Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Estudo sobre Redes Neurais Recorrentes Rasas e Profundas Aplicadas à Solução de Jogos Digitais – Estudo de Caso: Enduro

Alec Ryo Emura

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2021

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. João José Costa Gondim

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof.ª Dr.ª Carla Denise Castanho — CIC/UnB
Prof. Dr. Marcelo Grandi Mandelli — CIC/UnB

CIP — Catalogação Internacional na Publicação

Emura, Alec Ryo.

Um Estudo sobre Redes Neurais Recorrentes Rasas e Profundas Aplicadas à Solução de Jogos Digitais – Estudo de Caso: Enduro / Alec Ryo Emura. Brasília : UnB, 2021.
75 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2021.

1. Inteligência Artificial, 2. Aprendizagem de máquina,
3. Aprendizagem Supervisionada, 4. Redes Neurais Recorrentes,
5. OpenAI Gym

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Estudo sobre Redes Neurais Recorrentes Rasas e Profundas Aplicadas à Solução de Jogos Digitais – Estudo de Caso: Enduro

Alec Ryo Emura

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof.ª Dr.ª Carla Denise Castanho
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. João José Costa Gondim
Coordenador do Curso de Engenharia da Computação

Brasília, 18 de Novembro de 2021

Dedicatória

Dedico esse trabalho aos meus pais, amigos e professores que me deram suporte durante a graduação.

Agradecimentos

Gostaria de agradecer, primeiramente, aos meus pais que acompanharam por todo esse período me aconselhando e dando suporte. Aos meus amigos de curso, aos meus amigos da república por tornarem meus dias mais divertidos. Ao Prof. Lamar, por sempre ser solícito e responder as perguntas. À Universidade de Brasília que me deu oportunidade cursar a graduação. Aos supervisores dos estágios que me deram *feedback* e pude crescer profissionalmente.

Resumo

Este trabalho é um estudo comparativo sobre a adequação de redes neurais recorrentes rasas e profundas na solução de jogos digitais utilizando apenas imagens, frame a frame da tela, como entrada. São testadas as redes recorrentes rasa de Elman e profunda LSTM, em diversas configurações, inclusive com o uso de camadas convolucionais. Como estudo de caso, foi escolhido o jogo *Enduro*, de 1983 feito para o console ATARI 2600.

O sistema proposto neste trabalho foi testado com quatro tipos de modelos diferentes variando a estruturação das sequências de dados. Os modelos avaliados foram: i) Rede de Elman; ii) Rede LSTM; ii) Autoencoder + LSTM; e iv) Rede convolucional + LSTM.

O melhor desempenho foi obtido pela rede LSTM chegando a média de 127 pontos de recompensa. Apesar da rede com camada convolucional ser ideal para imagens, utilizar apenas uma camada simples deste tipo pareceu não ser o suficiente para execução desta tarefa.

Palavras-chave: Inteligência Artificial, Aprendizagem de máquina, Aprendizagem Supervisionada, Redes Neurais Recorrentes, OpenAI Gym

Abstract

This work is a comparative study on the adequacy of shallow and deep recurrent neural networks in the solution of digital games, having as input only images, frame by frame, of the screen. The networks tested are Elman Network and LSTM with various configurations, including with convolutional layers. The game chosen for study is Enduro, game released in 1983 for ATARI 2600 console. Trained models should be able to get satisfatory score by passing cars avoiding crashing into them.

The system proposed in this work was tested with four different types of models varying the format of the data sequences. The proposed models were: i) Elman Network; ii) LSTM network; ii) Data processed by an autoencoder encoder followed by an LSTM network; and iv) Network with convolutional layer and LSTM layer.

The best performance was obtained by the LSTM network averaging 127 reward points. Although the network with convolutional layer is ideal for images, using only a simple layer of this type did not seem to be enough to perform this task.

Keywords: Artificial Intelligence, Machine Learning, Supervised learning, Recurrent Neural Network, OpenAI Gym

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API *Application Programming Interface.*

Dota2 *Defense of the Ancients 2.*

GPU *Graphics Processing Unit.*

IA Inteligência Artificial.

IBM *International Business Machines.*

LSTM *Long Short-Term Memory.*

ReLU *Rectified Linear Units.*

RNA Rede Neural Artificial.

RNN *Recurrent Neural Network.*

SGD *Stochastic Gradient Descent.*

Capítulo 1

Introdução

A Inteligência Artificial (IA) é uma área na Ciência da Computação responsável por simular as capacidades cognitivas do ser humano por meio de máquinas. Diversas técnicas têm sido propostas e entre elas, uma que está em destaque na atualidade é a Rede Neural Artificial (RNA), uma estrutura baseada no funcionamento do cérebro humano [25].

Na área de IA é comum utilizar jogos eletrônicos para desenvolver e avaliar algoritmos de aprendizado de máquina. Os jogos permitem a validação do cumprimento de regras sem assistência humana e sua eficiência pode ser medida pelas pontuações obtidas nele. O custo de testar em simulações como esta tornam os experimentos mais baratos do que se fosse testado no mundo físico [26].

Desde que Alan Turing escreveu o primeiro programa capaz de jogar Xadrez em 1950 diversos outros estudos surgiram, chegando a ser desenvolvido, em 1997, uma máquina da IBM, chamada Deep Blue, capaz de ganhar do campeão mundial Garry Kasparov [27].

Em 2016, foi desenvolvido pela Deepmind uma inteligência artificial capaz de jogar Go - um jogo que possui uma imensa quantidade de jogadas possíveis - e ganhar do campeão Lee Sedol [28].

Em 2019, a inteligência artificial OpenAI Five ganhou o campeonato mundial de Dota2 contra competidores humanos profissionais. O que surpreende é o fato deste jogo *multi-player* possuir situações imprevisíveis e contínuas, além de depender da cooperação em equipe, análises do ambiente e da necessidade de fazer previsões de jogadas adversárias [29].

Tarefas executadas em jogos podem ser comparáveis a diversas atividades do dia dia. No estudo deste trabalho será testado a capacidade da máquina aprender a jogar um jogo de corrida. Um paralelo possível ao mundo físico seria a automação de carros.

Existem, atualmente, duas empresas com abordagens diferentes liderando a implementação de carros autônomos. A Waymo, utilizando a abordagem Level 4 e a Tesla utilizando a abordagem Level 2. A abordagem Level 2 consiste no ser humano ser fundamentalmente

responsável pela supervisão do sistema de IA e a abordagem Level 4, consiste no sistema de IA ser responsável pelas ações sem o ser humano necessitar supervisiona-lo. A Waymo, em Janeiro de 2020, conseguiu percorrer, aproximadamente, 32 milhões de quilômetros na estrada e 16 bilhões de quilômetros na simulação, já iniciando testes nas estradas com clientes reais sem um condutor humano. A Tesla possui aproximadamente 700.000 sistemas de direção automática em circulação, onde é utilizado redes neurais multitarefas para perceber, predizer e agir [30].

A implementação destes sistemas é complexo, exigindo a especialização de diversas áreas, sendo algo desafiador combinar todas elas em uma única rede neural. No artigo [31] foi feita uma pesquisa do estado da arte da maioria destas áreas: a percepção, o mapeamento e localização, a predição, o planejamento e controle, a simulação, o V2X e segurança, entre outros.

Em 2020 foi publicado uma pesquisa [32] na qual se experimentou treinar uma rede neural a dirigir carros utilizando o conceito de clonagem de comportamento. O jogador humano gerava imagens a partir de uma simulação e, com elas, treinava uma rede neural. Esta rede consistia em cinco camadas convolucionais e três camadas densas. O modelo proposto teve um desempenho melhor que outros modelos de aprendizado profundo, apontando que o trabalho apresentado pode ser realizado no mundo real para construir veículos capazes de dirigir de forma autônoma. Como trabalhos futuros é proposto a adição de dados de treino com trajetos reais em várias situações e climas a fim de tornar o seu sistema mais robusto.

A fim de estudar as estruturas com aptidões para treinar um carro a dirigir, foi utilizado as redes neurais recorrentes. Estas redes são capazes de aprender e utilizar a noção de eventos sequenciais.

Um ambiente que proporciona isso é o jogo *Enduro* (Seção 2.6). Este jogo foi escolhido por ser: simples; possuir boa representação de uma corrida de carros; e existir outros trabalhos que utilizam este jogo como estudo, como o Muzero [24], facilitando a comparação das performances entre os algoritmos.

1.1 Motivação

No dia 17 de Junho de 2018 a Organização Mundial da Saúde publicou um relatório chamado *Global status report on road safety 2018* [33]. Nele é apresentado que as mortes nas estradas continuam aumentando em todo o mundo e mais de 1,35 milhão de pessoas perdem a vida todos os anos em decorrência de acidentes de trânsito, o que significa que, em média, morre uma pessoa a cada 24 segundos. Além destes, entre 20 e 50 milhões sofrem lesões não fatais, com muitas ficando incapacitadas como resultado de suas lesões.

Também é apontado que o alcoolismo, o excesso de velocidade e a distração ao volante são as principais causas desses acidentes.

Uma das soluções a esta questão seria a aplicação de inteligência artificial em veículos, trocando os motoristas humanos pelas máquinas condutoras. A intenção deste trabalho é estudar a aplicação de redes neurais recorrentes a este tipo de tarefa utilizando o jogo Enduro (Seção 2.6).

1.2 Hipótese de pesquisa

É possível treinar uma Rede Neural Recorrente RNN para jogar o jogo para Atari Enduro (Seção 2.6), usando apenas visão computacional através do ambiente OpenIA Gym.

1.3 Objetivos

O objetivo geral deste projeto é fazer um estudo comparativo sobre a adequação de Redes Neurais Recorrentes RNN rasas e profundas para aprender a jogar Enduro (Seção 2.6) e verificar a eficiência de seu uso na tarefa observando as pontuações obtidas.

Como objetivos específicos temos:

- Gerar banco de dados de frames
- Definir o pré processamento
- Treinar utilizando rede Elman
- Treinar utilizando rede LSTM
- Aplicação de melhorias utilizando outras redes

1.4 Apresentação do manuscrito

Este manuscrito está organizado em cinco capítulos: No Capítulo 2 é feita uma revisão dos conceitos utilizados para realização deste projeto. No Capítulo 3 serão apresentadas propostas para resolução do problema, explicitando as ferramentas e argumentos para tomadas de decisão. No Capítulo 4 serão expostos os resultados do projeto. No Capítulo 5 conterão a conclusão e as futuras melhorias possíveis de serem feitas.

Capítulo 2

Referencial Teórico

Neste capítulo é descrito o funcionamento das redes neurais, Redes Neurais Recorrentes (Rede Elman e Rede LSTM), Autoencoder, Redes Convolucionais, ferramentas e materiais utilizados.

2.1 Inteligência Artificial (IA)

A Inteligência Artificial (IA) é um ramo da ciência da computação que se propõe na elaboração de técnicas que permitem as máquinas simularem a capacidade humana de raciocínio e solução de problemas [2]. O desenvolvimento nesta área tem sido impulsionada pela rápida evolução do poder computacional e na maior quantidade de dados disponíveis.

2.2 Rede Neural Artificial (RNA)

Fazendo parte do ramo da inteligência artificial, as redes neurais simulam a arquitetura do cérebro humano, mais precisamente na transmissão de sinais dos neurônios biológicos [25]. Uma RNA é composta por vários neurônios que se conectam entre si, como mostrado na Figura 2.1. Essas conexões possuem um peso ω e todos os valores que passam por essas conexões são multiplicados pelo respectivo ω. Os valores são recebidos pelo próximo neurônio e somados. A estrutura do neurônio pode ser vista na Figura 2.2 [2] [34].

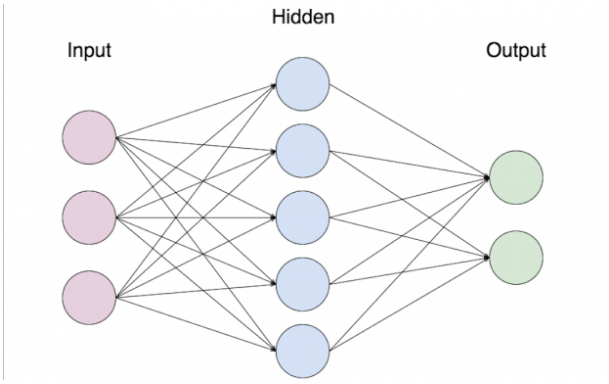


Figura 2.1: Estrutura de uma Rede Neural com 3 entradas, cinco neurônios na camada escondida e 2 neurônio de saída. Fonte: [1].

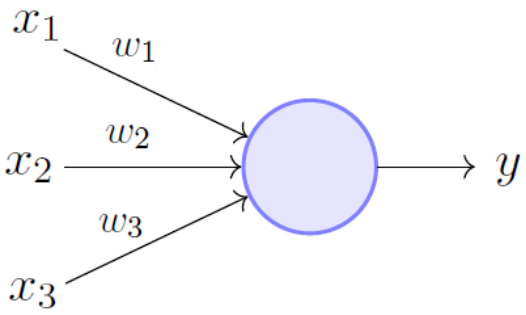


Figura 2.2: Modelo de um neurônio. Fonte: [2].

O treinamento consiste em adaptar os parâmetros peso ω e viés b (*bias* em inglês) dos neurônios a fim de maximizar o acerto. A função de ativação φ, é aplicada na saída do neurônio para determinar o intervalo de valor de saída e possibilitar o ajuste da rede de forma mais precisa. A Equação 2.1 explicita esse funcionamento, sendo então φ a função de ativação, ω_j os pesos da camada j, x os valores das entradas e b os viéses [35].

$$y_j = \varphi(\omega_j \cdot x + b) \tag{2.1}$$

É listado alguns exemplos de função de ativação:

1. Função Linear é definida na Equação 2.2, onde m é uma constante.

$$R(x) = mx \tag{2.2}$$

2. Função Sigmoide definida na Equação 2.3, limitada entre 0 e 1. Muito útil para classificação. Sua desvantagem é que, quando o gradiente estiver nas extremidades, pequenas alterações praticamente não serão sentidas.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

3. Função tangente Hiperbólica definida na Equação 2.4, muito semelhante à função sigmoide, porém, varia entre -1 e 1. Possui as vantagens e desvantagens da sigmoide.

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.4}$$

4. Função Unidade Linear Retificada *Rectified Linear Units* (ReLU) é definida na Equação 2.5,

$$ReLU(x) = \begin{cases} x, & \text{se } x > 0 \\ 0, & \text{caso contrário} \end{cases} \tag{2.5}$$

Para a saída, podemos ainda utilizar a camada *softmax*, dada pela Equação 2.6:

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.6}$$

Essa função calcula a probabilidade de cada classe i dentro de todas as n possíveis. É muito utilizada para classificação.

A Figura 2.3 mostra o gráfico das funções de ativações apresentadas.

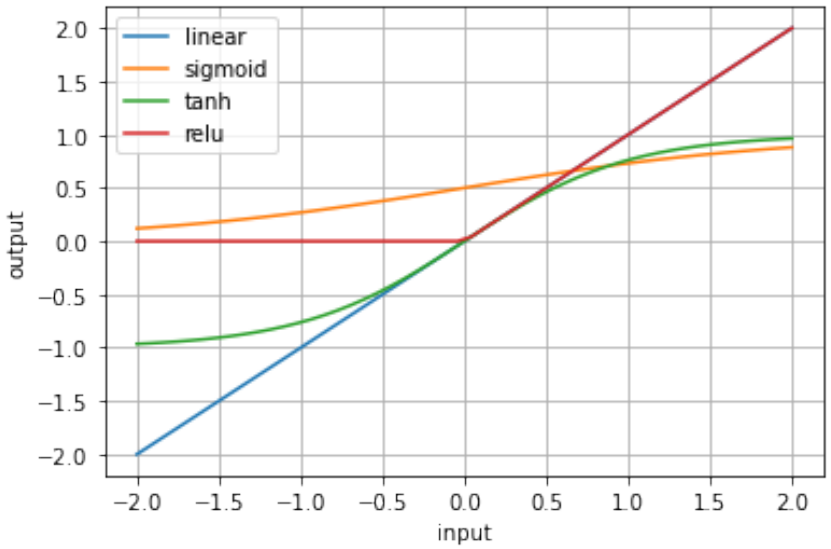


Figura 2.3: Funções de ativação: linear, sigmoide, tanh, ReLU.

2.2.1 Função de custo/perda

A função de custo ou função de perda é utilizada para verificar a qualidade da saída das redes, comparando o resultado obtido com o esperado. Uma função de custo muito comum é o do erro médio quadrático conforme Equação 2.7, onde em cada iteração n é subtraído a saída desejada $d_j(n)$ e o resultado obtido $y_j(n)$ nos neurônios j [36].

$$C_j(n) = \frac{1}{2}(d_j(n) - y_j(n))^2 \tag{2.7}$$

É possível reescreve-la para obter a função de custo total da rede, mostrando a contribuição dos N exemplos passados conforme a Equação 2.8.

$$C(N) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in S}(d_j(n) - y_j(n))^2 \tag{2.8}$$

Em uma rede neural, os resultados são apresentados na última camada correspondente a camada de saída, assim, se consideram somente os neurônios j da camada de saída [1].

Deste modo, quanto menor o valor da função de custo, mais ou resultados se aproximam da saída esperada.

Esses valores da função de custo, gerado a cada entrada de dados, pode ser usado para desenhar um gráfico de treinamento. A melhor rede é a que possui menor função de custo. Ou seja, aquela que apresenta o menor erro. É importante lembrar que cada ponto deste gráfico é gerado durante o treinamento, assim, não é possível o algoritmo ter certeza se

está no mínimo possível. Estes mínimos são representadas como a posição mais ao fundo de um "vale" da função apresentada na Figura 2.4.

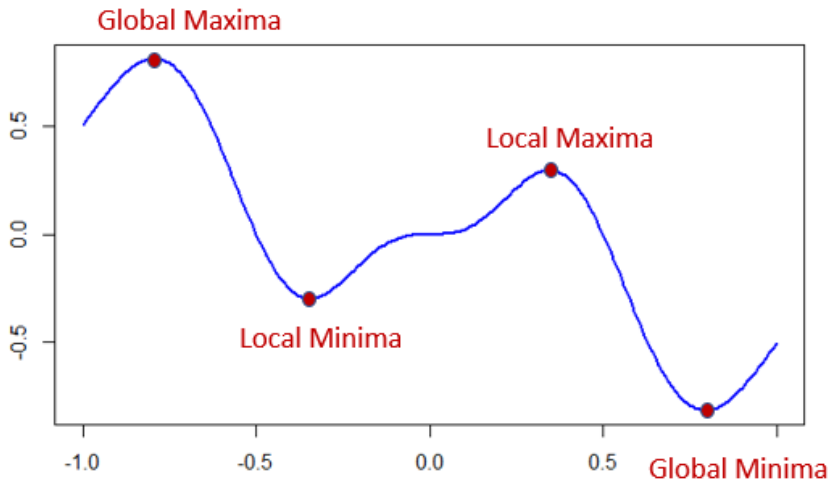


Figura 2.4: Exemplo ilustrativo dos mínimos e máximos da função. Fonte: [3].

