frente
  - Virar a câmera para direita e para esquerda
- Rewards:
  - -0.001 por cada *frame de jogo* vivo, incentivando o agente a localizar a plataforma de recarga o mais rápido possível.
  - -0.01 por cada 5 frames de jogo que o agente fique na mesma posição, incentivando o agente a não ficar parado executando ações, e consequentemente gastando bateria em vão, em uma mesma localização.
  - -0.1 por colisão em alguma parede ou objeto.
  - +10 ao atingir o objetivo final (a plataforma de recarga de bateria).
- Fim de episódio: Quando o agente chegar a plataforma de recarga de bateria ou o tempo acabar (3000 frames de jogo = 750 frames de simulação [devido ao fato que a mesma ação é executa por 4 frames de jogo dentro do algoritmo])

4.2.1 Primeiro mapa tridimensional

Para ensinar o agente a navegar no primeiro dos mapas tridimensionais criados, foi utilizado os algoritmos DQN e DRQN. Os hiperparâmetros fixados para todas as simulações dessa sessão são mostrados pela figura 4.12 e cada agente foi simulado por 5 milhões de frames. Os hiperparâmetros foram escolhidos com base nas simulações do jogo Pong mostradas no tópico anterior, e também nos recursos computacionais disponíveis.

Parâmetros fixos	Valores
Frame skip (Frames)	4 frames
Discount factor (γ)	0.99
ε	[1.0, 0.1]
Decaimento linear	250000 frames
Atualiza Q-target	10000 frames
Tamanho da replay memory	250000 (experiências)
Número de frames no qual o agente toma somente ações aleatórias (preencher a replay memory antes do começo do treinamento)	50000
Batch-size	32
Optimizer	Adam
Learning Rate	0.0001
Random seed	1

Figura 4.12: Hiperparâmetros fixos usados durante o aprendizado de navegação no primeiro mapa tridimensional.

A primeira simulação de aprendizado foi realizada com o algoritmo DQN e um tamanho de histórico igual a 4, ou seja, um estado é formado por 4 frames concatenados tendo o formato final de [84, 84, 4]. Nesta simulação, devido a complexidade dos ambientes tridimensionais, não era esperado um sucesso na aprendizagem. Entretanto, o algoritmo com esses hiperparâmetros foi capaz de aprender com sucesso a navegar no mapa tridimensional, mostrando mais uma vez a robustez do DQN. A figura 4.13 mostra os rewards médios ao longo do treinamento. Na imagem é possível observar que o agente aprende a evitar colisões antes de completar as primeiras 5 *epochs* (250000 frames), devido ao desconto no *reward* ser bem maior que os demais efeitos negativos. E após 60 *epochs*, pode ser dito que agente aprendeu seu caminho até a plataforma de recarga de bateria. Em outras palavras, após esse ponto, quase todas as vezes, ele consegue alcançar a plataforma de forma a evitar colisões no menor tempo possível.

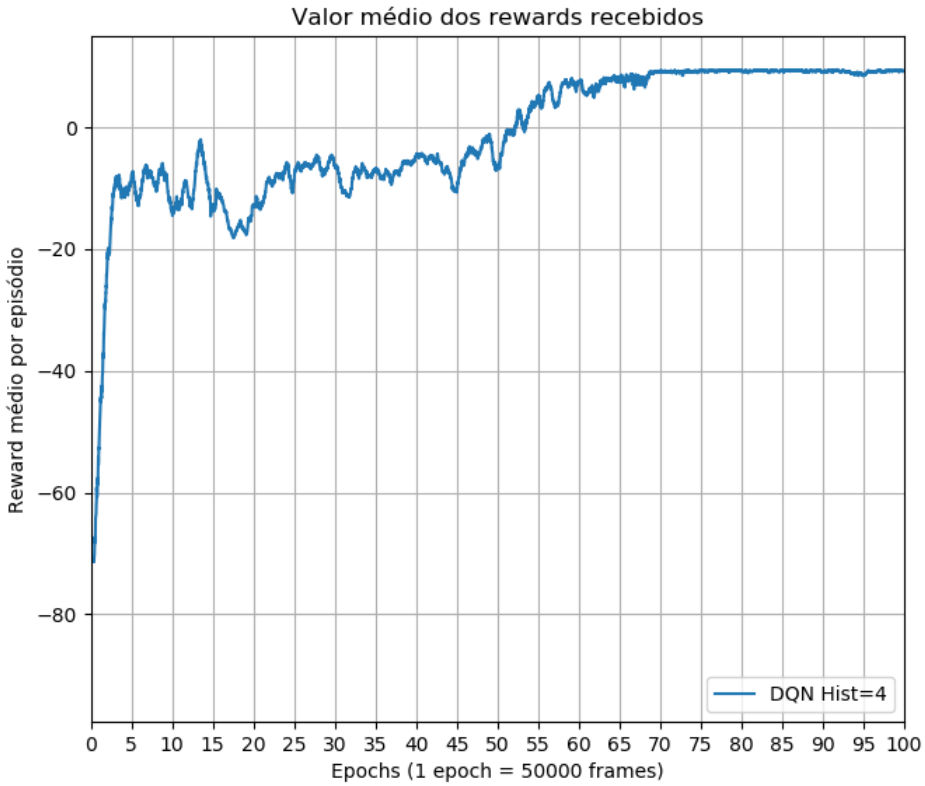

Figura 4.13: Reward médio obtido durante a primeira simulação de aprendizado utilizando o DQN

```
Total params: 1,696,547
Trainable params: 1,695,203
Non-trainable params: 1,344
```

