UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE FÍSICA

CURSO DE GRADUAÇÃO EM FÍSICA

THIAGO LUIZ ALVES COSTA

SOLUÇÃO DA EQUAÇÃO DE SCHRÖDINGER UNIDIMENSIONAL INDEPENDENTE DO TEMPO POR REDES NEURAIS

FORTALEZA

2022

THIAGO LUIZ ALVES COSTA

SOLUÇÃO DA EQUAÇÃO DE SCHRÖDINGER UNIDIMENSIONAL INDEPENDENTE DO TEMPO POR REDES NEURAIS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Física do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Física.

Orientador: Prof. Dr. Jeanlex Soares de Sousa

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C876s Costa, Thiago Luiz Alves.
Solução da Equação de Schrodinger Unidimensional Independente do Tempo por Redes Neurais / Thiago Luiz Alves Costa. – 2022.
55 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Física, Fortaleza, 2022.
Orientação: Prof. Dr. Jeanlex Soares de Sousa.

1. Aprendizado de Máquina. 2. Rede Neural. 3. Mecânica Quântica. 4. Equação de Schrodinger. I. Título.
CDD 530

THIAGO LUIZ ALVES COSTA

SOLUÇÃO DA EQUAÇÃO DE SCHRÖDINGER UNIDIMENSIONAL INDEPENDENTE DO TEMPO POR REDES NEURAIS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Física do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Física.

Aprovada em: 21/07/2022

BANCA EXAMINADORA

Prof. Dr. Jeanlex Soares de Sousa (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. João Milton Pereira Júnior
Universidade Federal do Ceará (UFC)

Dr. Diego Rabelo da Costa
Universidade Federal do Ceará (UFC)

Aos meus pais, Luzanira e Luiz, e à minha noiva, Moara.

AGRADECIMENTOS

Agradeço à minha família, em especial aos meus pais, Luzanira e Luiz, que me ensinaram sobre humildade e me amaram da melhor forma possível.

À minha noiva, Moara, pelo amor incondicional, dedicação e companheirismo que foram essenciais para me manterem de pé e focado nesses últimos meses.

A Deus, pela oportunidade da vida, pela saúde das pessoas que eu amo, e por estar comigo na realização dos meus sonhos.

Ao meu orientador Jeanlex, pelos conselhos e palavras quando eu pensei que não conseguiria concluir o curso, além dos ensinamentos e da paciência.

Às minhas madrinhas de casamento, Virna e Ju, pelo apoio emocional, carinho e dedicação.

Aos meus professores José Ramos e Saulo Reis, por terem sido marcantes na minha caminhada durante esse curso.

Aos meus amigos, Gabriel Silvério e Lucas Sievers, por terem sido meu auxílio no estudo e no desenvolvimento desse trabalho.

RESUMO

Neste trabalho, apresenta-se a solução independente do tempo para equação de Schrödinger unidimensional, tanto por intermédio de métodos numéricos computacionais, quanto pelo uso de algoritmos de aprendizado de máquina, no qual a técnica utilizada foi a resolução por redes neurais. Ao longo do procedimento, ocorreu o teste de hiper-parâmetros para a construção de uma rede neural mais eficiente, na qual foi utilizada para solucionar essa equação dado um potencial novo, visto que seu conjunto de treino foram potenciais gerados aleatoriamente junto à suas respectivas funções de onda. Além disso, houve a comparação entre a solução numérica e a solução por redes neurais para os potenciais mais conhecidos da mecânica quântica, o poço quadrado infinito e o oscilador harmônico simples.

Palavras-chave: aprendizado de máquina; rede neural; mecânica quântica; equação de Schrödinger.

ABSTRACT

In this work, we present the time-independent solution for the one-dimensional Schrödinger equation, both through computational numerical methods and through the use of machine learning algorithms, in which the technique used was the resolution by neural networks. Throughout the procedure, the hyper-parameter test took place to build a more efficient neural network, in which it was used to solve this equation given a new potential, since its training set were potentials generated randomly along with their respective wavefunctions. In addition, there was a comparison between the numerical solution and the solution by neural networks for the known potentials of quantum mechanics, the infinite square well and the simple harmonic oscillator.