Este ponto mais fundo possível do gráfico é o mínimo global. No entanto, geralmente existem outros pontos que devem ser evitados, os mínimos locais. O mínimo local é o ponto onde a rede, durante o treinamento, pode acreditar estar no menor ponto possível por estar num "vale", porém ainda existe um "vale" mais profundo.

2.2.2 Treinamento de redes neurais

Nesta seção será descrito a técnica *backpropagation* e os seus otimizadores *Stochastic Gradient Descent* (SGD) [37] e Adam [7].

Backpropagation

Este método é uma implementação da regra da cadeia composta por duas fases: o *forward pass* e *backward pass*. Na primeira fase *forward pass* são apresentados os dados na entrada da rede, com pesos fixados, passando por toda a rede e obtendo as previsões na saída.

Na segunda fase *backward pass* a saída da etapa anterior é comparada com a saída desejada, e então propaga-se o gradiente da função de custo da camada de saída até a camada de entrada atualizando todos os pesos.

Com os gradientes calculados para cada peso com a respectiva função de custo, a correção Δw_{ij} a ser aplicada no peso w_{ij} da camada i e neurônio j é definida na Equação 2.9, sendo η o parâmetro da taxa de aprendizado da rede, comumente referido em inglês como *learning rate* [1].

$$\Delta w_{ij} = -\eta \frac{\partial C}{\partial w_{ij}} \tag{2.9}$$

Então cada peso é atualizado conforme

$$w_t = w_{t-1} - \eta \frac{\partial C}{\partial w_{ij}} \tag{2.10}$$

Como mostrado na Figura 2.5, o erro das camadas anteriores é calculado utilizando a derivada da função de custo (*loss function*) e regra da cadeira.

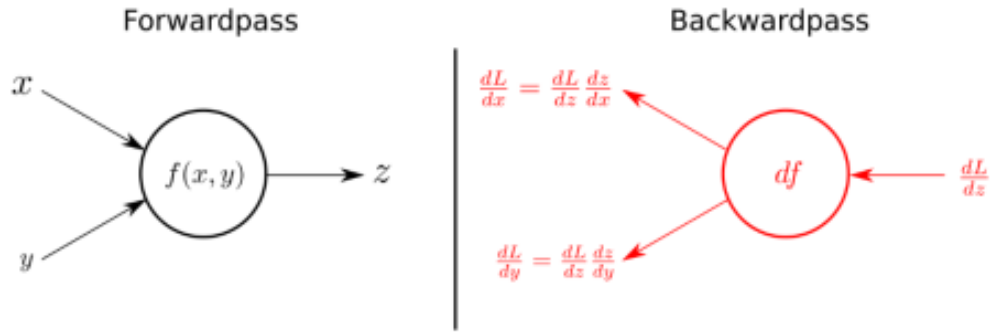


Figura 2.5: Resumo do funcionamento do *forward pass* e *backward pass*. Fonte: [4].

Momentum

A componente Momentum pode ser acrescida ao *backpropagation* para acelerar o treinamento e impedir que caia em mínimos locais. Com sua adição parte de seu peso anterior é mantido, e deste modo, mantêm parte de seu vetor anterior. A Equação 2.11 mostra a função de atualização dos pesos do *backpropagation* acrescido com a componente Momentum, conservando uma parte μ do peso anterior. Na Figura 2.6 é ilustrado um exemplo comparativo do algoritmo sem e com a componente adicionada.

$$w_t = w_{t-1} - \eta \frac{\partial C}{\partial w_{ij}} + \mu \cdot w_{t-1} \tag{2.11}$$

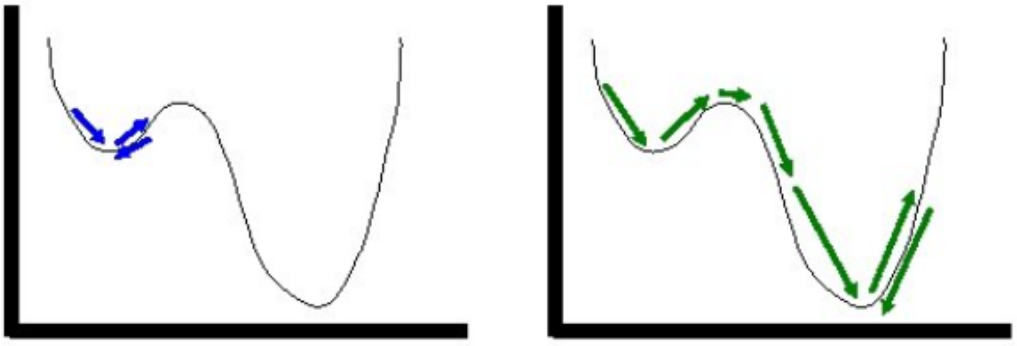


Figura 2.6: Exemplo comparativo considerando o uso de momentum. A esquerda um gráfico sem a utilização do momentum e a direita com a sua utilização. Fonte: [5].

Stochastic Gradient Descent

Com o aumento no número de dados apresentados na entrada da rede, se começou a apresentar problemas na utilização do método do gradiente.

O método do gradiente usa todos os dados de treinamento para computar a próxima atualização dos parâmetros em cada iteração no objetivo de convergir ao menor valor. Entretanto na prática, calcular o custo e o gradiente de todo o conjunto de treinamento pode ser muito lento ou impraticável em uma única máquina se o conjunto de dados de treino for muito grande para caber na memória. Outro problema é a dificuldade de introduzir novos dados durante a execução do treinamento. Esses problemas são contornados pelo *Stochastic Gradient Descent* (SGD), um algoritmo que aleatoriza as componentes da função de custo. O SGD segue o gradiente negativo da função depois de ver apenas um dado ou um lote contendo alguns dados - *minibatch* [37] [38].

O seu código [39] é demonstrado no Algoritmo 1, considerando ΔQ_i a variação do peso da operação *backpropagation* na iteração i:

Algoritmo 1 *Algoritmo do SGD*

1: Escolher um vetor inicial de parâmetro ω e learning rate η
2: **while** Repetir até atingir o mínimo apropriado **do**
3: Embaralhe aleatoriamente os dados de treino
4: **while** i $<$ n **do**
5: $\omega := \omega - \eta \Delta Q_i(\omega)$
6: **end while**
7: **end while**

A Figura 2.7 ilustra a convergência utilizando SGD de forma visual.

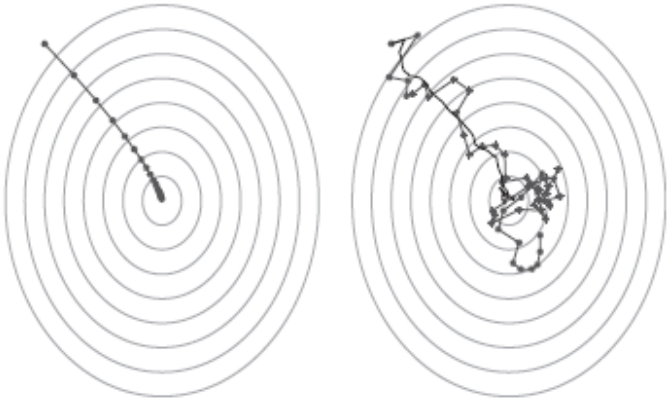


Figura 2.7: Exemplo de convergência utilizando *Gradient Descendent* a esquerda e *Stochastic Gradient Descent (SGD) a direita.* Fonte: [6].

Adam

É uma variação do *Stochastic Gradient Descent* (SGD), recomendada como algoritmo padrão para treinamento de redes neurais [40]. Este tem mostrado grande performance em comparação a outros algoritmos variantes do gradiente descendente em um grande número de problemas. Seu nome é derivado de *Adaptive Moment estimation* por usar estimativas de primeiro e segundo momento do gradiente para realizar atualizações [40].

O algoritmo de Adam primeiro computa o gradiente g_t da função de custo em relação aos parâmetros θ, em seguida, computa e armazena o primeiro e o segundo momentos do gradiente, m_t e v_t respectivamente, como na Equação 2.12 e Equação 2.13 [40],

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{2.12}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{2.13}$$

onde β_1 e β_2 são hiper-parâmetros que estão no intervalo entre zero e um. Esses parâmetros podem ser vistos como taxas de decaimento exponencial dos momentos estimados, pois o valor anterior é multiplicado sucessivamente pelo valor menor que 1 em cada iteração. Os autores do artigo original sugerem valores $\beta_1 = 0{,}9$ e $\beta_2 = 0{,}999$. Na notação atual, a primeira iteração do algoritmo está em $t = 1$ e ambos, m_0 e v_0 são inicializados em zero. Uma vez que ambos os momentos são inicializados em zero, nas etapas iniciais de tempo, esses valores são enviesados em direção a zero. Para contrariar isso, os autores propuseram uma atualização corrigida para m_t e v_t como nas Equação 2.14 e Equação 2.15 [40].

$$\hat{m} = \frac{m_t}{(1 - \beta_1^t)} \tag{2.14}$$

$$\hat{v} = \frac{v_t}{(1 - \beta_2^t)} \tag{2.15}$$

Por fim, a atualização do parâmetro é computada como na Equação 2.16

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.16}$$

onde ϵ é uma pequena constante para estabilidade. Os autores recomendam o valor de $\epsilon = 10^{-8}$ [40]

Na Figura 2.8 ilustra a implementação do código descrito no artigo que apresenta o algoritmo Adam [7]

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figura 2.8: Algoritmo Adam. Fonte: [7].

2.2.3 Underfit e Overfit

Em redes neurais, deve-se achar um equilíbrio entre a falta de treinamento e o seu exagero. Quando a rede não consegue se especializar numa tarefa a rede está sofrendo de *Underfitting*. Já quando a rede fica especializada demais para aquele conjunto dados, sem generalizar o problema, a rede sofre de *Overfitting*. Neste caso, quando tenta-se predizer um novo dado, a rede fica confusa e não prediz corretamente. Na Figura 2.9 é ilustrado um caso hipotético de resultados de três modelos, sendo os pontos a resposta desejada

e a linha tracejada a predição dos modelos. O modelo do gráfico mais a esquerda sofre de *underfitting*, a do meio representa uma rede bem treinada, conseguindo generalizar o problema, e a da direita uma rede sofrendo de *overfitting*.

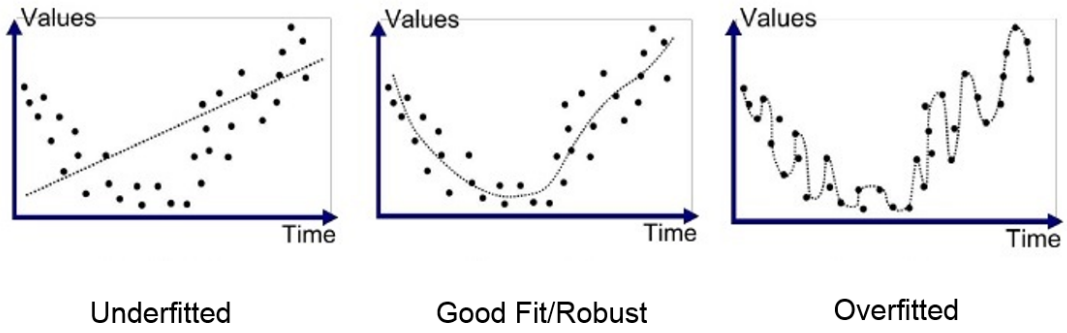


Figura 2.9: Exemplo comparativo de resultado de três modelos. Fonte: [8].

2.2.4 Codificação One Hot

É uma codificação de dados composta por 0 e 1, em que cada dado possui sua posição numa sequência de zeros e é preenchido com um na posição correspondente ao dado [41]. Por exemplo, se tiver como opções de saída de uma rede os valores: azul, vermelho e verde, na codificação one hot ficaria: azul = 100, vermelho = 010, verde = 001.

2.3 Redes Neurais Recorrentes

Derivadas de redes neurais *feedforward*, as Redes Neurais Recorrentes RNN podem usar seu estado interno (memória) para processar, como entradas, sequências de comprimentos variáveis [42].

Neste tipo de rede os dados anteriormente passados a rede são considerados e possuem relevância na predição. Por exemplo, no processamento de textos, onde não se pode concluir o contexto analisando apenas uma palavra ou letra (“banana” ao invés de “caí numa casca de banana”).

2.3.1 Rede de Elman

Publicado por Jeffrey L. Elman em seu artigo [9], a rede de Elman propõe uma solução aos problemas de predição em dados relacionados a tempo.

Este tipo de rede é constituída de três camadas mais um conjunto de unidades de contexto [42]:

- camada de entrada
- camada escondida
- camada de saída
- unidades de contexto

As três primeiras camadas funcionam de maneira igual a uma rede neural feedforward. O que as diferenciam são as unidades de contexto que funcionam como uma memória do estado anterior da camada escondida [42].

Sua estrutura pode ser observada na Figura 2.10. A entrada da rede é alimentada com os dados atuais, processadas na camada escondida, salvas na camada de contexto e o resultado entregue na camada de saída.

Quando um novo dado é alimentado em sequência, os dados entram pela camada de entrada da rede, as camadas escondidas recebem os dados mais os dados das unidades de contexto, e após o seu processamento, é entregue o novo resultado.

Assim, os resultados anteriores conseguem impactar nos resultados mais recentes.

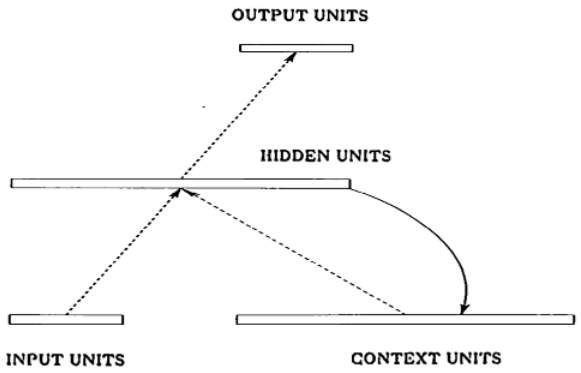


Figura 2.10: Esboço da estrutura de Elman. Fonte: [9].

As Equações 2.17 e 2.18 apresentam os cálculos da saída da camada escondida e da camada de saída respectivamente.

$$h_t = \varphi_h(W_h x_t + U_h h_{t-1} + b_h) \tag{2.17}$$

$$y_t = \varphi_y(W_y h_t + b_y) \tag{2.18}$$

sendo x_t os dados de entrada, h_t os valores da camada escondida, y_t a saída, W, U e b os parâmetros matriciais de pesos e o vetor de bias, φ_h e φ_y as funções de ativação [42].

2.3.2 Rede LSTM

A rede *Long Short-Term Memory* (LSTM) foi proposta com a intenção de superar o *vanishing gradient problem* [43]. Este problema ocorre porque, ao calcular os erros no processo de *backpropagation* para calibrar os pesos da rede, usam-se derivadas para encontrar o quanto houve de mudança em relação a camadas anterior.

Supondo que se deseja treinar uma rede feedforward com diversas camadas e, entre elas, uma camada com função de ativação sigmoide. De acordo com o algoritmo *backpropagation* a derivada da sigmoide fica entre zero e um. Propagando esse valor através de várias camadas, quando chega nas primeiras, o seu valor fica muito pequeno e quase não afeta nas atualizações. Se ao invés de utilizar a função de ativação sigmoide fosse usada a ReLU também se teria um problema. O seu derivativo seria maior que um, e quando esse valor chegasse nas primeiras camadas o seu valor seria muito grande, configurando no *exploding gradient problem*.

Trabalhando com as redes neurais recorrentes ocorre esse mesmo problema. A cada instante de tempo é necessário atualizar a camada de tempo anterior. Com o passar do tempo, a rede tenderá a sofrer o fenômeno do *vanish gradient problem* "esquecendo" sobre os primeiros dados de entrada.

A Figura 2.11 ilustra o *forward pass* e o *backpropagation* no treinamento de uma rede. O *vanishing gradient problem* é encontrado conforme o t aumenta, ficando cada vez mais difícil atualizar as redes de eventos muito passados.

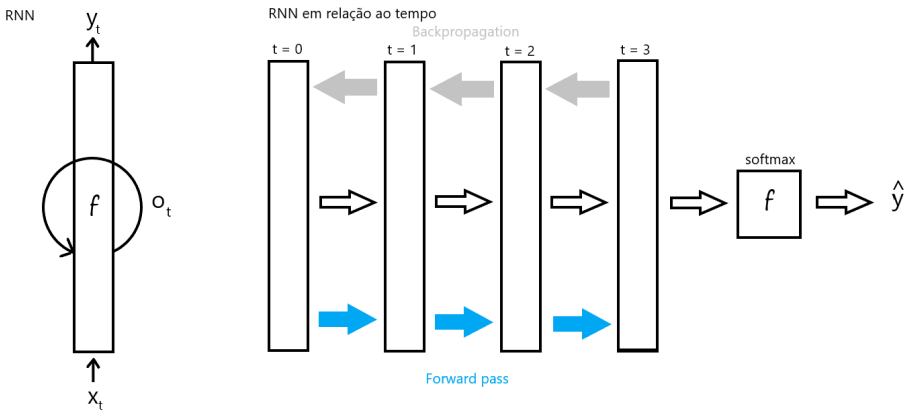


Figura 2.11: Treinamento de uma rede recorrente desmembrada em relação ao tempo.

A fim de solucionar este problema foi proposto a Rede LSTM [43], consistindo em substituir os neurônios tradicionais da camada escondida pelas células LSTM.

Uma célula LSTM comum é composta de uma porta de entrada, uma porta de saída e uma porta de esquecimento. A célula armazena valores sobre um intervalo de tempo arbitrário e as três portas regulam o fluxo de informação dentro da célula [43].

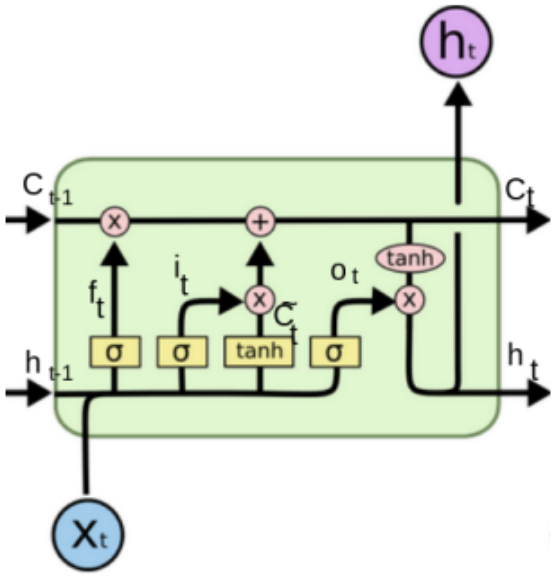


Figura 2.12: Estrutura de um neurônio. Fonte: [10].

A estrutura de uma rede neural composta por células LSTM é apresentada na Figura 2.13.