Figura 4.14: Arquitetura da rede neural do DQN com regularização em todas as suas camadas.

Com a primeira simulação já cumprindo o objetivo desse trabalho, foi teorizado se o aprendizado do agente poderia ser feito em um tempo menor. Para isso, primeiramente foi testado se um agente com um tamanho de histórico maior aprenderia mais rápido, logo, foi testada o aprendizado de um agente utilizando o DQN com tamanho de histórico igual a 8, ou seja, volume de entrada com formato de [84, 84, 8].

O segundo teste foi a verificar se os métodos de regularização aplicados a rede neural do agente conseguiriam alguma melhora no aprendizado. Comparando o número total de parâmetros mostrados pela figura 4.14 com o número de parâmetros do DQN padrão mostrados pela figura 4.2, vemos que os métodos de regularização quase não adicionam parâmetros a serem treinados no modelo. Temos 1685667 parâmetros na arquitetura do DQN clássico e 1696547 na arquitetura com regularização, ou seja, 10880 parâmetros a mais na arquitetura com regularização.

O último teste para o aprendizado nesse mapa tridimensional foi utilizar a arquitetura do algoritmo DRQN descrito na sessão 2.3.20 e no qual o número total de parâmetros é mostrado pela figura 4.15. Como pode ser observado, ao se inserir uma rede neural do tipo LSTM no lugar da primeira camada densa, aumentamos mais de 4 vezes o número de parâmetros a serem treinados, o que gera um custo computacional muito maior durante o treinamento.

```
Total params: 7,546,531
Trainable params: 7,546,531
Non-trainable params: 0
```

Figura 4.15: Arquitetura da rede neural do DRQN aplicado a navegação tridimensional.

Após modificações no código, foram simulados o aprendizado do agente usando o DRQN e tamanho de histórico igual a 4, logo, o volume de entrada possui as seguintes dimensões [4, 84, 84, 1]. A figura 4.16 mostra todas as simulações executadas para o aprendizado de navegação em um ambiente tridimensional. É possível observar que aumentar o tamanho do histórico (no gráfico DQN Hist=8) e a inserção dos métodos de regularização (no gráfico DQN Hist=8 Reg) não gerou melhoras significativas na aprendizagem do agente, ambos os agentes aprenderam a chegar no objetivo ao mesmo tempo que o DQN padrão (no gráfico DQN Hist=4). Entretanto, o DRQN mostrado em vermelho consegue convergir muito mais rápido que as demais simulações.

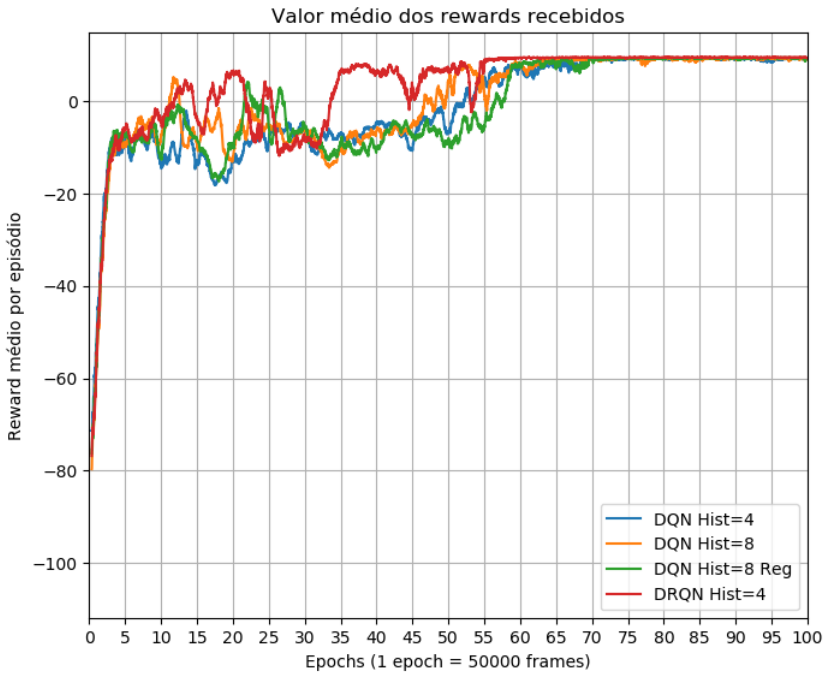

Figura 4.16: DRQN comparado as demais simulação usando DQN.