Keywords: machine learning; neural network; quantum mechanics; Schrödinger equation.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

DFT	*Density Functional Theory*
DL	*Deep Learning*
IA	Inteligência Artificial
ML	*Machine Learning*
MSE	*Mean Square Error*
NN	*Neural Network*
RN	Rede Neural
SGD	*Stochastic Gradient Descent*
TF	*Tensor Flow*

LISTA DE SÍMBOLOS

y_i	Rótulos
x_i	*Features* de entrada
$\mathbf{X}$	Vetor de instâncias (entradas)
θ_i	Parâmetro de peso para Regressões Linear e Logística
y_p	Função de previsão
h_θ	Função de hipótese
σ	Função sigmoide
f_A	Função de ativação
ω_{ij}	Parâmetro de peso para Redes Neurais
b	Parâmetro de viés
J	Função custo
η	Parâmetro de aprendizagem

SUMÁRIO

1 INTRODUÇÃO

O *Machine Learning* é a subárea da Inteligência Artificial (IA), o qual baseia-se em algoritmos computacionais que aprendem características novas ao serem expostos em conjuntos de dados. Essa ferramenta, por sua vez, abrange um conjunto de técnicas de previsão e de classificação que vão desde redes neurais artificias até árvores de decisão e *Deep Learning*, (SOUZA, 2020). Dessa forma, atualmente notam-se grandes incentivos relacionados a essa área, visto que suas utilidades se tornaram mais comuns no dia a dia e foram impulsionadas pela evolução tecnológica.

As primeiras esquematizações sobre o que seria o Aprendizado de Máquina surgiram com a analogia de comparação com o sistema nervoso do cérebro humano, na qual persistia o questionamento se seria possível fazer com que a máquina pudesse manejar informações e aprender com elas, tal como o cérebro humano é capaz. Essa correlação foi bastante útil para a visualização e o entendimento de como os códigos iriam fluir em um grafo de informações. No entanto, a maneira como essa ideia foi divulgada junto à falta de informação, corroborou para que essa área passasse por um período de falta de investimento e de *déficit* de pesquisas na década de 80, pois foi criada uma alta expectativa do que poderia ser feito sem a análise da força tecnológica disponível.

Apesar disso, o auxílio da evolução tecnológica e o aumento do conjunto de abordagens criadas faz com que, hoje em dia, essa área receba um quantidade muito maior de investimentos, de grandes empresas até microempresários, os quais buscam melhorar a eficiência na abordagem de clientes e no ganho de capital financeiro. Com isso, as aplicações atuais do *Machine Learning* representam um conjunto de possibilidades, desde as mais simples e diárias até aplicações na descoberta de exoplanetas, (JIN; CHIANG, 2022). Portanto, percebe-se a importância dessa área e a necessidade de seu estudo, visto que as informações estão cada vez mais rápidas, sendo preciso o treinamento da máquina com o intuito de interpretá-las.

Dentro do conjunto de modelos aplicados em *Machine Learning*, existe o das redes neurais artificiais, que formam um subconjunto dentro do Aprendizado de Máquina. Esses modelos computacionais recebem esse nome, pois foram caracterizados com a intenção de se assemelhar às ligações entre neurônios biológicos de um sistema nervoso. Dessa forma, uma rede neural recebe em sua nomenclatura nomes como: 'neurônios', para representar uma nó de ativação por onde ocorrem os cálculos numéricos; 'camadas', as quais representam a profundidade da rede e sua complexidade; '*percéptrons*', redes neurais formadas por um único

neurônio. Essas nomenclaturas são didáticas, pois auxiliam no entendimento de como é criada a arquitetura de uma rede neural artificial dentro da máquina.

Outro subconjunto importante é o *Deep Learning*, que utiliza redes neurais artificiais para a resolução de problemas complexos, (IMPORTANCE. . . , 2022). O *Deep Learning* não se limita às redes neurais simples, e abrange um conjunto de técnicas que buscam otimizar os parâmetros utilizados para melhorar a generalização de um determinado modelo. Dessa forma, esse subconjunto surgiu com o intuito de utilizar o máximo de rendimento computacional disponível na atualidade, para solucionar problemas com alto teor de complexidade.