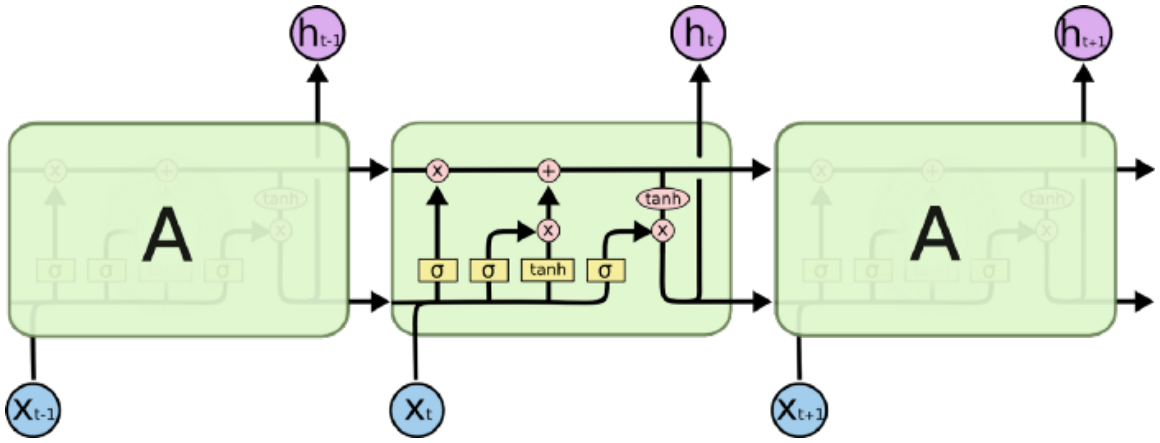


Figura 2.13: Estrutura de uma Rede Neural com duas entradas, cinco neurônios na camada escondida e 1 neurônio de saída. Fonte: [11].

Inicialmente, a LSTM deve calcular qual informação não considerar na célula, o que ela faz através do portão de esquecimento formado pela sigmoide (Equação 2.19). Analisa-se

h_{t-1} e x_t e gera uma saída, no intervalo de zero a um, para cada número do estado C_{t-1} representado o quanto manter de informação.

Em seguida, calcula-se qual nova informação vai guardar, fazendo dois processos: i) o portão de entrada, formado pela camada sigmoide i_t, decide qual valor será atualizado ii) a função de ativação *tanh* cria um vetor de novos candidatos $\widetilde{C}_t$ (Equação 2.20), que pode ser adicionado ao estado.

Depois disso, multiplica-se o estado antigo de f_t por C_{t-1}, para que sejam esquecidos as informações que se julgou desnecessárias, e multiplica-se i_t por $\widetilde{C}_{t-1}$, para manter as informações novas que são úteis (Equação 2.21).

Em seguida, soma-se o resultado dessas duas multiplicações.

Decide-se a saída, para isso, a camada sigmoide (Equação 2.22) decide que partes do estado da célula irão para saída, então multiplica-se isso pela *tanh* do estado da célula C_t, para que saia apenas as informações que a rede aprendeu que seriam importantes (Equação 2.23) [10].

$$f_t = \theta_t(W_f[h_{t-1}, x_t] + b_i) \tag{2.19}$$

$$\widetilde{C}_t = tanh(W_c[h_{t-1}, x_t] + b_c) \tag{2.20}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t \tag{2.21}$$

$$o_t = \theta(W_o[h_{t-1}, x_t] + b_o) \tag{2.22}$$

$$h_t = o_t \odot tanh(C_t) \tag{2.23}$$

2.4 Rede Convolucional

As redes neurais convolucionais são estruturas muito conhecidas na área de processamento de imagens. Foi inspirada nos mecanismos de percepção visual dos seres vivos observada por Hubel & Wiesel [44]. Eles encontraram células no córtex visual dos animais responsáveis por detectar luz nas regiões de percepção (*Receptive Fields*) [45].

Em sua estrutura é composta por uma camada de entrada, uma camada escondida e uma camada de saída. Neste tipo de rede, na camada intermediária, é incluso uma camada que faz operação de convolução [46].

Normalmente se tem uma camada que realiza um produto escalar do filtro da convolução com uma matriz de entrada.

O filtro da convolução desliza sobre a matriz de entrada para a camada gerar uma máscara de características (*Feature Map*) que alimenta a próxima camada. Esta pode ser seguida de outras camadas como a *pooling layer*, camada densa e camada de normalização.

2.4.1 Camada Convolucional

A camada convolucional faz a operação de convolução nos dados de entrada e passa o resultado para a próxima camada. Os parâmetros desta camada consiste no conjunto de filtros adaptáveis. A operação de convolução consiste em deslizar um filtro sobre a imagem e extrair informações sobre a área analisada. Esses filtros devem ser menores que o tamanho da entrada. Cada filtro desliza sobre a imagem e constrói um mapa de características em duas dimensões. Esse mapa de características é passado para a próxima camada. Na Figura 2.14 é ilustrado a construção deste mapa. Um filtro 5×5 mapeia o resultado a uma posição. Este filtro é deslocado e extraido informações sobre a área, esta área é nomeada *Receptive Fields*. A entrada dos dados deve ter o formato (número de imagens $\times$ número de canais $\times$ altura $\times$ largura) [46] [47].

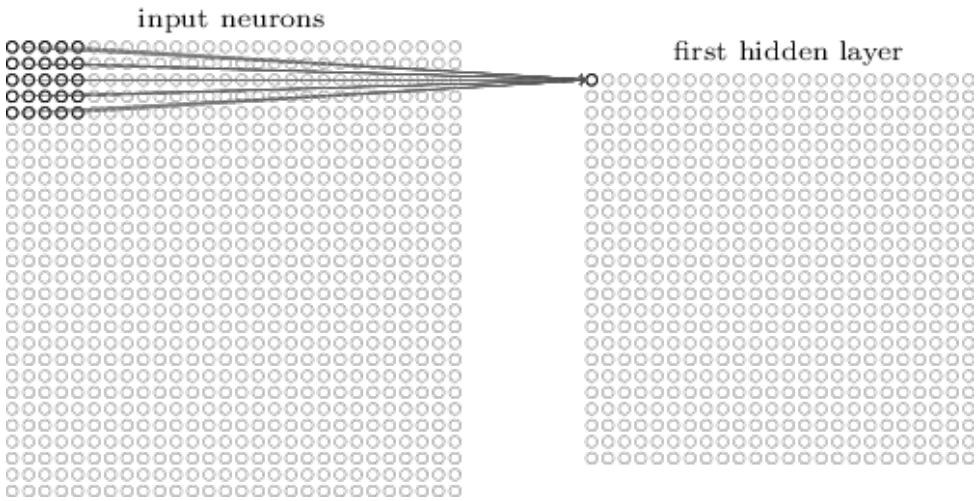


Figura 2.14: Mapeamento da convolução. Fonte: [12].

O filtro/kernel é formado por pesos iniciados aleatoriamente, atualizando-os a cada nova entrada.

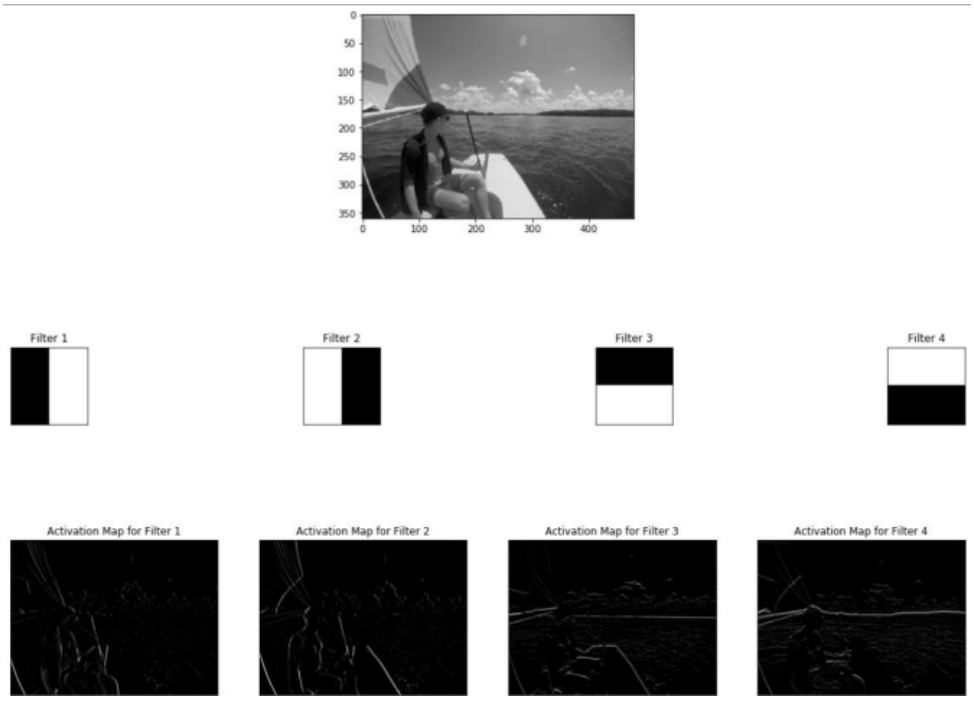


Figura 2.15: Exemplos de mapa de ativação gerados usando quatro filtros. Fonte: [13].

O *stride* é um hiper-parâmetro que representa de quanto o filtro deslizará pela imagem durante a análise. A Figura 2.16 ilustra casos com *stride* igual a 1 e 2.

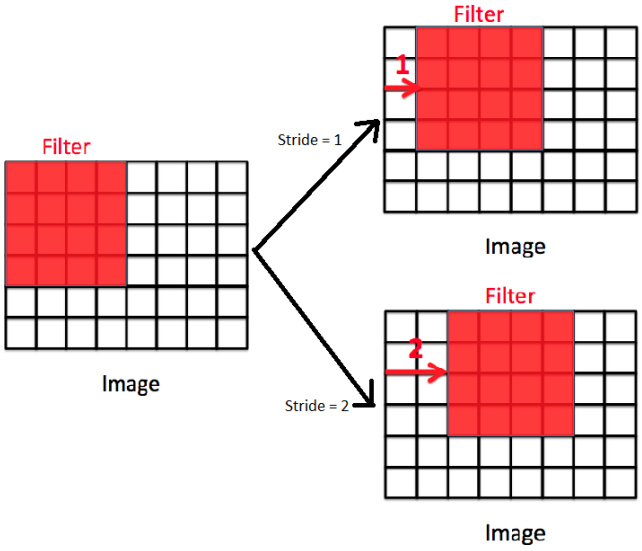


Figura 2.16: Exemplo do deslizamento do filtro configurado com 1 e 2 de *stride*. Adaptado da Fonte: [14].

Quando a imagem passa pela camada de convolução a quantidade de dados vai diminuindo. As vezes deseja-se preservar o tamanho da imagem. Para isso, pode ser colocado

uma borda em volta da imagem original utilizando o hiper-parâmetro padding, como ilustrado na Figura 2.17.

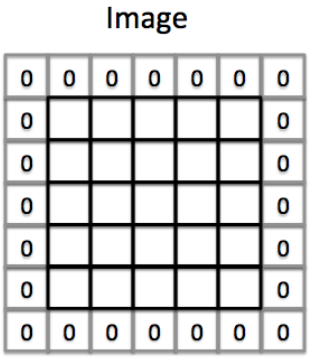


Figura 2.17: Padding na imagem. Fonte: [15].

2.5 Autoencoder

Uma estrutura de rede, a qual se tenta extrair informações relevantes e comprimir a imagem, armazena numa quantidade definida de valores, e remonta os imagem com estes dados de características.

É composto por 3 componentes: o codificador, a representação comprimida e o decodificador. O codificador extrai as características da entrada e produz uma representação comprimida. Em seguida o decodificador reconstrói a imagem, com o mesmo tamanho original, a partir desta representação [48].

Então foram definidos os parâmetros: tamanho da representação comprimida, número de camadas, número de neurônios por camada e função de custo.

Na Figura 2.18 é ilustrado a sua arquitetura e na Figura 2.19 como comporta a entrada e saída de dados.

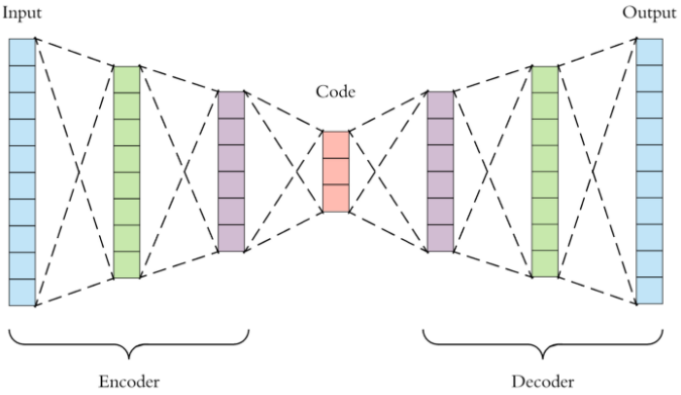


Figura 2.18: Arquitetura do autoencoder. Fonte: [16].

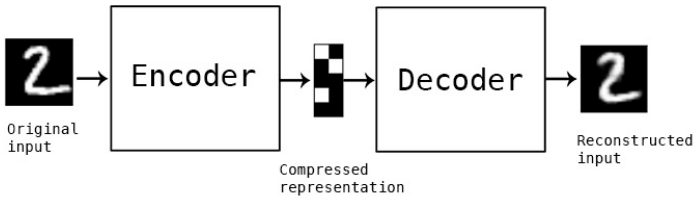


Figura 2.19: Exemplo de processamento dos dados do autoencoder. Fonte: [17].

2.6 Atari 2600 - Jogo Enduro

Lançado pela Atari em 1977, o Video Computer System (VCS) - ou simplesmente Atari 2600, é um video game projetado por Jay Miner, que teve muita popularidade possuindo muitos jogos clássicos como Space Invaders e Enduro em sua biblioteca [18].

Figura 2.20: Imagem do console Atari 2600. Fonte: [18].

O Enduro é um jogo eletrônico de corrida lançado pela Activision em 1983. O objetivo é ultrapassar uma certa quantidade de carros a cada dia, para permitir ao jogador continuar

correndo no dia seguinte [49]. Deve-se ultrapassar 200 carros no primeiro dia e 300 nos dias seguintes. O dia é composto por seis fases: de dia com o tempo ensolarado, de gelo, entardecendo, com neblina, noite e ao amanhecer.

(a) Fase de dia (b) Fase de gelo (c) Fase do entardecer

(d) Fase da neblina (e) Fase noturna (f) Fase do amanhecer

Figura 2.21: Fases do Enduro

Neste trabalho foi utilizada a Equação 2.24 para calcular a recompensa, sendo *objetivo* a quantidade de carros a ultrapassar para seguir ao próximo dia.

$$recompensa = \begin{cases} 0 & \text{se } posição <= 0 \\ posição & \text{se } 0 < posição < objetivo \\ objetivo & \text{se } posição >= objetivo \end{cases} \tag{2.24}$$

A *posição* do carro na corrida é calculada subtraindo a quantidade de carros que o jogador ultrapassou $c_{ultrapassou}$ pela quantidade de carros que o ultrapassaram $c_{ultrapassada}$, conforme

$$posição = c_{ultrapassou} - c_{ultrapassada}. \tag{2.25}$$

2.7 Validação-Cruzada: K-Fold

Esta é uma ferramenta para verificar a acurácia de uma rede neural. Consiste em dividir os dados em k partes de mesmo tamanho e utilizar uma dessas partes para teste e as outras para treinamento.

Figura 2.22: Exemplo representativo da validação cruzada/K-fold. Fonte: [19].

A Figura 2.22 mostra um exemplo utilizando cinco folds ($k = 5$). Os dados de treinamento em verde são inseridos para treinar a rede e, em seguida, testado com o dado em rosa.

2.8 Ferramentas

Nesta seção serão descritas as ferramentas usadas neste trabalho.

2.8.1 Python

Python é uma linguagem de programação de alto nível, muito popular nas áreas de análise de dados, desenvolvimento de algoritmos e de inteligência artificial, possuindo diversas bibliotecas como o Pytorch e o TensorFlow [50].

2.8.2 Pytorch

Ferramenta para Python, desenvolvido para criação de modelos de inteligência artificial. Seu principal desenvolvedor é o laboratório de pesquisa em AI do Facebook (FAIR, Facebook's AI Research). Foi lançado em 2016, e hoje, é o principal concorrente de outro framework com o mesmo propósito: Tensorflow, da Google [51].

2.8.3 Google Colab Pro

Para executar o treinamento dos modelos foi utilizado a plataforma Google Colab Pro. Nesta plataforma foi possível alugar uma GPU para o treinamento dos modelos. No momento da escrita desta monografia o valor do serviço é de R$58,00 por mês [52].

2.9 Trabalhos relacionados

Para contextualizar o projeto serão apresentados alguns projetos recentes na área.

2.9.1 AlphaGo

É um programa de computador capaz de jogar o jogo de tabuleiro Go e foi desenvolvido pela equipe *DeepMind*. Este projeto possui versões sucessoras que foram se tornando cada vez mais poderosas: o *AlphaGo Zero*, o *AlphaZero* e *MuZero* [53]. A Figura 2.23 ilustra suas sucessões.

Estes algoritmos utilizam um algoritmo de busca em árvore *Monte Carlo* para encontrar seus movimentos com base no conhecimento previamente adquirido por aprendizado de máquina, especificamente por uma rede neural artificial por meio de treinamento extensivo. Uma rede neural é treinada para identificar os melhores movimentos e as porcentagens de vitória desses movimentos. Essa rede melhora a força da busca em árvore, resultando em uma seleção de movimentos mais forte na próxima iteração. Após vencer do campeão *Lee Sedol* foi premiado com um 9-dan honorário da Associação de Baduk da Coreia e sua vitória escolhida pela revista científica *The Science* como uma das vice-campeãs do Breakthrough of the Year - uma premiação anual do desenvolvimento mais significativo nas pesquisas científicas feito pelo *AAAS journal Science* [54] - em 2016 [53]. Na Figura 2.24 ilustra uma representação a porcentagem de vitoria para as possíveis jogadas calculadas pelo algoritmo.