A figura 4.17 mostra um zoom na região compreendida entre 55 e 85 *epochs*. É possível observar que o DRQN converge próximo a 57 *epochs* os outros algoritmos conseguem convergir ao mesmo valor apenas próximo a 70 *epochs*, logo 650 mil frames após. Além disso o *reward* obtido pelo DQRN oscila bem menos que os demais, sinalizando que o aprendizado do agente foi muito mais robusto, logo o agente falha em encontrar a plataforma de bateria menos vezes que os demais.

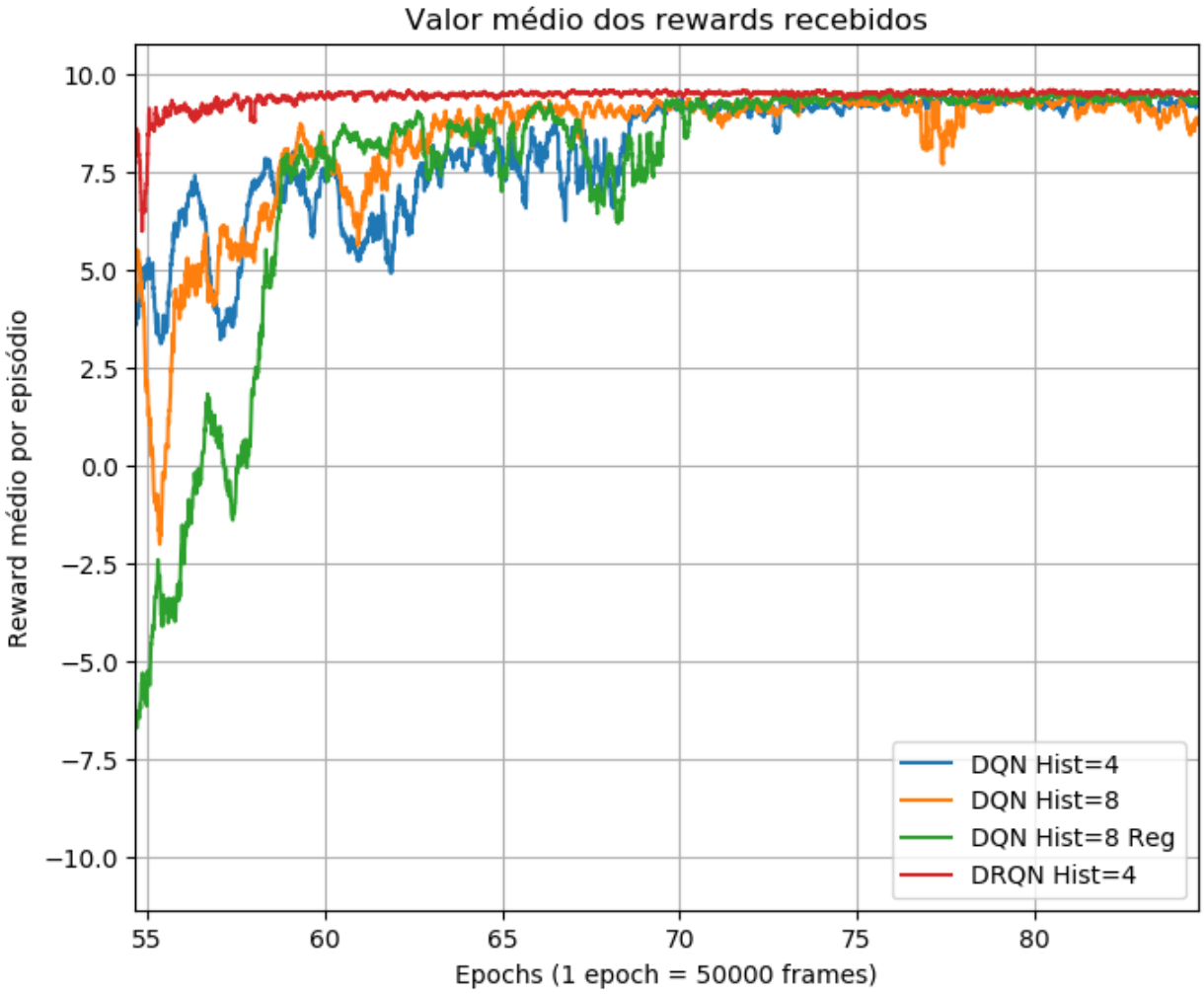

Figura 4.17: Zoom na região de 55 a 85 *epochs* dos rewards médios de todas as simulações

A imagem a esquerda na figura 4.18 mostra o valor médio da *Loss* ao longo do treinamento de cada um dos testes de aprendizado executados. Novamente o DQRN se destaca possuindo uma *loss* muito menor que os demais. Isso significa, que o agente aprende mais rápido quais as ações que geram o maior *reward* ao final do episódio. Já a imagem da direita demonstra os valores médio de Q por episodio para todas as simulações executadas. O agente treinado com o DRQN possui bem menos *overshoot* que os demais.