Posto isso, os algoritmos de *Deep Learning* são bastante úteis no estudo das soluções numéricas de equações complexas ou não-lineares, por utilizarem funções internas também não linearizadas. Consequentemente, utilizam-se esse modelos em diversos campos de atuação da Física, como na Física de Sistemas Complexos e na Física Quântica, além de serem aplicados em ramificações relacionadas à Química, Biologia e Medicina, (ESTEVA; RAMSUNDAR, 2019). Portanto, um estudo específico será abordado ao decorrer desse trabalho, na aplicação das técnicas de *Deep Learning*, com redes neurais artificiais de alta dimensionalidade para encontrar as funções de onda do estado fundamental da equação de Schrödinger independente do tempo.

2 APRENDIZADO DE MÁQUINA E REDES NEURAIS

O Aprendizado de Máquina, conhecido também como *Machine Learning* (ML) pertence ao ramo da inteligência artificial e da ciência da computação, o qual utiliza bancos de dados e algoritmos de aprendizagem que são treinados para o melhoramento gradual de sua precisão, a fim de prever ou classificar conjuntos de dados futuros, revelando características na mineração desses dados. Ademais, pontua-se que o surgimento dessa área de ramificação teve sua base no interesse de constatar a funcionalidade das máquinas na utilização de dados para a automação de sua aprendizagem, sendo estudada por pesquisadores de inteligência artificial. Dessa forma, nos últimos cinquenta anos, o uso de algoritmos de ML evoluiu desde o surgimento da automatização em jogos de xadrez até o desenvolvimento de carros autônomos, mecanismos de recomendação e detecção de fraudes, (GERON, 2019).

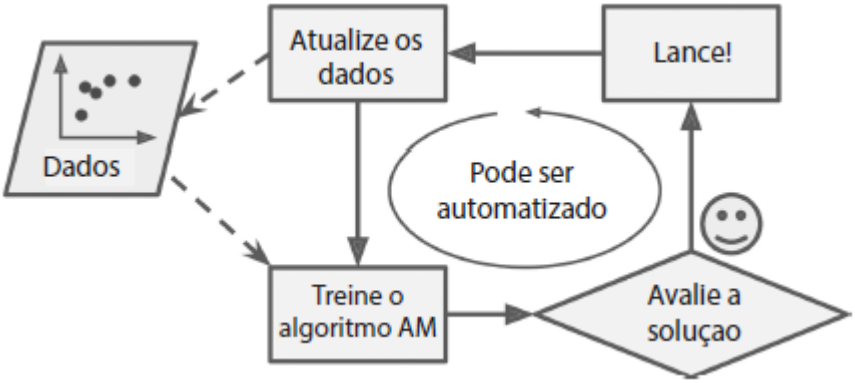

Figura 1 – Visualização do algoritmo de *Machine Learning*. Imagem retirada de (GERON, 2019).

Em meados da primeira década dos anos 2000, houve uma evolução nos algoritmos de *Machine Learning* e um aumento na utilização desses métodos, por causa do advento da internet, a qual trouxe consigo um volume muito maior de dados disponíveis comparados aos anteriores, em decorrência da melhoria dos processadores e das placas de armazenamento presentes nos novos computadores construídos para que suportassem essa grande quantidade de dados, gerando o surgimento de novas aplicações de ML. Com isso, encontraram-se diversas utilidades para essa nova forma de programação, como a utilização do Aprendizado de Máquina para a classificação de mensagens *spam* nos correios eletrônicos, a automação do preenchimento em textos digitais e a identificação de rostos em aparelhos com câmera, (GERON, 2019). O uso do ML cresceu e o seu envolvimento tornou-se notório nas áreas de automação, processamento de imagens e de linguagem natural, ciências sociais, químicas e biológicas. Além disso, a

utilização desses modelos evoluiu para o estudo mais aprofundado da Física e Química Quântica levando a algumas aplicações, por exemplo: aprendizado da densidade eletrônica datada pelo uso do *Density Functional Theory* (DFT), cálculo de barreiras energéticas de compostos químicos, cálculo da função de onda e das funções de base e aprendizado aplicado nas propriedades de materiais, como no estudo de materiais bidimensionais, (SCHLEDER; FAZZIO, 2021).