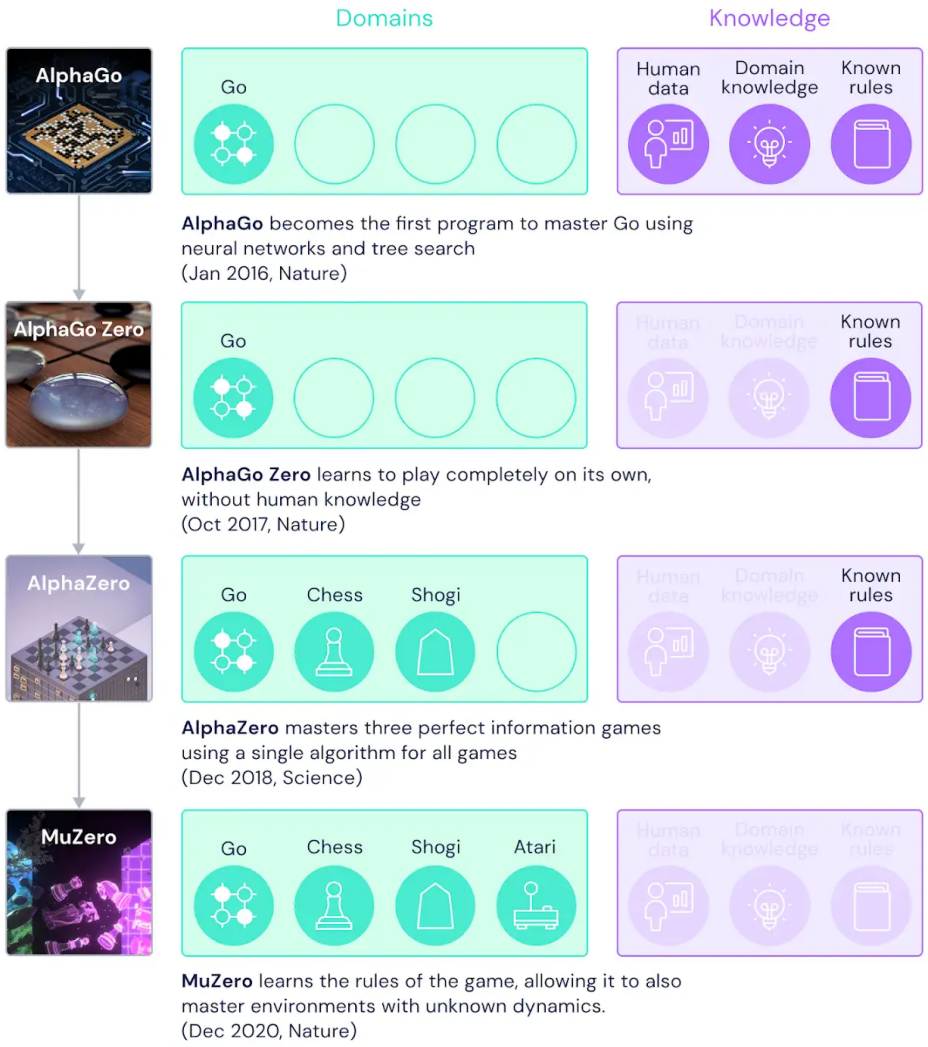


Figura 2.23: Algoritmos de inteligência artificial do DeepMind. Fonte: [20] .

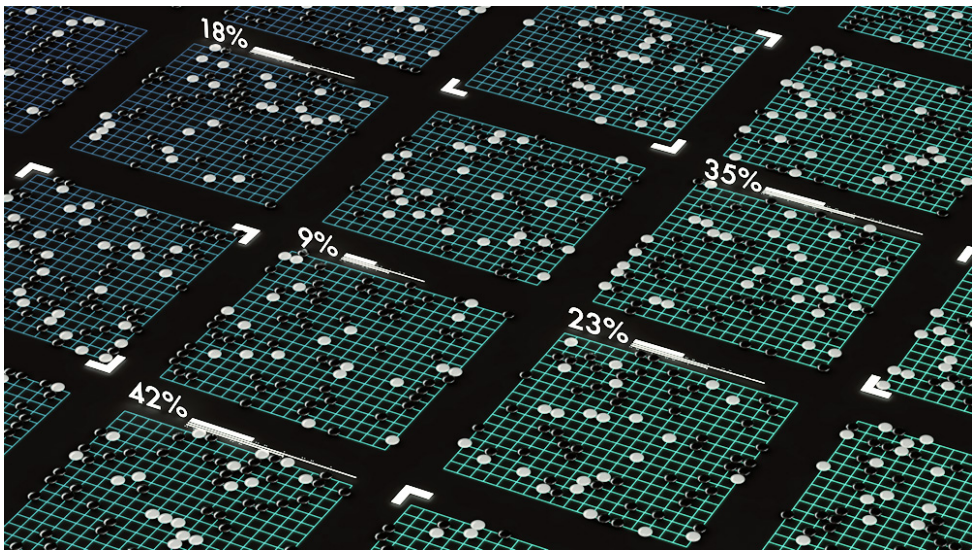


Figura 2.24: Representação das possíveis jogadas calculadas pelo AlphaGo. Fonte: [21].

2.9.2 OpenAI Five

Este projeto de inteligência artificial ficou famoso após vencer no campeonato mundial de *Defense of the Ancients 2* (Dota2) - um jogo competitivo multiplayer - em 13 de Abril de 2019 contra o time profissional OG. Na Figura 2.25 ilustra a tela ao ganhar a partida.

Figura 2.25: Tela de vitória no jogo Dota2. Fonte: [22].

O sistema foi desenvolvido pela OpenAI e a companhia utilizou este jogo para capturar as situações de natureza imprevisível e contínuas do mundo que o jogo proporciona. Foi afirmado que a natureza complexa do jogo e sua forte dependência no trabalho em equipe foram as principais razões para escolha. O algoritmo treinado foi utilizado em outros projetos como no controle de mãos robóticas, ilustrado na Figura 2.26.

O projeto tem sido comparado com outros IAs competitivos contra humanos como o *Deep Blue* da IBM para jogar xadrez e *AlphaGo* para o jogar Go.

Este algoritmo possui como estrutura uma camada LSTM com 4096 unidades que observa o estado atual do jogo extraído por uma API. A rede neural conduz as ações por meio de várias possibilidades, e cada a previsão possui uma análise. Foi utilizado o método de aprendizagem por reforço para o treinamento na infraestrutura "Rapid" - que consiste em duas camadas: uma troca as milhares de máquinas e as ajuda na comunicação entre si e a segunda camada executa o software [55].

Foi falado que o algoritmo foi expandido além dos jogos eletrônicos e com esse mesmo algoritmo foi capaz de treinar um braço robótico. Este braço foi capaz de resolver o cubo de Rubik, como ilustrado na Figura 2.26.

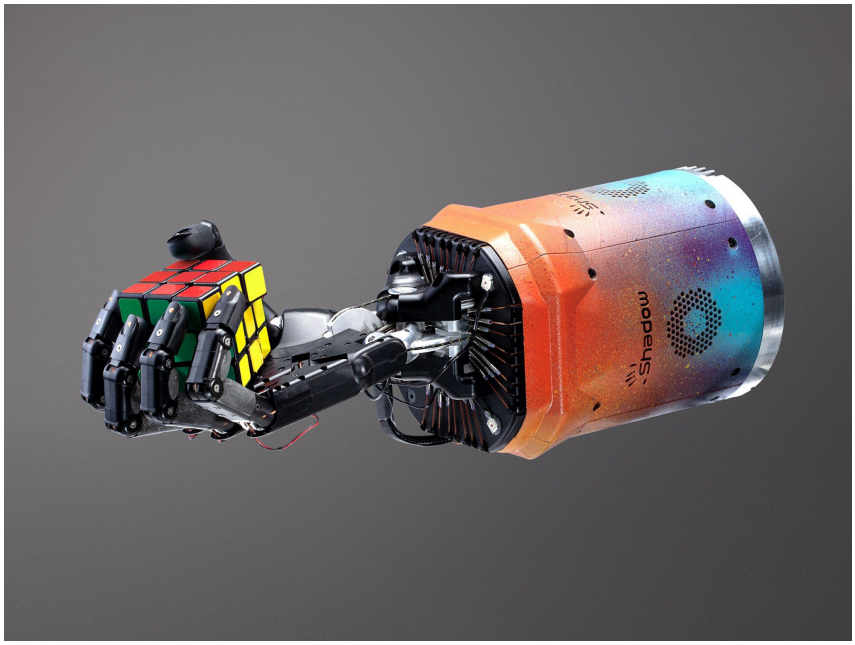


Figura 2.26: Mão robótica treinado com o mesmo algoritmo de treinamento por reforço. Fonte: [23].

2.9.3 MuZero

Após o sucesso ao treinar um modelo a jogar Go, a equipe da Deepmind desenvolveu um algoritmo capaz de jogar profissionalmente mais de 50 jogos sem conhecer suas regras, entre eles Shogi, xadrez, go e uma variedade de jogos de Atari.

Em seu artigo foram testados diversos outros algoritmos jogando Atari e é demonstrado que se tornou o estado da arte nesta tarefa. Após 30 partidas jogadas por 30 minutos o algoritmo conseguiu superar, em média, as pontuações dos jogadores humanos em quase todos os jogos listados na Figura 2.27.

O Muzero não conseguiu jogar bem em Montezuma's revenge. Neste jogo o algoritmo encontrou dificuldades em explorar o ambiente e armazenar referências de longo prazo.

Na Tabela 2.27 possui, para cada jogo, as performances do Muzero após 30 partidas jogadas por 30 minutos.

Game	Random	Human	SimPLe [20]	Ape-X [18]	R2D2 [21]	*MuZero*	*MuZero* normalized
alien	227.75	7,127.80	616.90	40,805.00	229,496.90	**741,812.63**	10,747.5 %
amidar	5.77	1,719.53	74.30	8,659.00	**29,321.40**	28,634.39	1,670.5 %
assault	222.39	742.00	527.20	24,559.00	108,197.00	**143,972.03**	27,664.9 %
asterix	210.00	8,503.33	1,128.30	313,305.00	**999,153.30**	998,425.00	12,036.4 %
asteroids	719.10	47,388.67	793.60	155,495.00	357,867.70	**678,558.64**	1,452.4 %
atlantis	12,850.00	29,028.13	20,992.50	944,498.00	1,620,764.00	**1,674,767.20**	10,272.6 %
bank heist	14.20	753.13	34.20	1,716.00	**24,235.90**	1,278.98	171.2 %
battle zone	2,360.00	37,187.50	4,031.20	98,895.00	751,880.00	**848,623.00**	2,429.9 %
beam rider	363.88	16,926.53	621.60	63,305.00	188,257.40	**454,993.53**	2,744.9 %
berzerk	123.65	2,630.42	-	57,197.00	53,318.70	**85,932.60**	3,423.1 %
bowling	23.11	160.73	30.00	18.00	219.50	**260.13**	172.2 %
boxing	0.05	12.06	7.80	**100.00**	98.50	**100.00**	832.2 %
breakout	1.72	30.47	16.40	801.00	837.70	**864.00**	2,999.2 %
centipede	2,090.87	12,017.04	-	12,974.00	599,140.30	**1,159,049.27**	11,655.6 %
chopper command	811.00	7,387.80	979.40	721,851.00	986,652.00	**991,039.70**	15,056.4 %
crazy climber	10,780.50	35,829.41	62,583.60	320,426.00	366,690.70	**458,315.40**	1,786.6 %
defender	2,874.50	18,688.89	-	411,944.00	665,792.00	**839,642.95**	5,291.2 %
demon attack	152.07	1,971.00	208.10	133,086.00	140,002.30	**143,964.26**	7,906.4 %
double dunk	-18.55	-16.40	-	**24.00**	23.70	23.94	1,976.3 %
enduro	0.00	860.53	-	2,177.00	2,372.70	**2,382.44**	276.9 %
fishing derby	-91.71	-38.80	-90.70	44.00	85.80	**91.16**	345.6 %
freeway	0.01	29.60	16.70	**34.00**	32.50	33.03	111.6 %
frostbite	65.20	4,334.67	236.90	9,329.00	315,456.40	**631,378.53**	14,786.7 %
gopher	257.60	2,412.50	596.80	120,501.00	124,776.30	**130,345.58**	6,036.8 %
gravitar	173.00	3,351.43	173.40	1,599.00	**15,680.70**	6,682.70	204.8 %
hero	1,026.97	30,826.38	2,656.60	31,656.00	39,537.10	**49,244.11**	161.8 %
ice hockey	-11.15	0.88	-11.60	33.00	**79.30**	67.04	650.0 %
jamesbond	29.00	302.80	100.50	21,323.00	25,354.00	**41,063.25**	14,986.9 %
kangaroo	52.00	3,035.00	51.20	1,416.00	14,130.70	**16,763.60**	560.2 %
krull	1,598.05	2,665.53	2,204.80	11,741.00	218,448.10	**269,358.27**	25,083.4 %
kung fu master	258.50	22,736.25	14,862.50	97,830.00	**233,413.30**	204,824.00	910.1 %
montezuma revenge	0.00	**4,753.33**	-	2,500.00	2,061.30	0.00	0.0 %
ms pacman	307.30	6,951.60	1,480.00	11,255.00	42,281.70	**243,401.10**	3,658.7 %
name this game	2,292.35	8,049.00	2,420.70	25,783.00	58,182.70	**157,177.85**	2,690.5 %
phoenix	761.40	7,242.60	-	224,491.00	864,020.00	**955,137.84**	14,725.3 %

Figura 2.27: Tabela com as performances do Muzero após 30 partidas jogadas por, cada uma, 30 minutos. A linha em destaque representa os resultados jogando Enduro. Adaptada da Fonte: [24].

Na primeira coluna é representado o jogo, na segunda a pontuação ao colocar comandos aleatórios no controle, na terceira coluna a pontuação máxima testada com os humanos, na quarta coluna a pontuação ao colocar o algoritmo SimPLe [56] para jogar, na quinta coluna a pontuação ao colocar o algoritmo Ape-X [57] para jogar, na sexta coluna o resultado com o algoritmo R2-D2 [58] jogando, na sétima coluna a pontuação do Muzero jogando e a última coluna "normalizada" representa o quanto o Muzero desempenhou melhor que a pontuação humana calculado a partir da Equação 2.26

$$s_{normalized} = \frac{s_{agent} - s_{random}}{s_{human} - s_{random}} \tag{2.26}$$

onde S_{agent} é a média das recompensas obtida pelo algoritmo Muzero, S_{random} a média das recompensas ao passar comandos aleatórios e S_{human} a média das recompensas obtida pelo jogador humano.

Capítulo 3

Metodologia Proposta

Neste capítulo será descrito como foi realizado este estudo, detalhando como se usou as ferramentas e a teoria vistas no capítulo anterior. Utilizou-se a ferramenta OpenAI Gym como ambiente de execução do jogo Enduro. Foram jogados 10 partidas e armazenados os seus frames e comandos no banco de dados.

Para gerar as redes neurais foi utilizado a linguagem Python e a biblioteca Pytorch. Em todas as estruturas de redes foi utilizada as redes recorrentes. Após os testes com modelos apenas com as redes recorrentes foi colocado camada convolucional para tentar melhorar os resultados.

A execução deste trabalho pode ser dividida em:

- Geração de dados
- Definição do método de avaliação
- Definição e treinamento do modelo
- Carregamento e avaliação dos modelos

3.1 Geração do banco de dados

O Banco de Dados é composto de diversas frames adquiridas durante 10 jogos executados por um agente humano, bem como os comandos que este enviou em cada situação.

Para gerar os frames foi utilizado a ferramenta OpenAI Gym. Esta ferramenta provê um ambiente de emulações de jogos de Atari e possibilita extrair os comandos executados pelo jogador ao ver cada frame.

O esquemático da criação do banco de dados é ilustrado na Figura 3.1.

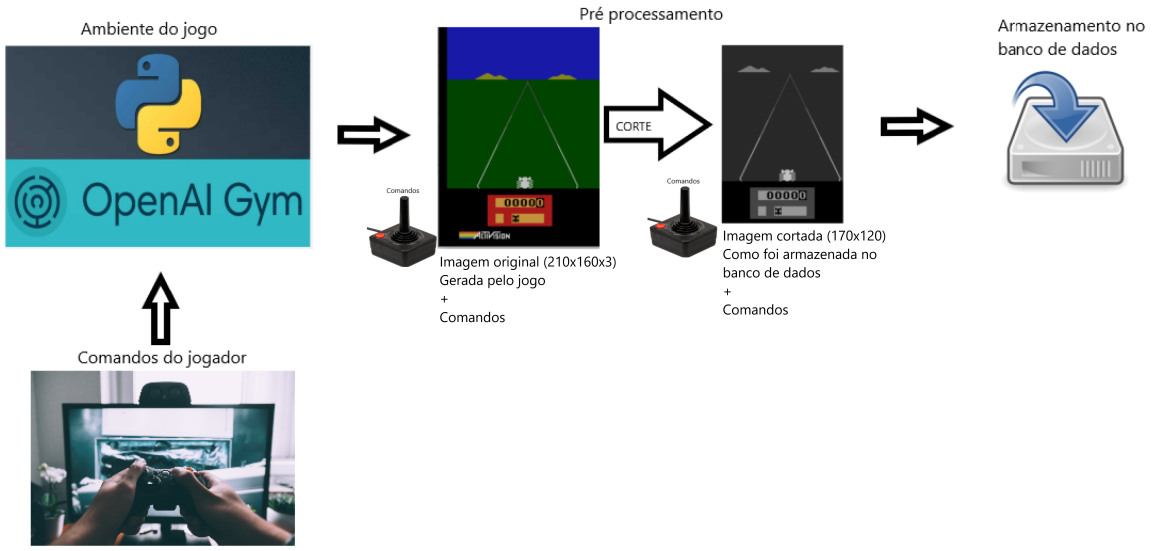


Figura 3.1: Esquemático da geração do banco de dados e treinamento.

3.1.1 Pré processamento das imagens

O frame gerado pela ferramenta está no formato RGB e possui dimensões de 210×160 pixels. Antes de armazenar o frame no banco de dados é feito os seguintes processamentos:

- Transformação dos três canais RGB em um canal de tons de cinza
- Corte da faixa preta a esquerda e o emblema da *Activision* na parte de baixo da imagem

Resultando em imagens de formato 170×120 pixels, e enfim, colocado no banco de dados.

Verificou-se que com 4500 frames de jogo era possível completar a primeira volta, passando por todas as cinco fases. Então executou-se dois tipos de partidas sendo a primeira usada apenas para controle e testes:

- 1 sequência de 4500 frames fazendo movimentos aleatórios
- 10 partidas de 4500 frames jogadas por humano

O resultado do jogo com comandos aleatórios será o primeiro teste e será comparado com os resultados do artigo [24], na qual o resultado da recompensa foi zero. Dos nove comandos o jogador executou apenas a de acelerar, acelerar para direita e acelerar para esquerda. Deste modo achou-se que o carro conseguiria ultrapassar o maior número de carros.

Para gerar e armazenar os dados implementado o Algoritmo 2, onde *frame* é a imagem gerada pelo ambiente e *action* o comando do jogador no controle. Os frames são armazenados em um buffer e salvo ao final de cada partida.

Algoritmo 2 *Gerar frames do jogo Enduro*

```
1: inicializa o array frame buffer
2: inicializa o array action buffer
3: while número de frames a serem gerados do
4:     Espera o tempo para atualizar a tela do jogo
5:     Atualiza a tela do jogo
6:     action ← comando do teclado
7:     frame ← redimensiona os frames a 170 × 120
8:     frame ← converte para tons de cinza
9:     frame buffer ← grava o frame no array frame buffer
10:    action buffer ← grava a ação no array action buffer
11: end while
12: salva o array frame buffer
13: salva o array action buffer
```

3.1.2 Segmentação de partidas

Após a geração de 10 jogos com 4500 frames foi realizada uma análise para segmentar manualmente as sequências de frames em acontecimentos relevantes - como o desvio de um carro, uma ultrapassagem ou manobra mais elaborada. Segmentando as partidas se fez um filtro eliminando os segmentos que apresentam batidas ou com menos de oito frames. Obtendo os dados foi montado a Tabela 3.1.