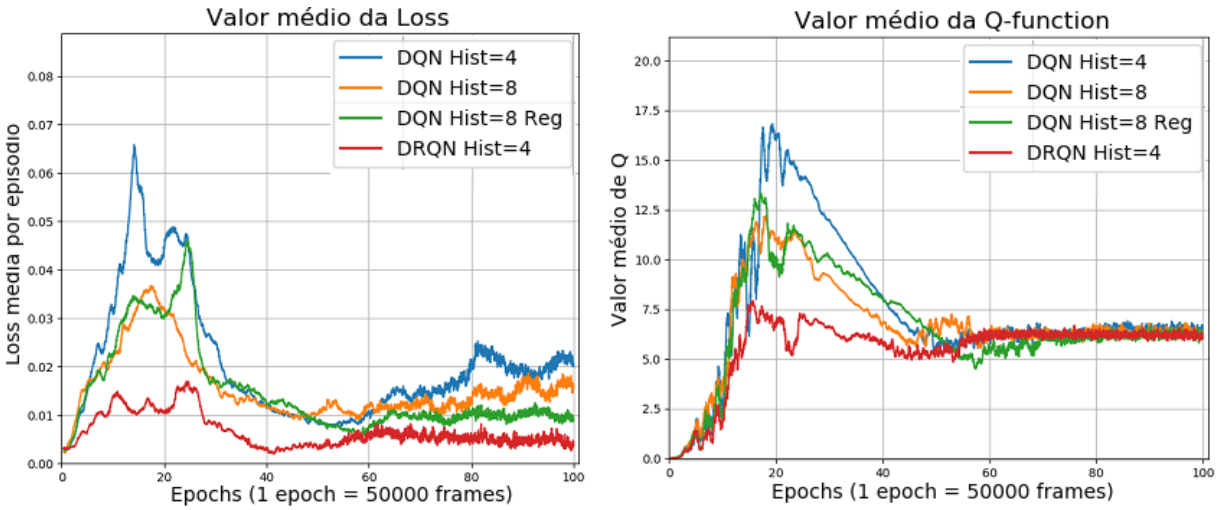

Figura 4.18: Valores médio por episódio da loss para todos os 4 agentes treinados.

4.2.2 Segundo mapa tridimensional

O segundo mapa tridimensional foi criado para testar a capacidade de generalização de aprendizado de agentes que foram ensinados no primeiro mapa. Logo, os agentes foram treinados nesse mapa por apenas 500 mil frames e com hiperparâmetros mais rígidos, como a variável de exploração ϵ fixa desde o começo do aprendizado. A figura 4.19 mostra todos hiperparâmetros fixos durantes as simulações.

Parâmetros fixos	**Valores**
Frame skip (Frames)	4 frames
Discount factor (γ)	0.99
ε	0.05
Decaimento linear	Sem decaimento (ε fixo)
Atualiza Q-target	10000 frames
Tamanho da replay memory	50000 (experiências)
Número de frames no qual o agente toma somente ações aleatórias (preencher a replay memory antes do começo do treinamento)	0
Batch-size	32
Optimizer	Adam
Learning Rate	0.0001
Random seed	1

Figura 4.19: Hiperparâmetros fixos usados durante o aprendizado de navegação no segundo mapa tridimensional.

A imagem 4.20 mostra o *reward* médio por episodio durante o treinamento dos agentes no segundo mapa. O agente DRQN conseguiu achar a plataforma de recarga de bateria desde o começo da simulação pois ele sempre consegue pontuações positivas. Entre os agentes que utilizam o DQN, o que se saiu melhor foi o que usa a rede neural com regularização, isso comprova que a regularização contribui para um aprendizado mais genérico dos agentes. Já os outros dois agentes tiveram uma performance similar.