2.1 Estratégias e métodos do Aprendizado de Máquina

A estratégia de pré-abordagem de um problema envolvendo o uso do *Machine Learning* é feita seguindo 4 etapas principais: (*i*) enquadrar o problema, (*ii*) buscar uma medida de referência, (*iii*) selecionar um modelo de desempenho e (*iv*) verificar a hipótese. Nesta abordagem, o algoritmo é treinado com a utilização de um conjunto de dados conhecidos, chamado de 'conjunto de treino', e de uma característica de interesse que depende dos valores desse conjunto. Os resultados (*outputs*) desse código irão treinar a máquina para gerar previsões y_p em um novo conjunto de dados, tal que a sua capacidade de prevê-los corretamente é chamada de 'generalização do modelo'. Após isso, ocorre a escolha do tipo de aprendizado e do método que será utilizado na abordagem do problema, tendo em vista a identidade do conjunto de dados, o preço computacional e o tempo gasto para resolvê-lo. Dessa forma, o algoritmo do Aprendizado de Máquina pode ser dividido em quatro tipos de aprendizagem, de acordo com a identidade dos dados de treino, que são: Aprendizado Supervisionado, Aprendizado Não-Supervisionado, Aprendizado Semi-Supervisionado e Aprendizado por Reforço, (GERON, 2019).

Além disso, após a análise dos dados e a verificação de quais tipos de abordagem utilizadas, deve ser escolhido o tipo de modelo que otimize o custo operacional junto ao tempo de resolução do problema. Com isso, os algoritmos de previsão ou classificação que serão citados nesse trabalho, são: Regressão Linear, Regressão Logística e Redes Neurais.

2.1.1 Aprendizado Supervisionado

Ocorre quando o algoritmo é treinando por um conjunto de dados rotulados, onde o termo 'rótulo' é o resultado desejado de um determinado dado de entrada, por exemplo: seja um conjunto de amostras que expressa os preços de imóveis e a quantidade de quartos construídos em cada imóvel, e deseja-se prever o preço de outros com base nas suas quantidades de quartos, nesse caso, o valor do imóvel é o rótulo, a quantidade de quartos é o *feature* e o imóvel é a

entrada. Com isso, um conjunto de dados (ou amostras) é rotulado, quando possui entradas que são relacionadas com saídas desejadas previamente conhecidas, fazendo com que a máquina preveja novos rótulos em um novo conjunto de dados. Além disso, a utilização desse tipo de abordagem pode ocorrer por meio de modelos como: Regressão, Classificação e Redes Neurais. Portanto, percebe-se a sua utilidade em previsões de eventos futuros, com base em dados de histórico.

2.1.2 Aprendizado Não Supervisionado

Tal abordagem ocorre à medida que o conjunto de treino se torna não-rotulado, ou seja, a 'resposta correta' para um dado de treino não é dita ao sistema. Com isso, a máquina é forçada a explorar os dados, de forma aleatória, e buscar características (*insights*) que sejam úteis para a resolução do problema. Dessa forma, esse tipo de aprendizado é utilizado em modelos de agrupamento (ex: agrupamento *k-means*), de recomendação de itens e de *Autoencoders* (Rede Neural (RN) para aprendizado não supervisionado).

2.1.3 Aprendizado Semi-Supervisionado

Esse tipo de análise ocorre quando o conjunto de treino é parcialmente rotulado e não-rotulado, isso se deve pela existência de um custo de rotulação, no qual torna inviável a rotulação completa do conjunto de dados. Com isso, a máquina aprende alguns padrões e é forçada a explorar o restante dos dados aplicando esses parâmetros de padronização. Dessa forma, esse tipo de aprendizado pode ser aplicado em modelos semelhantes ao Aprendizado Supervisionado, e é utilizado na identificação de rostos por *webcam*.