Tabela 3.1: Quantidade de segmentos com os eventos

	Total de segmentos	Batidas	Tamanho < 8	Utilizados
match 1	244	11	30	203
match 2	272	10	52	210
match 3	258	13	43	202
match 4	231	14	30	187
match 5	200	11	20	169
match 6	212	11	23	178
match 7	251	11	28	212
match 8	225	12	30	183
match 9	219	11	33	175
match 10	238	11	39	189

A primeira coluna corresponde a qual partida está sendo segmentada, a segunda coluna a quantidade de segmentos gerados pela análise, a terceira coluna número de segmentos apresentaram batidas em outros carros, a quarta coluna o número de segmentos que se tiveram menos de 8 frames de duração e a quinta coluna a quantidade de segmentos utilizados para o treinamento das redes.

Juntamente com os dados pré-processados foram armazenados os comandos do controle para cada frame. Os comandos possíveis corresponde a 9 ações do controle do Atari. Essas ações foram codificadas em *One Hot Encoder* conforme a Tabela 3.2.

Tabela 3.2: Codificação dos comandos

Comando	Ação	*One Hot Encoder*
0	Sem ação	100000000
1	Acelera	010000000
2	vai para direita	001000000
3	vai para esquerda	000100000
4	freia	000010000
5	freia e vai para direita	000001000
6	freia e vai para esquerda	000000100
7	acelera e vai para direita	000000010
8	acelera e vai para esquerda	000000001

Para processar o treinamento foi utilizado a ferramenta Google Colab Pro. Nele foi possível utilizar a GPU para acelerar o treinamento.

Deseja-se projetar uma rede neural capaz de imitar os comandos humanos por estímulos visuais. Apesar do pré-processamento durante o armazenamento dos frames no banco de dados, ainda existem informações desnecessárias como o céu e o placar. Então, estes elementos foram retirados antes de alimentá-los a rede. O processamento completo da imagem pode ser vista na Figura 3.2.

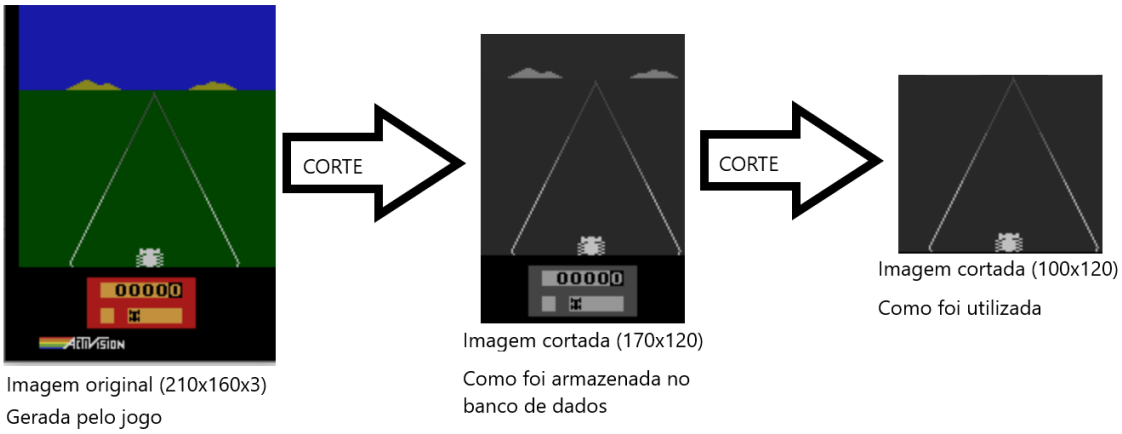


Figura 3.2: Os frames alimentados as redes.

3.2 Definição do método de avaliação

Para avaliar a performance das redes foi utilizado o método do artigo *Muzero* [24] baseado na contagem de recompensas (ou pontuação) recebida durante a partida. Neste trabalho a recompensa é calculada de acordo com as Equações 2.24 e 2.25 apresentadas anteriormente.

Para cada *fold* é calculada a recompensa S_{fold} recebida pela rede. Deste modo a média do desempenho da rede para os 10 $folds$ e seu desvio padrão são calculados pela Equações 3.1 e 3.2 respectivamente.

$$\overline{S}_{rede} = \frac{1}{10} \sum_{fold=1}^{10} S_{fold} \tag{3.1}$$

$$\sigma_{rede} = \sqrt{\sum_{fold=1}^{10} (S_{fold} - \bar{S}_{rede})^2}. \tag{3.2}$$

3.3 Definição e treinamento do modelo

Na Figura 3.3 é ilustrado o esquemático do treinamento da rede neural.

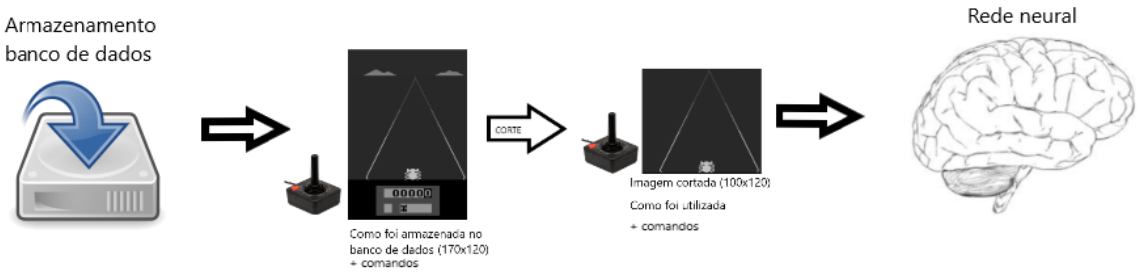


Figura 3.3: Esquemático do treinamento da rede neural.

As imagens e os comandos são lidos do banco de dados e, em seguida, aplica-se um tratamento na imagem cortando o céu e o placar por serem informações desnecessárias ao treinamento. Enfim é feito a normalização dos dados, para que os valores fiquem no intervalo $[0, 1]$, e codificação dos comandos (*One Hot Encoder*), e, finalmente, aplicado na entrada da rede.

Após a preparação dos dados foi definido alguns parâmetros das redes. Foi definido os treinamentos com dez mil épocas. Como função de custo o erro médio quadrático. O método de treinamento de neurais escolhido foi o Adam com as configurações padrões do Pytorch.

A saída das redes foi configurada com *Softmax* e utilizado a técnica de validação cruzada. O *k-fold* utilizado foi de 10 folds, com 9 sequências utilizadas no treinamento e 1 para teste. Para este projeto foram propostos quatro tipos de rede, todos implementados em Pytorch:

- Rede de Elman
- Rede LSTM
- Autoencoder + LSTM
- Rede convolucional + LSTM

Estruturou-se e treinou cada tipo de rede com 100, 200 e 500 neurônios. O número de neurônios surgiu da prática de se usar inicialmente a média geométrica dos números de neurônios das camadas de entrada e de saída. Mais a frente deste capítulo será descrito que utilizou-se 12.000 neurônios na camada de entrada e 9 neurônios na camada de saída nas redes Elman e LSTM obtendo a média geométrica de 328,63, aproximadamente 350.

Também será visto que utilizou-se 1.200 neurônios na entrada e 9 neurônios na saída das camadas recorrente LSTM das estruturas Autoencoder + LSTM e Rede convolucional + LSTM. A média geométrica destes dois valores resulta em 103,92, aproximadamente 100. Portanto propôs-se o uso de 100 neurônios, seguindo o valor anteriormente obtido, e um valor equidistante do valor de 350, definindo em 200 e 500 neurônios.

Configurou-se para que fosse salva a melhor rede, ou seja, aquela que apresenta a menor função de perda do modelo durante o treinamento, conforme mostrado na Figura 3.4. Esta avaliação é feita a cada 10 épocas.

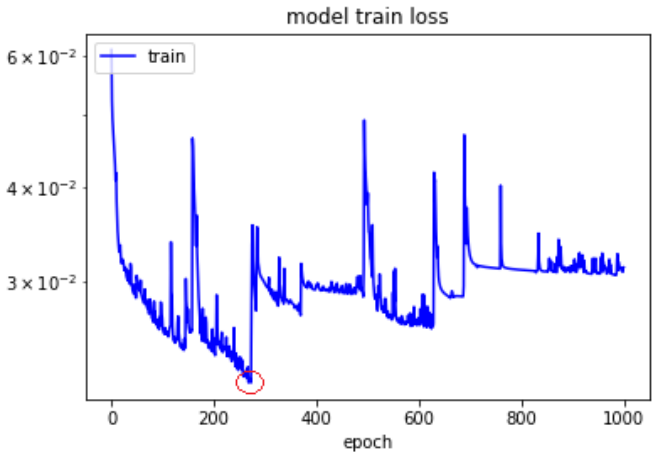


Figura 3.4: Exemplo do último modelo armazenado durante o treinamento desta rede (região marcada em vermelho) a época está representada de 1:10.

3.3.1 Rede de Elman

Propôs-se utilizar, inicialmente uma das redes recorrentes mais simples para avaliar seu desempenho. Uma RNN de Elman com uma camada de entrada, uma camada escondida com a camada de contexto e uma de saída. A ferramenta Pytorch disponibiliza uma camada chamada RNN [59] que aplica camadas da rede Elman com uma função de ativação tangente hiperbólica ou ReLU. Neste trabalho utilizou-se a configuração com a tangente hiperbólica que é com a qual o modelo foi originalmente proposto por Elman. Para a função de custo foi escolhida a função do erro médio quadrático definido pela Equação 2.8. O modelo foi então testado com 100, 200 e 500 neurônios na camada escondida. Para colocar na camada de entrada as imagens foram achatadas, transformando da dimensão 2D para 1D. Então, o frame de (100×120) foi transformado em um vetor de (12000×1). A saída da rede foi configurada com Softmax. A Figura 3.5 ilustra um esboço da rede Elman implementada.

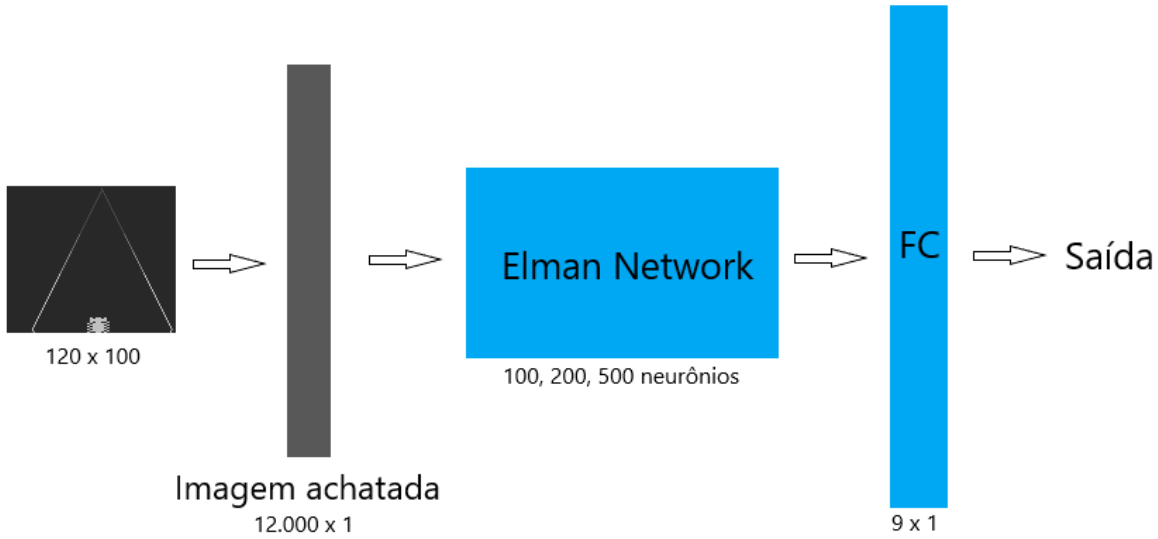


Figura 3.5: Esboço da Rede Elman implementada.

A imagem é achatada, passada para a camada de Elman, em seguida, é passada a camada densa (*Fully Connected*) e, na saída, é classificado entre os nove comandos possíveis.

3.3.2 Rede LSTM

A seguir propôs-se a utilização de uma rede neural profunda recorrente, estado da arte em processamento de sinais sequenciais, a rede LSTM. Uma rede LSTM simples composta por uma camada de entrada, uma camada escondida e uma camada de saída. Utilizou-se a camada LSTM provida pela ferramenta Pytorch. Os parâmetros escolhidos foram os mesmos da Rede de Elman, utilizando como função de custo o erro médio quadrático, variando de 100, 200, e 500 neurônios na camada escondida. Assim como nas camadas de entrada da rede Elman, as imagens foram achatadas transformando da dimensão 2D para 1D. Então, o frame de (100×120) foi transformado em um vetor (12000×1). A saída da rede foi configurada com Softmax.

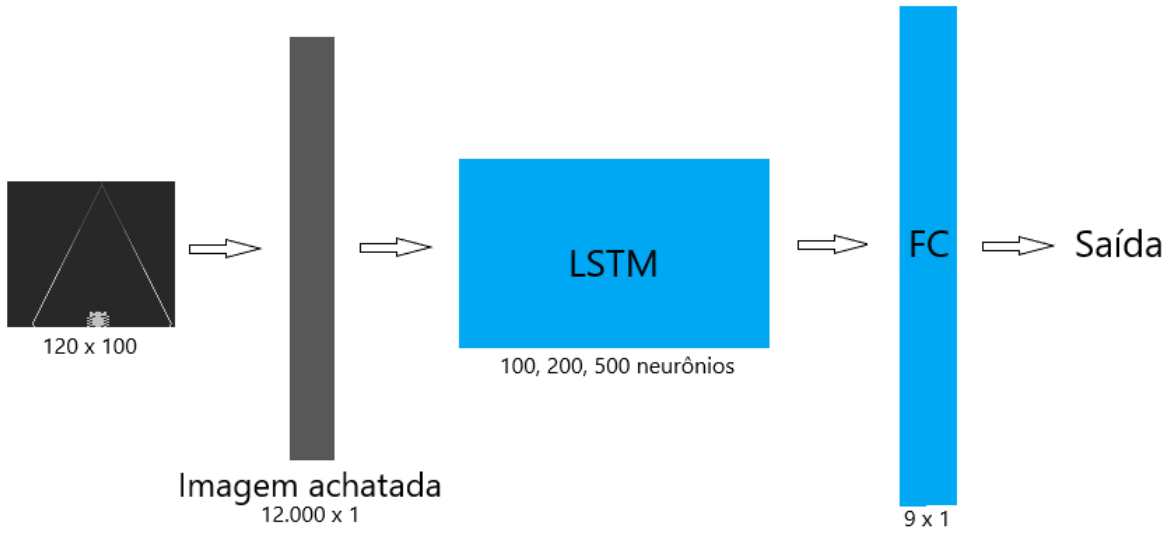


Figura 3.6: Esboço da Rede LSTM implementada.

A Figura 3.6 ilustra um esboço da rede Elman implementada. A imagem é achatada, passada para a camada LSTM, em seguida, é passada a camada densa (*Fully Connected*) e, na saída, é classificado entre os nove comandos possíveis.

3.3.3 Autoencoder

Sabendo da eficiência das camadas convolucionais em processamento de imagens, foi implementado um autoencoder para usar as informações da codificação para treinar um modelo de maneira mais rápida, devido à redução da dimensão dos dados de entrada das redes neurais recorrentes. Na entrada da rede as imagens foram passadas em formato (100×120) e gerados na saída dados em formato (1200×1). A camada convolucional do codificador foi definido com 1 canal de entrada, 4 canais de saída e filtro de 5×5. O *stride* foi configurado como 1 e o *padding* como 0. A camada densa do codificador é composta por 1200 neurônios, para gerar um código que representa 10% da quantidade de pixels de uma frame. Na saída da camada convolucional e da camada densa do codificador foi colocado uma função de ativação ReLU. Na parte da decodificação foi utilizado o `CONVTRANSPOSE2D`, disponibilizado em Pytorch, para espelhar a operação de convolução.

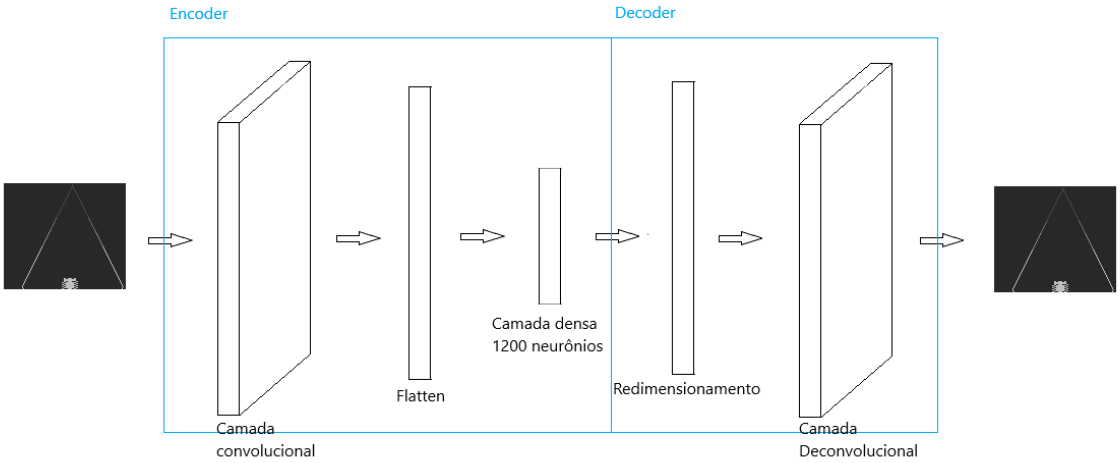


Figura 3.7: Esboço do autoencoder implementado.

Na Figura 3.7 é esboçado a estrutura do autoencoder implementado. As bordas quadradas em azul contornam as partes do codificador e do decodificador. Na entrada é passada a imagem em escala de cinza em formato (100×120). A parte do codificador é composto com uma camada convolucional, um achatamento para uma dimensão, então é ligada a uma camada densa que gera uma imagem codificada em tamanho (1200×1). Passando para o decodificador é feito o redimensionamento do formato para a operação de deconvolução, gerando a saída da rede autoencoder.

Após o treinamento do autoencoder, foi utilizado a parte codificadora para gerar um novo banco de dados compactado, transformando os frames de 12000 pixels em 1200 valores numéricos. Estes novos dados foram alimentados a rede LSTM e treinados como os modelos anteriores. Utilizou-se como função de custo o erro médio quadrático, variando de 100, 200, e 500 neurônios na camada escondida.