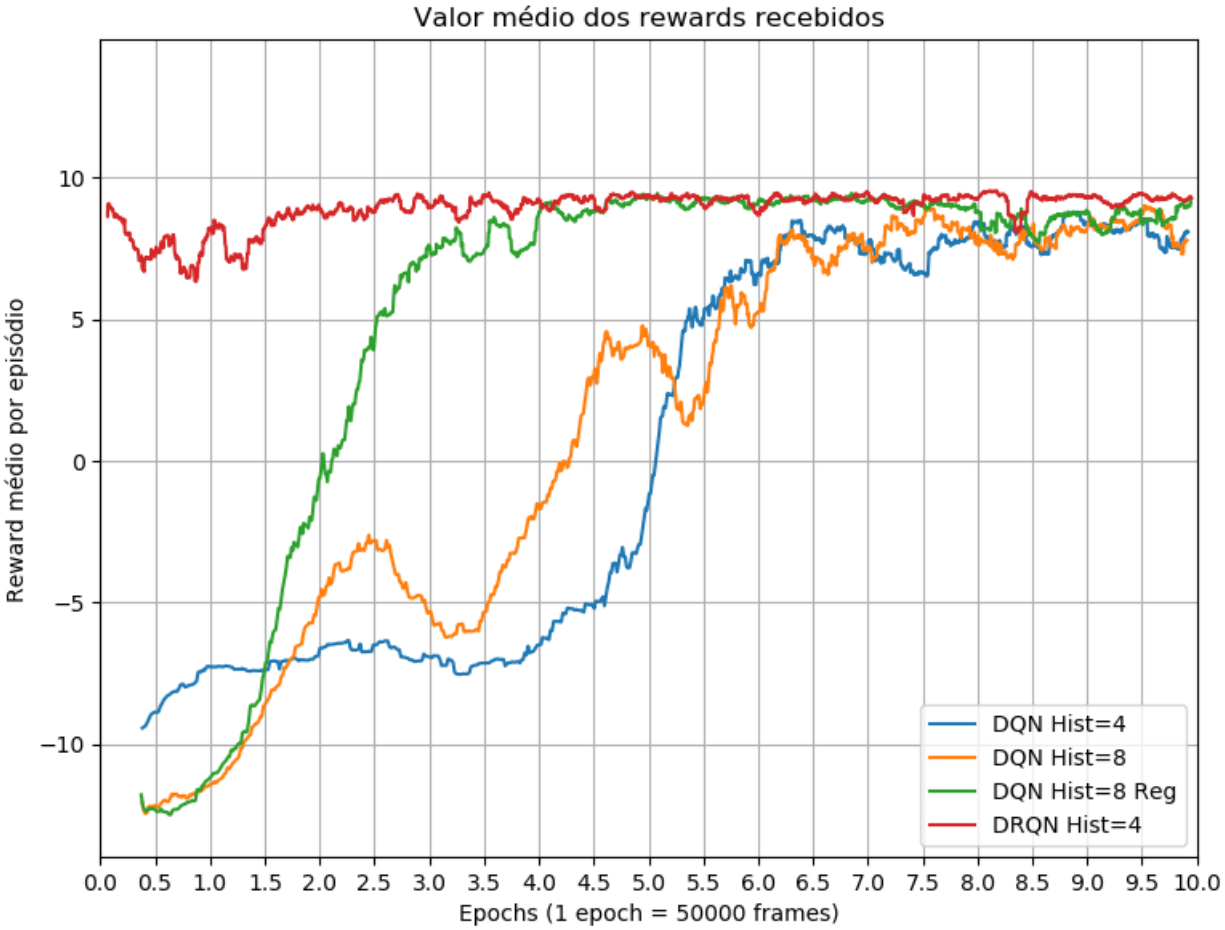

Figura 4.20: Reward médio por episódio de todos os agentes no segundo mapa.

Embora o DRQN tenha se saído melhor nos dois mapas tridimensionais, todos os agentes testados conseguiram navegar em ambos os mapas. Isso comprova que esses algoritmos de *reinforcement learning* possuem uma boa capacidade de generalização de aprendizado. Entretanto, necessitam de muito tempo de aprendizado e um grande poder de processamento.

4.2.3 Visualização do aprendizado das redes neurais

De forma similar ao que foi feito na sessão 4.1.1, exploraremos o que as redes neurais aprenderam durante o treinamento de navegação nos ambientes tridimensionais.

A figura 4.21 mostra as três maiores ativações no volume de entrada dos filtros da terceira camada de convolução em três períodos de tempo durante o treinamento. É possível observar que a rede neural totalmente treinada aprendeu a focar em cada uma das paredes e no canto inferior esquerdo perto ao chão. Para a navegação, faz sentido seguir as divisórias do chão com as paredes, pois em caso um estado seja apenas a visualização de paredes (sem mostrar o chão), isso significa que o agente esta muito próximo a uma parede e há grande probabilidade de colisão caso se mova para frente.