2.1.4 Aprendizado por Reforço

Esse tipo de abordagem é diferente das anteriores, pois utiliza dados de treino variáveis, e dados de testes do tipo "tentativa e erro"para descobrir qual ação rende mais recompensa em determinado problema. Com isso, a máquina é treinada com três classes: agente (quem faz a ação), ambiente (local onde a ação é feita) e ação (gerador de recompensas), onde o objetivo é a escolha da ação que maximize a recompensa esperada em determinado intervalo de tempo. Dessa forma, esse tipo de aprendizado é normalmente aplicado em robótica, jogos e meios de navegação pela internet.

2.1.5 Regressão Linear

A regressão linear é um modelo aproximativo de *Machine Learning*, no qual utiliza dados de treino rotulados e gera uma relação linear entre os rótulos e as entradas, (y_i) e (x_i), respectivamente. O vetor formado pelos dados de entrada é chamado de vetor de instâncias ($\mathbf{X}$). Nesse aspecto, para um conjunto de amostras $\{x_i, y_i\}_{i=1}^{N}$, o algoritmo recebe as entradas x_i e cria esse vetor X no formato de *array* (definição de uma lista em *Python*), onde i representa a localização dos dados de entrada x_i dentro dessa lista. Além disso, deve-se adicionar um termo na primeira posição (índice $i = 0$) do *array*, chamado de "termo de viés"(x_0), por causa da expressão que relaciona o rótulo de previsão com o vetor de instâncias.

$$y_p = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_N x_N \tag{2.1}$$

Essa relação expressa a função de previsão y_p em termos do produto entre x_i com os parâmetros θ_i, nos quais representam os pesos de cada uma das entradas. Ao reescrever essa equação, de forma vetorial, encontra-se que:

$$y_p = h_\theta(\mathbf{X}) = \Theta^T \cdot \mathbf{X}, \tag{2.2}$$

onde Θ é o vetor de parâmetros do modelo e $h_\theta(\mathbf{X})$ é a função hipótese, na qual se expressa pelo produto escalar entre o vetor de parâmetros e o vetor de instâncias.

Além disso, utiliza-se uma função que dita a medida de desempenho do modelo, ou seja, expressa quão bom ou ruim modelo se adapta ao conjunto de treino, assim deve-se encontrar os parâmetros que minimizem essa medida. Dessa forma, para o caso da regressão linear, a função que dita o desempenho é o Erro Quadrático Médio (do inglês *Mean Square Error* (MSE)).

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - y_p^{(i)})^2 \tag{2.3}$$

2.1.6 Regressão Logística

A regressão logística é outra técnica de *Machine Learning*, que utiliza a abordagem de Aprendizado Não Supervisionado e é muito útil na classificação de dados de entrada. Essa técnica consiste no cálculo da probabilidade de uma determinada instância pertencer a alguma classe imposta ao algoritmo. Com isso, seja um conjunto de amostras $\{x_i, y_i\}_{i=1}^{N}$, em que os

valores de y_i só apresentam dois resultados, que são identificados como classes. Dessa maneira, calcula-se a probabilidade de cada x_i pertencer a uma dessas duas classes. Para esse cálculo, utiliza-se a função sigmoide $\sigma(\mathbf{z})$ como hipótese, definida como:

$$h(z) = \sigma(z) = \frac{1}{1+e^{-\mathbf{z}}}, \tag{2.4}$$

onde $\mathbf{z}$ é produto escalar entre o vetor de parâmetros e o vetor de instâncias. Dessa forma, o cálculo das previsões para as determinadas classes é feito da seguinte maneira:

$$y_p(yi = classeA; \mathbf{X}) = \sigma(\theta^T \cdot \mathbf{X}), \tag{2.5}$$

$$y_p(yi = classeB; \mathbf{X}) = 1 - \sigma(\theta^T \cdot \mathbf{X}). \tag{2.6}$$

2.2 Redes Neurais

A Rede Neural, conhecida também como *Neural Network* (NN) é uma técnica de criação de modelos, na qual pode ser treinada com dados rotulados e não rotulados (ex: modelos de *Autoencoders*), e apresenta uma conceituação computacional que tenta se assemelhar aos ligamentos presentes no sistema nervoso biológico. Com isso, define-se o neurônio como a unidade fundamental de uma NN, no qual é capaz de receber dados de entrada (*inputs*) e retornar dados de saída (*outputs*). Dessa forma, a criação desse modelo utiliza como parâmetro ajustável ou hiper-parâmetro, a quantidade de neurônios que serão necessários para a resolução do problema e a quantidade de camadas, desde a utilização de uma única célula em uma única camada (*Percéptron*) até centenas dessas (ex: *Deep Learning*).