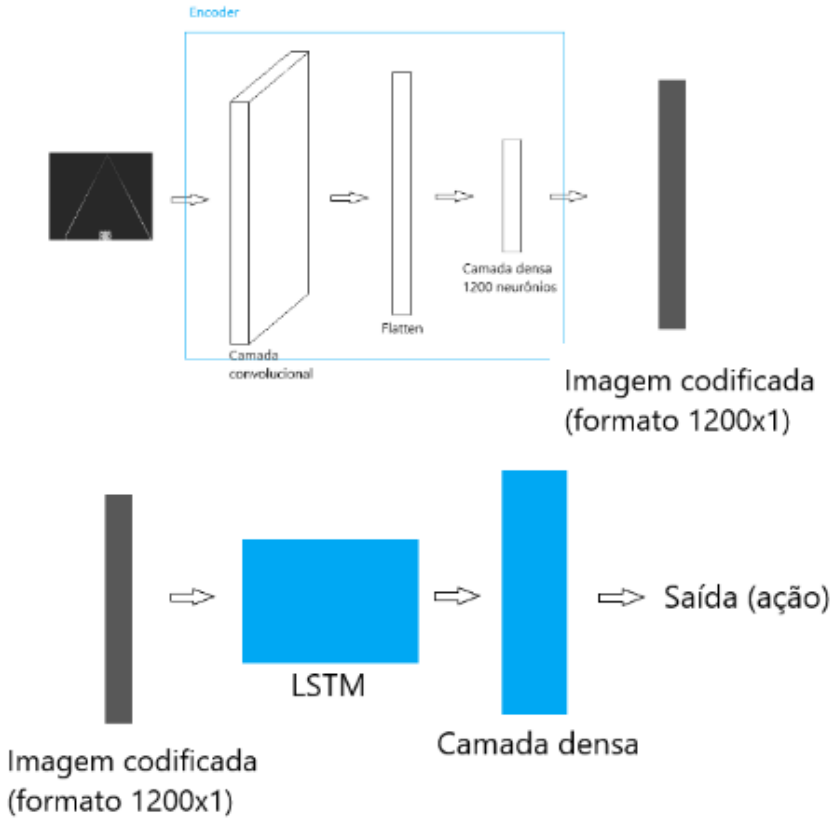


Figura 3.8: Esboço do treinamento da LSTM com os dados gerados com codificador do autoencoder.

Na Figura 3.8 é esboçado i) a rede que transforma os frames em dados codificados e ii) a rede de treinamento alimentada com os dados codificados. Na parte de cima é mostrado a estrutura codificadora usada para gerar dados transformados. Em sua entrada é passada a imagem em escala de cinza no formato (100×120), passada a uma camada convolucional, achatada para uma dimensão, que passa para a camada densa de 1200 neurônios e gera uma saída de tamanho (1200×1). Na parte de baixo, os dados codificados são colocados na entrada da rede LSTM, passando por uma camada densa e classificado em um dos possíveis comandos do jogo.

3.3.4 Rede Convolucional e rede LSTM

A fim de tentar melhorar o modelo e tirar melhor proveito das capacidades das convoluções foi implementado um modelo com uma camada convolucional seguida de uma camada recorrente LSTM, conforme apresentado na Figura 3.9.

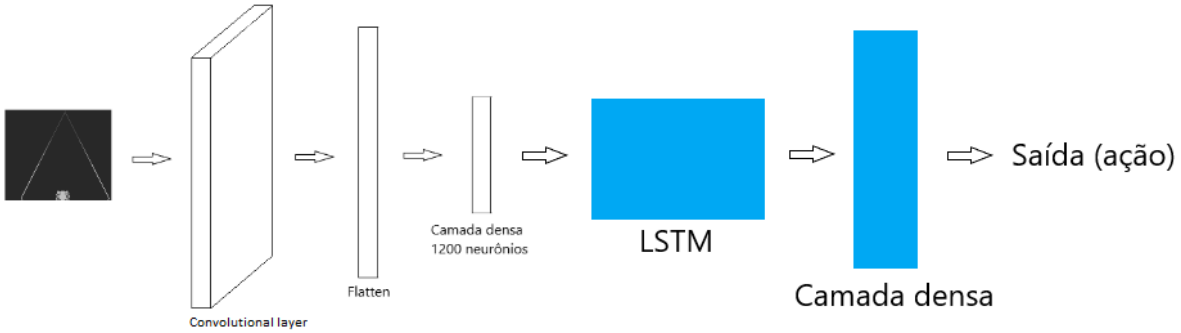


Figura 3.9: Esboço do modelo com camada convolucional e camada LSTM.

Assim como no codificador do autoencoder a camada convolucional possui 1 canal de entrada, 4 canais de saída, um filtro de tamanho 5×5, *stride* de 1 e *padding* de 0. Em sua saída foi colocada uma função de ativação ReLU. A camada densa foi configurada com 1200 neurônios como no codificador do autoencoder. A saída desta camada densa é conectada à camada recorrente LSTM, que é ligada a uma camada densa e, por fim, passa para os nove neurônios da camada de saída com *softmax* correspondendo ao número de comandos possíveis dentro do jogo.

3.4 Avaliação dos modelos

A Figura 3.10 ilustra um esquemático do funcionamento da rede durante o jogo.

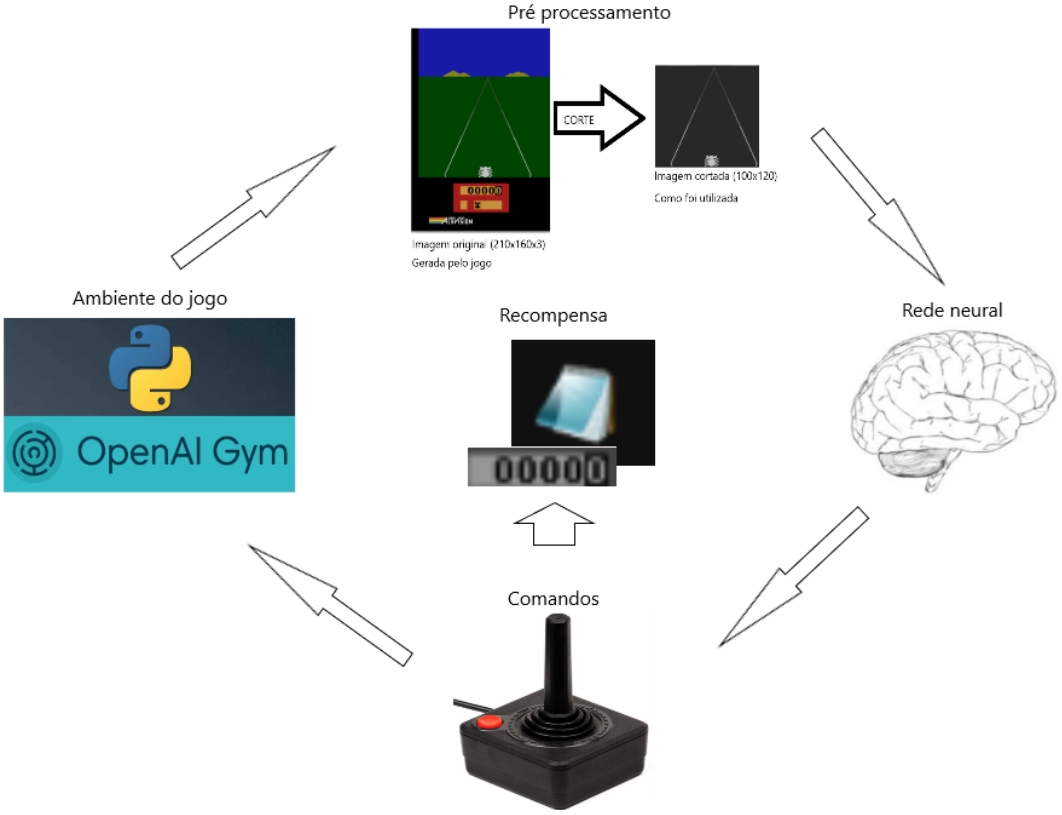


Figura 3.10: Esquemático do modelo colocado para jogar.

Implementou-se um script para carregar os modelos treinados. Para colocar o modelo para jogar, foi recriado as camadas separadamente copiando os pesos e empilhados-os na ordem como é executada no modelo.

Algoritmo 3 *Jogar Enduro com camadas criadas*

while número de frames a serem jogados **do**
    Espera o tempo para atualizar a tela do jogo
    frame ← Atualiza a tela do jogo
    frame ← transforma o frame em escala de cinza
    frame ← normaliza o frame
    ação ← Aplica a camada de rede recorrente
    ação ← Aplica a camada densa
    ação ← Aplica Softmax e obtêm a saída
    Alimenta o ambiente com a próxima ação prevista
end while

Após a obtenção das recompensas dos modelos gerados pelos *folds* de estrutura de rede neural é calculado a média e desvio padrão das recompensas na rede.

Capítulo 4

Resultados Obtidos

Neste capítulo são descritos os experimentos realizados para validar os modelos propostos no capítulo anterior, mostrando os seus resultados e a suas análises.

Inicialmente, jogou-se Enduro enviando comandos aleatórios ao ambiente, conforme metodologia apresentada em [24]. Assim como no artigo, a recompensa obtida foi zero.

O jogador humano gerou dez partidas jogadas por 4.500 frames, equivalente a dois minutos e 30 segundos se jogado a 30 frames por segundo. As recompensas obtidas são apresentados na Tabela 4.1

Tabela 4.1: Recompensas dos dados de treinamento

	recompensas
match 1	212
match 2	213
match 3	214
match 4	204
match 5	213
match 6	213
match 7	200
match 8	212
match 9	208
match 10	202
Média	208

Em seguida os modelos foram implementados e treinados com os dados gerados pelo humano. Como se tem dez partidas, e $k = 10$, cada k do k-fold é mapeado para uma partida. Deste modo, cada estrutura de rede gerará dez modelos, sempre com nove partidas como treinamento e uma partida como teste. Para avaliar uma partida, a rede

neural é colocada para jogar Enduro e verifica-se a recompensa recebida. Para medir o desempenho da rede é calculada a média e desvio padrão das recompensas ao longo do *k-fold*.

4.1 Rede de Elman

As redes de Elman possuem 12.000 neurônios na camada de entrada, foram testadas camadas escondidas com 100, 200 e 500 neurônios, e 9 neurônios na camada de saída. Escolheu-se um *batch size* de nove correspondendo ao número de sequência de dados.

Após o treinamento e a submissão dos modelos a simulação de jogo obteve-se os resultados mostrados na Tabela 4.2

Tabela 4.2: Recompensa da rede Elman

neurônios	100	200	500
recompensa média	103,5	109,5	115,5
desvio padrão	33,01	39,27	16,94
tempo de treinamento	17:28:49	22:47:34	26:17:34

A Tabela 4.2 apresenta para cada número de neurônios na camada escondida recorrente a pontuação média, seu desvio padrão e o tempo de treinamento dos 10 folds de um modelo no formato `hh:mm:ss`.

Na Figura 4.1 é mostrado o gráfico em boxplot das recompensas pelo número de neurônios na camada escondida e em comparação com a recompensa recebida pelo humano.

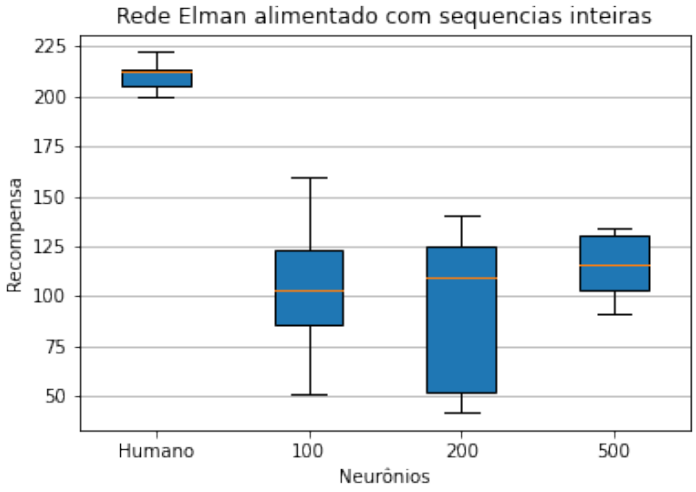
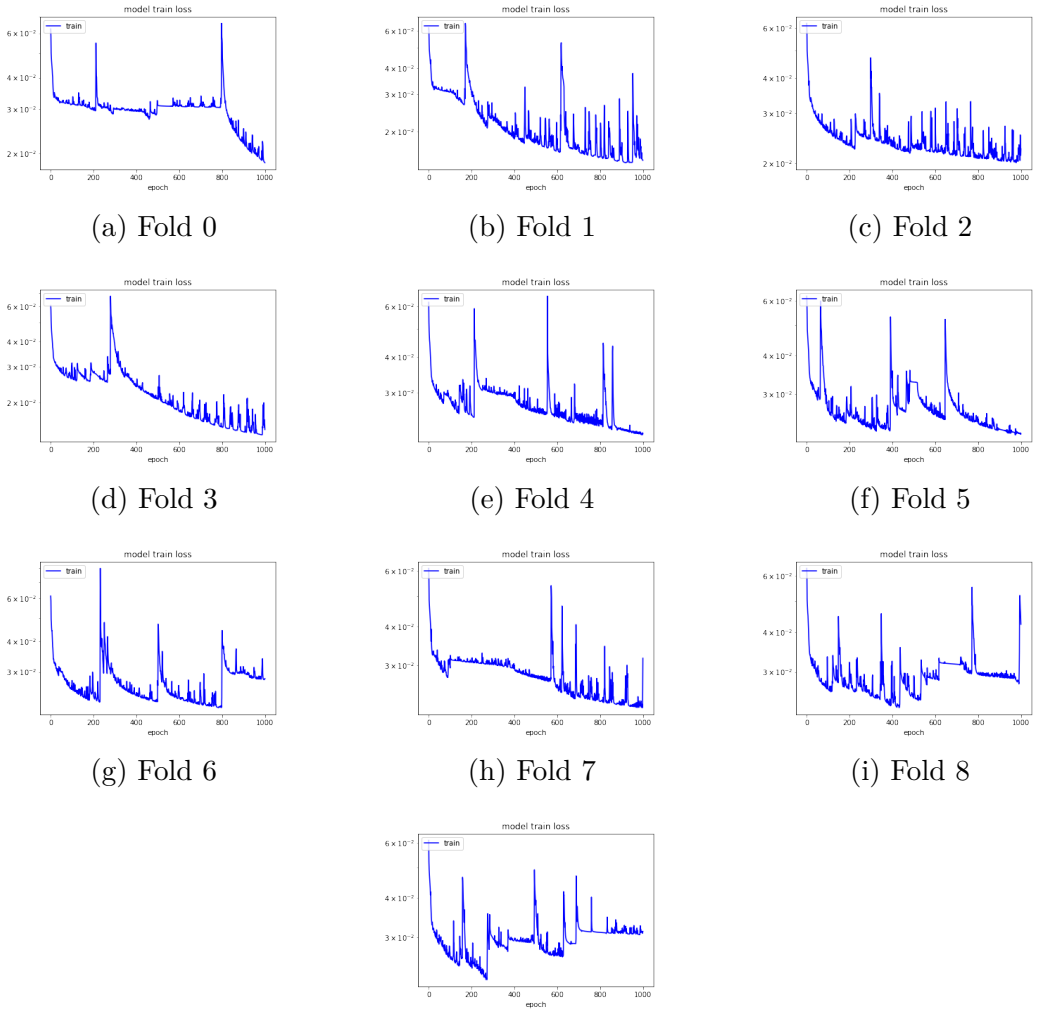


Figura 4.1: Gráfico de recompensas sobre neurônios da rede Elman alimentada com sequências inteiras.

Analisando a média das recompensas e o desvio padrão, a melhor média foi de 115,5 e o menor desvio padrão de 16,94. Assim, para ambos os parâmetros analisados a melhor rede foi obtida usando 500 neurônios. Esta estrutura conseguiu convergir bem, logo foram analisados os gráficos da função de custo ilustrados na Figura 4.2.



(a) Fold 0 (b) Fold 1 (c) Fold 2

(d) Fold 3 (e) Fold 4 (f) Fold 5

(g) Fold 6 (h) Fold 7 (i) Fold 8

(j) Fold 9

Figura 4.2: Gráficos da perda durante o treinamento do modelo Elman com 500 neurônios

Parece que, apesar da rede Elman com 500 neurônios mostrar melhores resultados, alguns de seus modelos tinham dificuldades em continuar convergindo. Esse fenômeno foi observado nos *folds* seis, oito e nove. No entanto, percebe-se que nos outros *folds* a continuação do treinamento para mais de 10.000 épocas seria aconselhável. Porém, o desempenho obtidos pelas redes neurais de Elman, ainda ficaram muito aquém do desempenho do ser humano que gerou os dados de treino.

A fim de tentar obter melhores resultados foram testadas redes LSTM.

4.2 Rede LSTM

A seguir, será apresentado os resultados utilizando a rede LSTM no lugar da rede Elman e feito uma análise sobre elas. As redes LSTM possuem 12.000 neurônios de entrada, foram testadas camadas escondidas com 100, 200 e 500 neurônios, e 9 neurônios na camada de saída. Para a análise do LSTM foi utilizado as sequências inteiras das partidas e as sequências segmentadas como descrito na Seção 3.1.2.

4.2.1 Alimentando com sequências inteiras

Utilizando o mesmo conceito do treinamento com a rede Elman foi utilizado *batch size* de nove. Treinou-se as redes e obteve-se os resultados apresentados na Tabela 4.3.

Tabela 4.3: Recompensa da rede LSTM alimentado com dados inteiros

neurônios	100	200	500
recompensa média	126,5	117,5	127
desvio padrão	22,39	11,35	16,57
tempo de treinamento	26:07:58	28:36:53	57:31:35

Na Figura 4.3 é mostrado o gráfico em boxplot das recompensas pelo número de neurônios da camada escondida.

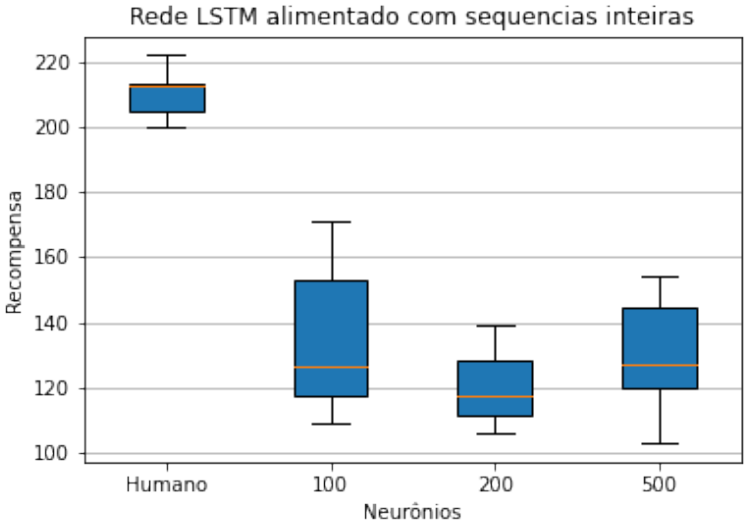


Figura 4.3: Gráfico de recompensas sobre neurônios da rede LSTM alimentada com sequências inteiras.

O melhor resultado obtido com rede Elman, com estrutura de 500 neurônios na camada escondida, foi média das recompensas de 115,5 e 16,94 de desvio padrão. Observa-se que todas as médias das recompensas da rede LSTM superaram o melhor resultado da rede Elman, já que, o pior resultado da rede LSTM foi de 117,5 (2 pontos a mais que a melhor recompensa da rede Elman).