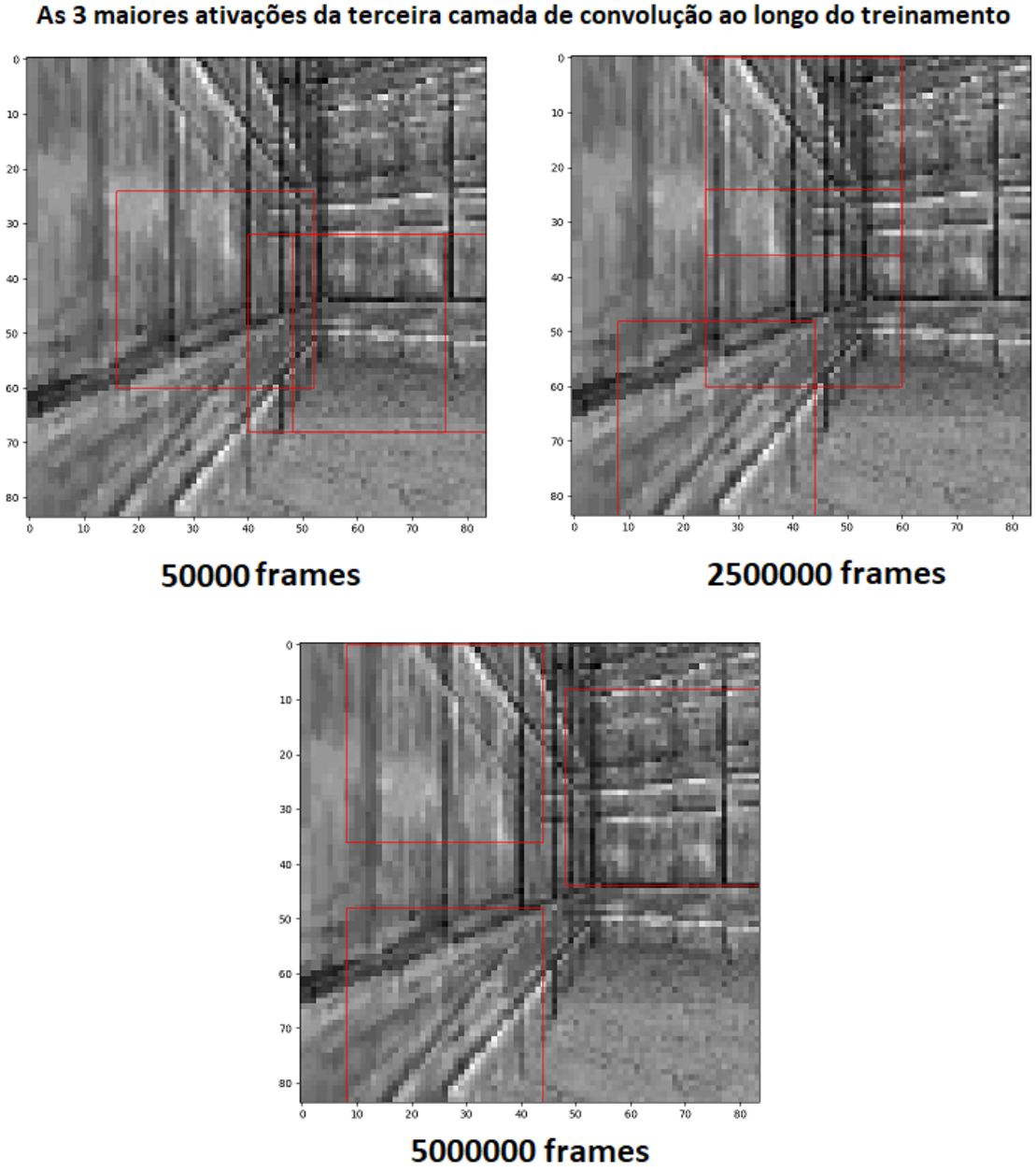

Figura 4.21: Locais do volume de entrada que geram as maiores ativações do filtros da terceira camada de convolução ao longo do treinamento.

A figura 4.22 demonstra os valores de Q para cada ação disponível para o estado avaliado em três períodos de tempo durante o treinamento. O agente treinado somente em 50 mil frames, acredita que a melhor ação a ser tomada nesse estado é virar a esquerda, o que gera grandes chances de uma colisão nos próximos estados. Com 2.5 milhões de frames o agente ainda acredita que virar a esquerda é a melhor solução. Entretanto, em 5 milhões de frames, por uma pouca diferença o agente acredita que mover em frente é a melhor opção (dessa forma a probabilidade de colisão é menor).