Na rede LSTM a melhor recompensa média pode ser alcançada utilizando 100 e 500 neurônios, atingindo 126,5 e 127 pontos respectivamente.

Visto que a rede LSTM obteve desempenho melhor em todos os número de neurônios propostos, foi descartado as análises com a rede Elman nos experimentos seguintes.

4.2.2 Alimentando com segmentos

Segmentando os dados, o número de *batch sizes* foi alterado. Para treinar a rede com dados segmentados o *batch size* variou de acordo com a quantidade de segmentos gerados em cada partida. Para ver quantos segmentos possui cada partida pode-se consultar a Tabela 3.1. Então para a primeira partida foram usados 203 segmentos, para a segunda partida 210 segmentos, para a terceira 202, para a quarta 187, para a quinta 169, para a sexta 178, para a sétima 212, para a oitava 183, para a nona 175 e para a décima 189.

Os resultados obtidos para os 10 folds são apresentados na Tabela 4.4

Tabela 4.4: Recompensa da rede LSTM alimentado com dados segmentados

neurônios	100	200	500
recompensa média	97,00	91,5	102,5
desvio padrão	33,80	26,46	23,70
tempo de treinamento	63:37:41	68:30:45	79:53:47

Na Figura 4.4 é mostrado o gráfico em boxplot das recompensas pelo número de neurônios na camada escondida.

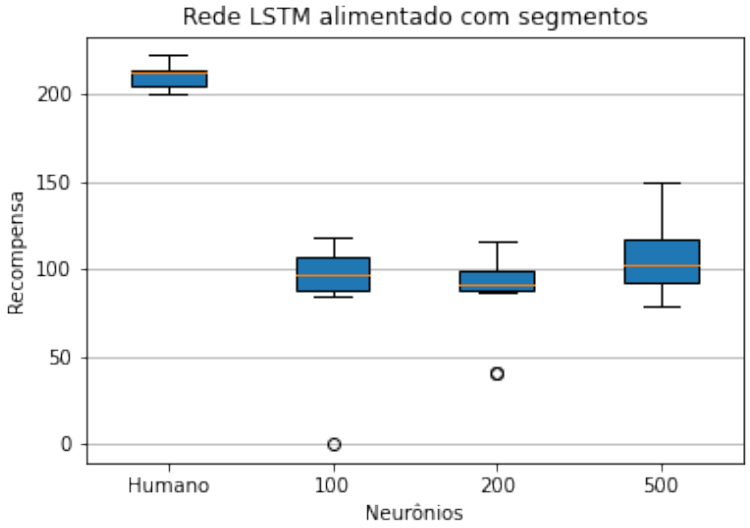


Figura 4.4: Gráfico de recompensas sobre neurônios da rede LSTM alimentada com sequências segmentadas.

Analisando a média das recompensas da rede LSTM alimentada com segmentos variando apenas o número de neurônios, a melhor recompensa média observada foi utilizando 500 neurônios com 102,5 pontos e desvio padrão de 23,70.

A rede LSTM alimentada com segmentos menores de dados apresentou um resultado pior do que alimentado com a sequência inteira. Um dos problemas que pode estar afetando o resultado é a quantidade de segmentos semelhantes criados ao separar as partidas, especializando algumas situações ao invés de generalizar o problema.

4.3 Autoencoder + LSTM

As redes até agora testadas não se aproveitavam da estrutura bidimensional da imagem, colocando diretamente os valores dos pixels em um grande vetor na entrada da rede. Porém, para próxima análise, se deseja aproveitar as estruturas das imagens com o uso de uma camada convolucional. Para a análise do Autoencoder + LSTM foi utilizado as sequências inteiras de partidas e sequências segmentadas como descrito na seção do pré-processamento.

Para estudar a influência de camadas convolucionais foi implementado um autoencoder e utilizado sua parte codificadora para gerar os dados transformados, para então, alimentar uma rede LSTM.

4.3.1 Geração do autoencoder

Para o treinamento do autoencoder foi utilizado um *batch size* de 4500, correspondendo a quantidade de frames em cada partida. O autoencoder gerado obteve as seguintes características:

- Época armazenada (melhor modelo): 6089
- Função de perda: $1,32 \times 10^{-3}$
- Tempo de treinamento: 21 horas e 39 minutos
- GPU: Tesla P100-PCIE-16GB

Do treinamento do autoencoder obteve-se o gráfico função de perda mostrada na Figura 4.5.

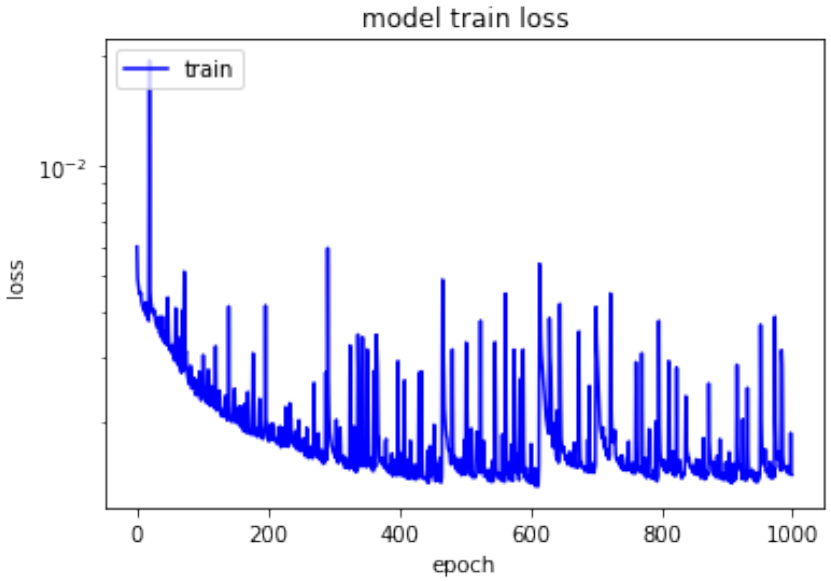


Figura 4.5: Função de custo do autoencoder.

Foi então verificada a saída reconstruída do autoencoder (Figura 4.7) comparada com a imagem de entrada (Figura 4.6).

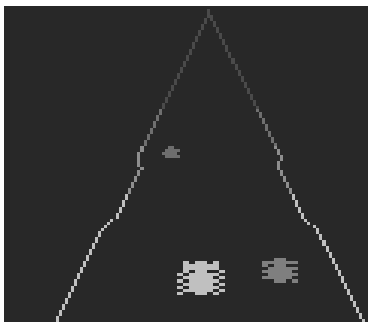


Figura 4.6: Imagem original

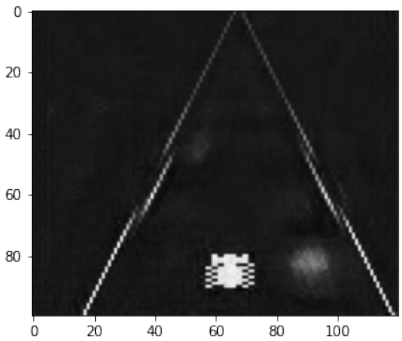


Figura 4.7: Imagem reconstruída a partir do autoencoder

Analisando a Figura 4.7 percebe-se que o modelo conseguiu extrair as linhas da pista e os carros localizados próximos ao jogador. Porém, os carros mais distantes tiveram prejuízo na representação, o que pode prejudicar na decisão das jogadas.

4.3.2 Codificação dos dados

Utilizando a parte codificadora do autoencoder transformou-se todas as imagens 100×120 em um vetor de dimensão 1200×1 como é ilustrado na Figura 4.8. Esses dados foram utilizados para treinar as redes LSTM.

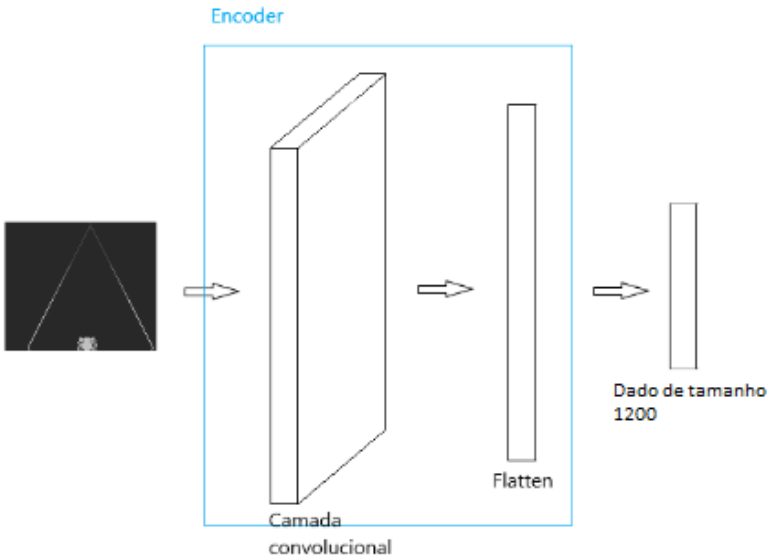


Figura 4.8: Esboço do tratamento das imagens na codificação.

4.3.3 Dados codificados + LSTM: Alimentado com sequências inteiras

Como nos modelos alimentados com sequências inteiras o *batch size* utilizado foi de tamanho 9. Visto que se tem 10 sequências e em cada *fold* uma sequência é usada para teste. Esses modelos foram alimentados com 9×4.500=405.000 vetores de formato 1200×1.

Os resultados obtidos são apresentados na Tabela 4.5.

Tabela 4.5: Recompensa da rede autoencoder + LSTM alimentado com dados inteiros

neurônios	100	200	500
recompensa média	27,0	21,0	14,0
desvio padrão	12,87	15,25	9,79
tempo de treinamento	6:28:03	8:46:44	18:03:59

Na Figura 4.9 é mostrado o gráfico em boxplot das recompensas pelo número de neurônios da camada escondida.

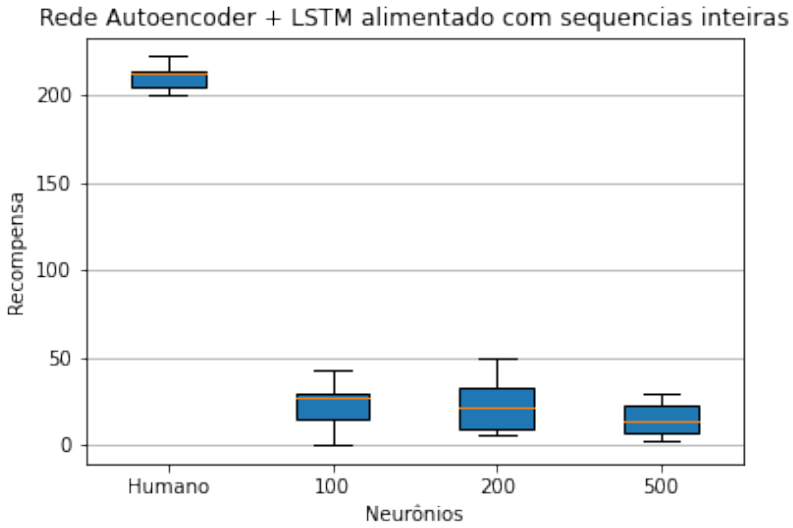


Figura 4.9: Gráfico de recompensas sobre neurônios da rede Autoencoder + LSTM alimentada com sequências inteiras.

A estrutura com a melhor recompensa média 27,0 foi obtida com 100 neurônios.

O treinamento dessas redes se mostrou ser veloz. Comparando com os tempos médios das redes treinadas com 500 neurônios, verificou-se um tempo 2,6 vezes menor que a rede Elman, 6 vezes que a rede LSTM treinada com sequências inteiras. Porém, os resultados não se mostraram satisfatórios, obtendo desempenho inferior aos modelos Elman e LSTM puro.

Talvez o codificador usado não esteja conseguindo extrair informações o suficiente, já que, estamos utilizando uma camada simples convolucional. Com isto, pode se estar perdendo informação demais não conseguindo identificar os carros mais distantes como observado na Figura 4.7.

4.3.4 Dados codificados + LSTM: Alimentando com segmentos

Foi feito o mesmo procedimento de segmentação dos dados e treinadas redes LSTM com camada de codificação. Os resultados obtidos são apresentados na Tabela 4.6.

Tabela 4.6: Recompensa da rede autoencoder + LSTM alimentado com dados segmentados

neurônios	100	200	500
recompensa média	86,5	91,5	111,5
desvio padrão	26,00	15,03	23,13
tempo de treinamento	8:36:15	8:54:32	9:50:51

Na Figura 4.10 é mostrado o gráfico em boxplot das recompensas pelo número de neurônios na camada escondida.

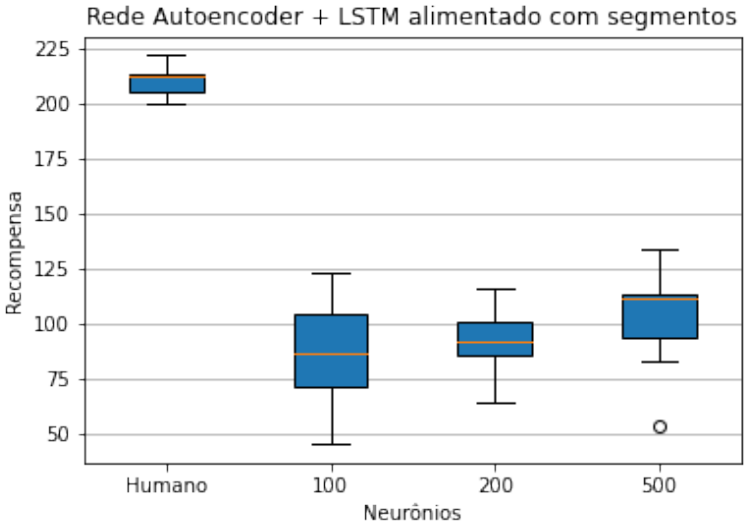


Figura 4.10: Gráfico de recompensas sobre neurônios da rede Autoencoder + LSTM alimentada com sequências segmentadas.

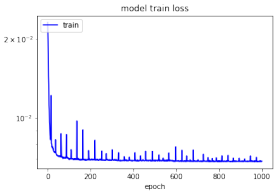
O treinamento com dados segmentados obtiveram melhores recompensas comparado com os dados de sequências inteiras. A rede que obteve a maior recompensa foi utilizando 500 neurônios com média de 111,5 pontos. Acredita-se que, quando segmentou os dados

e alimentou a rede, os dados induziam a uma especialização da rede pela repetição de sequências parecidas, porém, a camada convolucional do codificador estava piorando a representação destes frames. Então a rede LSTM recebia uma má representação de um frame (ou algum parecido) diversas vezes. Aquele frame mal representado era especializada, gerando resultados condizentes com a situação.

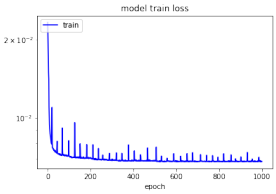
Apesar de se aproveitar da estrutura da imagem, o resultado utilizando o codificador apresentou uma pior performance comparada as redes alimentadas sem as camadas convolucionais. Acredita-se que a camada convolucional não esteja conseguindo extrair informações o suficiente para passar às redes recorrentes. Visualizando a Figura 4.7 os carros mais distantes foram praticamente apagados, isto realmente pode estar prejudicando na decisão das jogadas.

Buscando saber se o modelo conseguiria convergir mais com mais épocas foi desenhado o gráfico das funções de custo da estrutura com 500 neurônios (a estrutura que apresentou o maior média de pontuações).

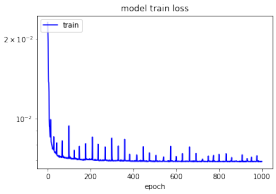
Analisando os gráficos da Figura 4.11 pode-se dizer que a rede teve dificuldades de convergência. Acredita-se que não conseguiu capturar informações o suficientes para a camada LSTM.

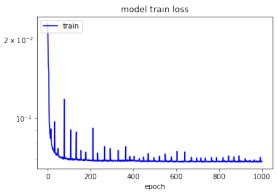


(a) Fold 0

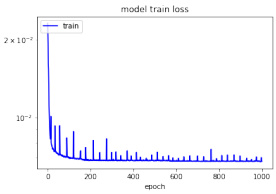


(b) Fold 1

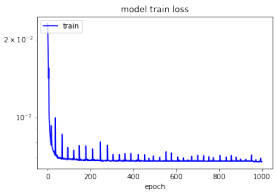




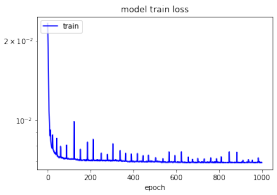
(c) Fold 3



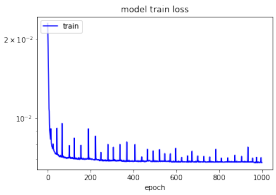
(d) Fold 4



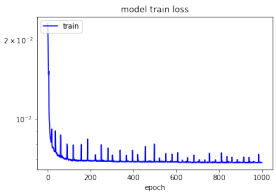
(e) Fold 5



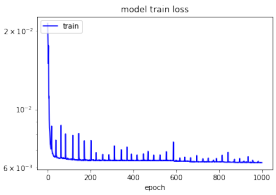
(f) Fold 6



(g) Fold 7



(h) Fold 8



(i) Fold 9

Figura 4.11: Gráficos da perda durante o treinamento do modelo autoencoder + LSTM com 500 neurônios

4.4 Rede Convolucional + LSTM

A fim de avaliar o desempenho utilizando camadas convolucionais foi implementado uma rede com camada convolucional e camada LSTM. Ao tentar usar o *batch size* de tamanho nove, como nos modelos treinados com sequências inteiras até então, houve falta de memória. Portanto o parâmetro do k-fold foi diminuído para k=4. Foi escolhido uma camada densa, após a camada convolucional, de 1.200 neurônios. Apesar do mesmo número de neurônios da rede Autoencoder + LSTM anteriormente analisada, o funcionamento destas duas redes são diferentes.

Nesta rede, a camada densa de 1.200 neurônios é treinada junto com a camada recorrente, esperando deste modo, melhorar seu desempenho. Os resultados obtidos são apresentados na Tabela 4.7.

Tabela 4.7: Recompensa da rede CNN + LSTM alimentado com dados inteiros

neurônios	100	200	500
recompensa média	99,5	69,5	80,5
desvio padrão	28,86	26,25	21,57
tempo de treinamento	60:29:32	65:31:06	84:21:09

Na Figura 4.12 é mostrado o gráfico em boxplot das recompensas pelo número de neurônios da camada escondida.