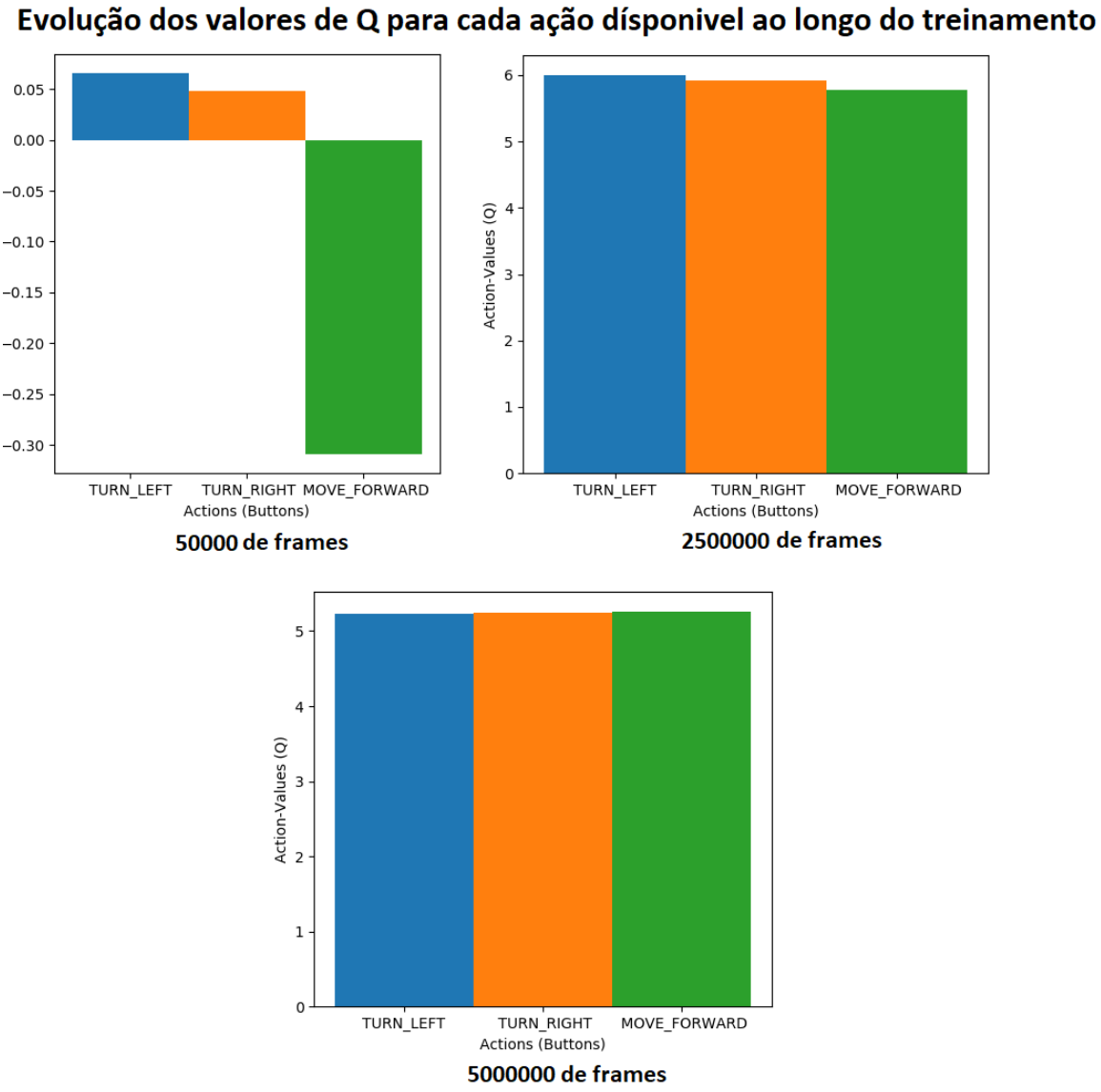

Figura 4.22: Valores de Q para cada ação disponível durante o treinamento de navegação tridimensional

Capítulo 5

Considerações finais e propostas de trabalhos futuros

Neste trabalho, fez-se, inicialmente o desenvolvimento algoritmo DQN, utilizando o ambiente do jogo Pong para testes. Durante o desenvolvimento deste algoritmo foram buscadas as melhores maneiras de aumentar o processamento de frames/segundo. Logo, foi desenvolvido uma abordagem para executá-lo em paralelo ganhando assim em média 35% no tempo de processamento. Entretanto, nesta abordagem é como se tivéssemos introduzido um atraso de uma amostragem no algoritmo. Assim, foram realizados alguns testes para verificar o impacto durante o aprendizado e foi constatado que o aprendizado não foi afetado. Todas as simulações do DQN e DRQN utilizaram a abordagem em paralelo.

Com o ambiente do jogo Pong foram testadas algumas *learning rates* com o método de otimização Adam e foi verificado que os efeitos de uma lr alta acontecem mesmo com métodos de gradiente adaptativo. Além disso, com esse ambiente a rede neural do DQN foi dessecada, mostrando todos *activation maps*, zonas de máxima ativação no volume de entrada e o que cada filtro da NN aprende a procurar.

E, por último, com todos os ensinamentos adquiridos ao longo do trabalho, foram avaliadas diversas arquiteturas de redes neurais para a navegação nos dois mapas tridimensionais criados, rodando na plataforma VizDoom. Todos os agentes testados conseguiram navegar em ambos os mapas. Isso comprova que esses algoritmos de *reinforcement learning* possuem uma boa capacidade de generalização de aprendizado. Entretanto, foi mostrado que a arquitetura do DRQN é muito superior as demais, aprendendo muito mais rápido e de forma mais robusta a navegar por ambos os mapas. Assim, é recomendado, mesmo levando muito mais tempo de treinamento devido à sua enorme quantidade de parâmetros, o uso de arquiteturas que utilizem redes neurais recorrentes para o aprendizado em ambientes parcialmente observáveis.

Como perspectivas de trabalhos futuros, existem uma diversidade de possibilidades.

Em relação aos algoritmos já explorados neste trabalho, é possível a investigação do aprendizado do agente com imagens coloridas e no caso do DRQN, quais são os impactos da inserção de métodos de regularização em suas camadas e ampliação do seu histórico.

Já na parte de desempenho de aprendizado, testar outras variantes mais avançadas do DQN como o Double DQN[36], o Dueling DQN[37], o método de priorização de experiências [38] e também com outros algoritmos de *reinforcement learning* como o A3C[39].

Apêndice A

Códigos

Neste apêndice contém os links para os códigos em python utilizados neste trabalho.

A.1 DQN/DRQN

O código encontra-se para download no link:
https://github.com/Leonardo-Viana/Reinforcement-Learning

Referências

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998. Online Draft <http://incompleteideas.net/book/the-book-2nd.html>. 2018.

[3] Edward L Thorndike. Animal intelligence: an experimental study of the associative processes in animals. *The Psychological Review: Monograph Supplements*, 2(4):i, 1898.

[4] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[6] Shane Legg and Marcus Hutter. Universal intelligence: A definition of machine intelligence. *Minds and Machines*, 17(4):391–444, 2007.

[7] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.

[8] Tomaso Poggio, Hrushikesh Mhaskar, Lorenzo Rosasco, Brando Miranda, and Qianli Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing*, 14(5): 503–519, 2017.

[9] Keith D. Foote. A brief history of deep learning. 2017. Disponível em: <http://www.dataversity.net/brief-history-deep-learning/>. Acesso em: 19 de abril de 2018.

[10] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527*, 2015.

[11] Jakob N Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*, 2016.

[12] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *AAAI*, pages 2140–2146, 2017.

[13] Devendra Singh Chaplot, Emilio Parisotto, and Ruslan Salakhutdinov. Active neural localization. *arXiv preprint arXiv:1801.08214*, 2018.

[14] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[15] Thomas Simonini. Deep reinforcement learning course. 2018, . Disponível em <https://medium.freecodecamp.org/an-introduction-to-reinforcement-learning-4339519de419>. Acesso em: 08 de junho de 2018.

[16] Thomas Simonini. Diving deeper into reinforcement learning with q-learning. 2018, . Disponível em <https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe>. Acesso em: 08 de junho de 2018.

[17] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com/.

[18] Andrej Karpathy. Stanford cs231n: Convolutional neural networks for visual recognition. 2018., . Disponível em: <http://cs231n.github.io/>. Acesso em: 14 de maio de 2018.

[19] Moacir A Ponti and Gabriel B Paranhos da Costa. Como funciona o deep learning.

[20] ML-CheatSheet. Loss functions. 2017. Disponível em: <http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html>. Acesso em: 26 de maio de 2018.

[21] Francois Chollet. *Deep learning with python*. Manning Publications Co., 2017.

[22] Hirsh R Agarwal and Andrew Huang. Tensor-based backpropagation in neural networks with non-sequential input. *arXiv preprint arXiv:1707.04324*, 2017.

[23] Jeremy Jordan. Normalizing your data (specifically, input and batch normalization). 2018. Disponível em: <https://www.jeremyjordan.me/batch-normalization/>. Acesso em: 03 de junho de 2018.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

[25] Sunita Nayak. Batch normalization in deep networks. 2018. Disponível em <https://www.learnopencv.com/batch-normalization-in-deep-networks/>. Acesso em: 07 de novembro de 2018.

[26] Ujjwalkarn. An intuitive explanation of convolutional neural networks. 2016. Disponível em <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>. Acesso em: 08 de junho de 2018.

[27] Machinelearninguru.com. Image filtering: A comprehensive tutorial towards 2d convolution and image filtering. 2018. Disponível em <machinelearninguru.com/computer_vision/basics/convolution/image_convolution_1>. Acesso em: 03 de junho de 2018.

[28] Utkarsh Sinha. Image convolution examples. 2017. Disponível em <http://aishack.in/tutorials/image-convolution-examples/>. Acesso em: 03 de junho de 2018.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[31] Christopher Olah. Understanding lstm networks. 27 de agosto 2015. Disponível em <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Acesso em: 07 de novembro de 2018.

[32] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. 21 de maio 2015, . Disponível em <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>. Acesso em: 07 de novembro de 2018.

[33] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[34] Max Lapan. Speeding up dqn on pytorch: how to solve pong in 30 minutes. 23 de novembro 2017. Disponível em <https://medium.com/mlreview/speeding-up-dqn-on-pytorch-solving-pong-in-30-minutes-81a1bd2dff55>. Acesso em: 07 de novembro de 2018.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

[36] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`.

[37] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL `http://arxiv.org/abs/1511.06581`.

[38] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. URL `http://arxiv.org/abs/1511.05952`.

[39] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL `http://arxiv.org/abs/1602.01783`.