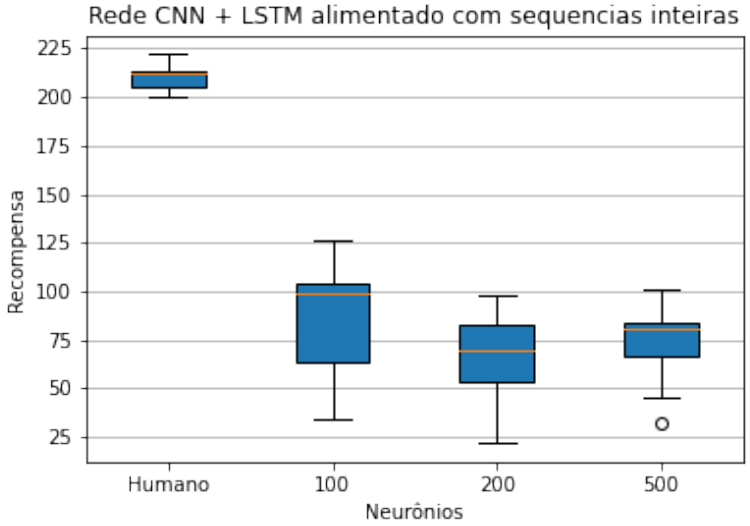
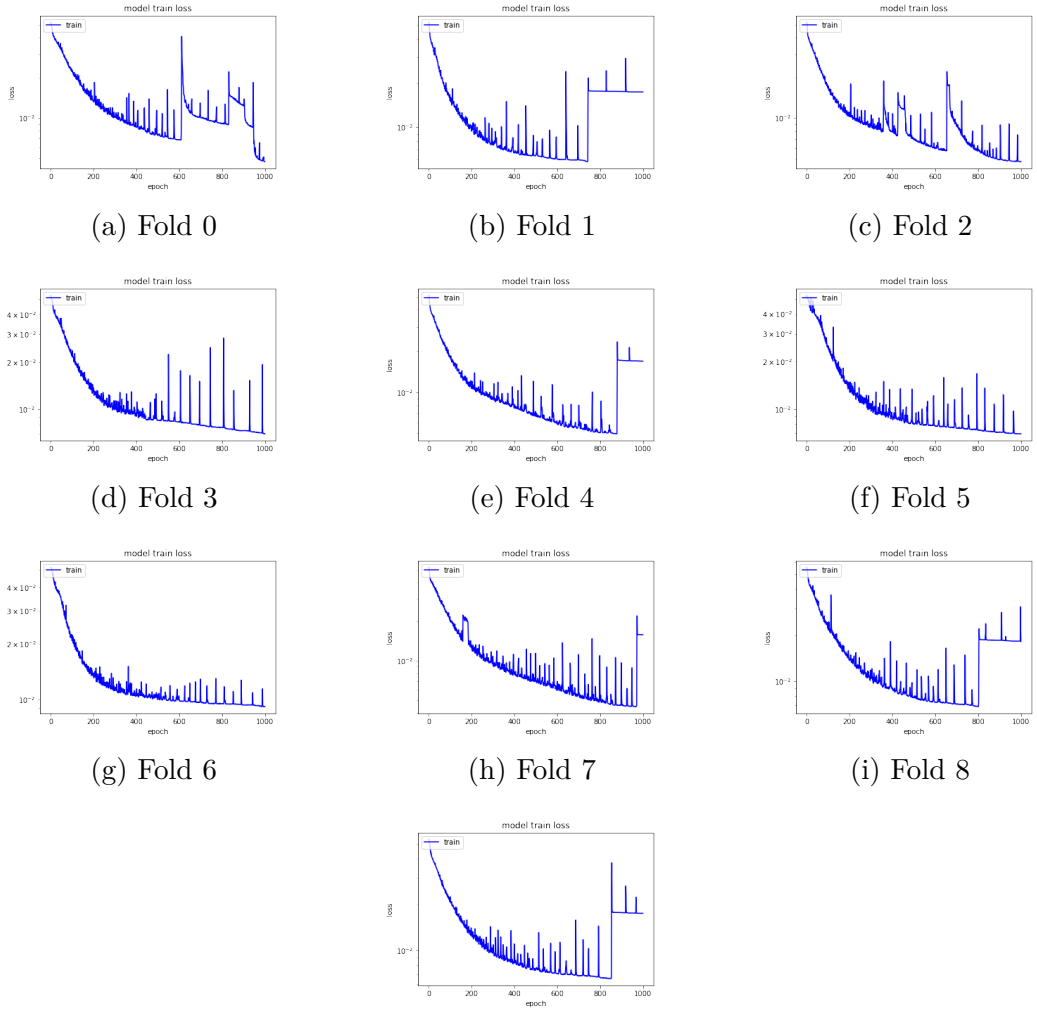


Figura 4.12: Gráfico de recompensas sobre neurônios da rede CNN + LSTM alimentada com sequências inteiras.

A melhor média das recompensas foi encontrada utilizando 100 neurônios. Para entender o treinamento destas rede são apresentados na Figura 4.13 os gráficos de treinamento.



(a) Fold 0 (b) Fold 1 (c) Fold 2

(d) Fold 3 (e) Fold 4 (f) Fold 5

(g) Fold 6 (h) Fold 7 (i) Fold 8

(j) Fold 9

Figura 4.13: Gráfico da perda do treinamento do modelo CNN + LSTM com 100 neurônios

Analisando os gráficos da Figura 4.13, verifica-se que alguns modelos tiveram dificuldades em continuar convergindo devido a um incremento brusco da função de perda. Esse fenômeno foi observado nos *folds* um, quatro, sete, oito e nove. Percebe-se que nos outros *folds* a continuação do treinamento para mais de 10.000 épocas poderia reduzir ainda mais o valor da função de perda ou acarretar o mesmo efeito de salto. Assim, o treinamento desses modelos deve ser cuidadosamente monitorado, sempre salvando a rede com menor função de perda.

Assim como no uso dos dados do codificador do autoencoder, acredita-se que, o modelo não conseguiu extrair informações suficientes pela falta de camadas convolucionais. Outro fator que pode estar prejudicando é a quantidade de neurônios seguida da camada convolucional, podendo aumentar os atuais 1.200 neurônios.

Capítulo 5

Conclusão

Este projeto apresentou um estudo sobre o uso de redes neurais recorrentes rasas, rede de Elman, e profundas, rede LSTM, para o controle de jogos. Como estudo de caso, o jogo Enduro da Atari foi utilizado. Apesar de não ter sido possível treinar a camada convolucional utilizando dados segmentados por conta da limitação do poder computacional e tempo disponível, chegou-se a conclusão que uma camada convolucional simples com apenas 4 filtros não é o suficiente para o algoritmo aprender a jogar Enduro como um jogador humano.

Os resultados mostram que a rede LSTM possui um desempenho melhor que a rede Elman. A média das recompensas da rede LSTM utilizando os mesmos dados da rede Elman foram maiores e teve menor desvio padrão. Tem-se 115,5 pontos e desvio padrão de 16,94 para o melhor modelo da rede Elman, enquanto com a LSTM, a pior rede fez 117,5 pontos e desvio padrão de 11,35.

Uma causa que poderia estar dificultando atingir recompensas maiores, além da simplicidade da rede, seria a falta de dados. Talvez a rede não tenha sido treinada com situações que apareceram nos teste, considerando que o jogador possuía a tendência de ficar mais no meio da pista, faltando dados com o carro nas extremidades, como podem ser vistos nas imagens da Figura 5.1.

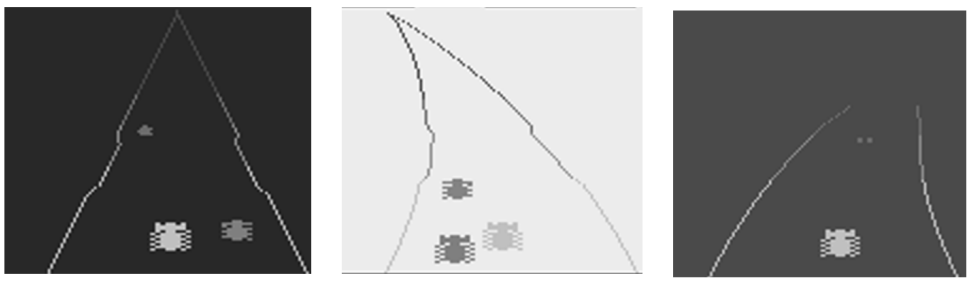


Figura 5.1: Três frames gerados pelo jogador.

Além da característica do jogador, o fundo do jogo muda com o passar das fases, como pode ser visto comparando a primeira e a segunda imagem da Figura 5.1. Durante a fase noturna, visto na terceira imagem da Figura 5.1, a representação do carrinho muda, transformando-se em dois faróis traseiros.

A hipótese da possibilidade de treinar redes neurais recorrentes para jogar jogos digitais foi confirmada, visto que as redes conseguiram ser treinadas. Porém, para a obtenção resultados comparáveis com o jogador humano ou com o estado-da-arte da IA, há a necessidade do projeto de redes neurais mais complexas.

Como trabalhos futuros sugere-se o uso de redes neurais mais profundas e uma exploração maior das redes convolucionais; o uso de um autoencoder mais eficiente aumentando o número de camadas e filtros; verificar as limitações do poder computacional e as GPUs para executar os treinamentos. Em nossos experimentos o Google Colab apresentou momentos de desconexão durante os treinamentos.

Referências

[1] Cesar Augusto de Carvalho. Um Sistema Portátil de Tradução de Posturas do Alfabeto Manual de Libras em Voz Utilizando Luva Instrumentalizada com Sensores IMU. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Computação), Universidade de Brasília, 2019. Disponível em https://bdm.unb.br/handle/10483/26529. x, 5, 7, 8

[2] Ícaro Da Costa Mota. Aprendizagem por Reforço Utilizando Q-Learning e Redes Neurais Artificiais em Jogos Eletrônicos. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Mecatrônica), Universidade de Brasília, 2018. Disponível em http://bdm.unb.br/handle/10483/22068. x, 4, 5

[3] What are Local Minima and Global Minima in Gradient Descent? https://www.i2tutorials.com/what-are-local-minima-and-global-minima-in-gradient-descent/. Online; Acesso: 28-10-2021. x, 8

[4] Mayank Agarwal. Back Propagation in Convolutional Neural Networks — Intuition and Code. https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199. Online; Acesso: 29-10-2019. x, 9

[5] Chris Marriott Ryan Shirley & CJ Baker & Thomas Tannahill. Neural Nets Using Backpropagation. https://slidetodoc.com/neural-nets-using-backpropagation-n-n-chris-marriott/. Online; Acesso: 04-11-2021. x, 10

[6] Gabriella Melki. *Fast Online Training of L1 Support Vector Machines*. PhD thesis, |, 04 2016. x, 11

[7] Diederik P. Kingma e Jimmy Ba. Adam: A Method for Stochastic Optimization. https://arxiv.org/abs/1412.6980, 2017. x, 8, 12

[8] Henrique Branco. Overfitting e underfitting em machine learning. https://abracd.org/overfitting-e-underfitting-em-machine-learning/, 2020. Online; Acesso: 05-11-2021. x, 13

[9] Jeffrey L. Elman. Finding Structure in Time. *Cognitive Science*, 14(2):179–211, 1990. x, 13, 14

[10] Karen Braga Enes. Redes neurais artificiais com processamento temporal - um estudo sobre redes estáticas temporais, redes recorrentes e aplicações. Trabalho de Conclusão

de Curso (Bacharelado em Ciência da Computação, Universidade Federal de Juiz de Fora, 2013. Disponível em http://monografias.ice.ufjf.br/tcc-web/exibePdf?id=68. x, 16, 17

[11] Christopher Olah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Online; Acesso: 04-11-2021. x, 16

[12] About Gisely Alves. Entendendo Redes Convolucionais (CNNs). https://medium.com/neuronio-br/entendendo-redes-convolucionais-cnns-d10359f21184. Online; Acesso: 23-10-2021. x, 18

[13] Jeremy Jordan. Convolutional neural networks. https://www.jeremyjordan.me/convolutional-neural-networks/. Online; Acesso: 03-11-2021. x, 19

[14] Ting-Hao Chen. What is "stride" in convolutional neural network? https://medium.com/machine-learning-algorithms/what-is-stride-in-convolutional-neural-network-e3b4ae9baedb. Online; Acesso: 28-10-2021. x, 19

[15] Ting-Hao Chen. What is "padding" in convolutional neural network? https://medium.com/machine-learning-algorithms/what-is-padding-in-convolutional-neural-network-c120077469cc. Online; Acesso: 23-10-2021. x, 20

[16] Arden Dertat. Applied deep learning - part 3: Autoencoders. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798. Online; Acesso: 23-10-2021. x, 21

[17] Building autoencoders in keras. https://blog.keras.io/building-autoencoders-in-keras.html. Online; Acesso: 04-11-2021. x, 21

[18] Wikipédia. Atari 2600. https://pt.wikipedia.org/wiki/Atari_2600. Online; Acesso: 31-10-2021. x, 21

[19] Rodrigo Leite. Introdução a validação-cruzada: K-fold. https://drigols.medium.com/introduç~ao-a-validaç~ao-cruzada-k-fold-2a6bced32a90. Online; Acesso: 29-10-2021. x, 23

[20] Muzero: Mastering go, chess, shogi and atari without rules. https://deepmind.com/blog/article/muzero-mastering-go-chess-shogi-and-atari-without-rules. Online; Acesso: 31-10-2021. x, 25

[21] Michael Nielsen. Is alphago really such a big deal? https://www.quantamagazine.org/is-alphago-really-such-a-big-deal-20160329/, 2016. Online; Acesso: 05-14-2021. xi, 25

[22] Dota 2 open ai five arena concludes with a 99.4% winrate. https://www.linuxgame.net/post/10. Online; Acesso: 31-10-2021. xi, 26

[23] Openai teaches robot hand to solve rubik's cube using reinforcement learning and randomized simulations, researchers taught this robot how to solve a rubik's cube one-handed. https://spectrum.ieee.org/openai-demonstrates-sim2real-by-with-onehanded-rubiks-cube-solving. Online; Acesso: 31-10-2021. xi, 27

[24] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, e David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265, 2019. xi, 2, 28, 30, 33, 42

[25] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999. 1, 4

[26] Jeff Craighead, Robin R. Murphy, Jennifer L. Burke, e Brian F. Goldiez. A survey of commercial & open source unmanned vehicle simulators. *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 852–857, 2007. 1

[27] Feng-hsiung Hsu Murray Campbell, A.Joseph Hoane. Deep blue. *Artificial Intelligence Volume 134, Issues 1–2, January 2002, Pages 57-83, published by Elsevier*, pages 852–857, 2002. 1

[28] DeepMind. Alphago - the movie | full documentary. https://www.youtube.com/watch?v=WXuK6gekU1Y, 2020. Online; Acesso: 05-11-2021. 1

[29] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, e Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019. 1

[30] Mit deep learning and artificial intelligence lectures. https://deeplearning.mit.edu. Online; Acesso: 19-11-2021. 2

[31] Yu Huang e Yue Chen. Survey of state-of-art autonomous driving technologies with deep learning. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 221–228, 2020. 2

[32] Aman Bhalla, Munipalle Sai Nikhila, e Pradeep Singh. Simulation of self-driving car using deep learning. In *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*, pages 519–525, 2020. 2

[33] Isaac Newton, Daniel Bernoulli, Colin MacLaurin, e Leonhard Euler. *Philosophiae naturalis principia mathematica*, volume 1. excudit G. Brookman; impensis TT et J. Tegg, Londini, 1833. 2

[34] 3Blue1Brown. But what is a neural network? | chapter 1, deep learning. https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Online; Acesso: 28-10-2021. 4

[35] Redes neurais artificiais. https://sites.icmc.usp.br/andre/research/neural/. Online; Acesso: 31-10-2021. 5

[36] Função de perda. https://en.wikipedia.org/wiki/Loss_function. Online; Acesso: 31-10-2021. 7

[37] Victor Jeronymo. Implementando métodos de otimização para treinamento de redes neurais com pytorch. Trabalho como parte dos requisitos para obtenção de créditos na disciplina projeto supervisionado, Universidade Estadual de Campinas (Unicamp, 2019. Disponível em https://www.ime.unicamp.br/~mac/db/2019-2S-157490.pdf. 8, 10

[38] Optimization: Stochastic gradient descent. http://deeplearning.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/. Online; Acesso: 04-11-2021. 10

[39] Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent. Online; Acesso: 04-11-2021. 10

[40] Nicholas Kincaid. Adam. https://optimization.cbe.cornell.edu/index.php?title=Adam. Online; Acesso: 01-11-2021. 11, 12

[41] One Hot. https://en.wikipedia.org/wiki/One-hot. Online; Acesso: 31-10-2021. 13

[42] Recurrent neural network. https://en.wikipedia.org/wiki/Recurrent_neural_network. Online; Acesso: 31-10-2021. 13, 14

[43] Long short-term memory. https://en.wikipedia.org/wiki/Long_short-term_memory. Online; Acesso: 31-10-2021. 15, 16

[44] The nobel prize in physiology or medicine 1981. https://www.nobelprize.org/prizes/medicine/1981/press-release/. Online; Acesso: 31-10-2021. 17

[45] Stanford CS Class. Convolutional neural networks (cnns / convnets). https://cs231n.github.io/convolutional-networks/. Online; Acesso: 03-11-2021. 17

[46] Convolutional neural network. https://en.wikipedia.org/wiki/Convolutional_neural_network. Online; Acesso: 31-10-2021. 17, 18

[47] Leonardo Cardoso Da Cunha. Redes neurais convolucionais e segmentação de imagens – uma revisão bibliográfica. Trabalho de Conclusão de Curso Bacharelado em Engenharia de Controle e Automação, UNIVERSIDADE FEDERAL DE OURO PRETO ESCOLA DE MINAS, 2020. Disponível em https://www.monografias.ufop.br/bitstream/35400000/2872/6/MONOGRAFIA_RedesNeuraisConvolucionais.pdf. 18

[48] Dor Bank, Noam Koenigstein, e Raja Giryes. Autoencoders. https://arxiv.org/abs/2003.05991, 2021. 20

[49] Atari 2600. https://pt.wikipedia.org/wiki/Enduro_(jogo_eletrônico). Online; Acesso: 31-10-2021. 22

[50] Python. https://pt.wikipedia.org/wiki/Python. Online; Acesso: 31-10-2021. 23

[51] Torch Contributors. Pytorch documentation. https://pytorch.org/docs/stable/index.html. Online; Acesso: 31-10-2021. 23

[52] Google. Google colaboratory faq. https://research.google.com/colaboratory/intl/pt-BR/faq.html. Online; Acesso: 31-10-2021. 24

[53] Alphago. https://pt.wikipedia.org/wiki/AlphaGo. Online; Acesso: 31-10-2021. 24

[54] Breakthrough of the year. https://en.wikipedia.org/wiki/Breakthrough_of_the_Year. Online; Acesso: 31-10-2021. 24

[55] Openai five. https://en.wikipedia.org/wiki/OpenAI_Five. Online; Acesso: 31-10-2021. 26

[56] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, e Henryk Michalewski. Model-based reinforcement learning for atari. https://arxiv.org/abs/1903.00374, 2020. 28

[57] Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, e Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019. 28

[58] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, e David Silver. Distributed prioritized experience replay. https://arxiv.org/abs/1803.00933, 2018. 28

[59] RNN. https://pytorch.org/docs/stable/generated/torch.nn.RNN.html. Online; Acesso: 31-10-2021. 35