A divisão de camadas presentes em uma *Neural Network* segue a seguinte nomenclatura, na qual refere-se à primeira camada como *input layer* que é responsável por receber os *inputs* e transformá-los em representações internas, no formato de múltiplos *arrays* que são chamados, na ciência da computação, de tensores. Após isso, esses tensores são recebidos pelas camadas intermediárias, chamadas de *hidden layers*, nas quais a quantidade dessas camadas é um parâmetro ajustável e sua função é transformar as entradas e enviá-las para a última camada. Portanto, essa última camada, chamada de *output layer*, retorna os dados de saída finais e finaliza o processo computacional de uma Rede Neural.

Ademais, é importante entender que esse modelo computacional geralmente ocorre por duas fases, utilizando conjuntos de dados distintos em cada uma delas. Posto isso, define-se a

primeira fase como “fase de treino”, onde a Rede Neural utiliza de uma parte aleatória dos dados (conjunto de treino) para treinar a eficiência e verificar as imprecisões de seu modelo. Após isso, condiciona-se o protótipo para a “fase de teste”, na qual utiliza-se um conjunto de dados inéditos para a máquina (conjunto de teste), com a finalidade de concluir se o modelo consegue encontrar as respostas esperadas em novos dados. Ao passar por essas fases, é concluído se a generalização da Rede Neural é precisa ou não, cabendo assim se há necessidade da utilização de novas funções de otimização ou de mudança nos hiper-parâmetros.

Nesse aspecto, se o modelo de NN possuir alta acurácia na “fase de treino”, mas for ineficaz no conjunto de teste, então classifica-se que o protótipo sofreu *Overfitting*, ou seja, é eficaz somente para um conjunto específico de dados e não pode ser generalizado. Caso o contrário, se acurácia for muito baixa nas duas fases, ou seja, o modelo não prever de forma correta os dois conjuntos, então classifica-se como *Underfitting*, onde também é ineficaz para a generalização do código. Portanto, nota-se que o objetivo principal na criação de códigos em *Machine Learning*, é a busca pelo equilíbrio entre essas duas classificações, a fim de obter uma Rede Neural que possa ser generalizada em novos conjuntos de dados.

O *Deep Learning* (DL), também conhecido como Aprendizado Profundo, faz parte de uma subseção do *Machine Learning* junto às Redes Neurais, e é criado com a implementação de redes que possuem grandes quantidades de camadas e de células de ativação (neurônios), e tem a finalidade de trabalhar em problemas com maior complexidade e maiores conjuntos de dados, utilizando um maior preço computacional comparado aos algoritmos mais simples. Com isso, o início da implementação de DL ocorre com a escolha do tipo de camada, na qual cita-se a mais utilizada: *Dense Layer*. Essa categoria caracteriza-se por ser totalmente conectada, pois cada neurônio de cada uma das camadas está conectado como todos os outros das adjacentes, onde os dados processados tornam-se os *outputs* da camada atual e os *inputs* das próximas.

A taxa de aprendizado de Redes Neurais ocorre pela utilização de parâmetros de peso ω, que são ajustados enquanto a rede está em funcionamento. Esses pesos são análogos aos parâmetros θ da Regressão Linear, na equação (2.1), e ditam a importância que cada *input* possui para a obtenção do *ouput*. Desse modo, para o modelo de *Percéptron* (rede com um único neurônio), o cálculo feito para a obtenção da função de previsão (y_p) é mostrado como:

$$y_p = f_A(\omega^T \cdot \mathbf{X} + b) = f_A(z), \tag{2.7}$$

em que a função de ativação f_A é aplicada no resultado da multiplicação entre o vetor transposto de *weights* e o vetor de instâncias $\mathbf{X}$, somado ao vetor de *bias* análogo a θ_0.

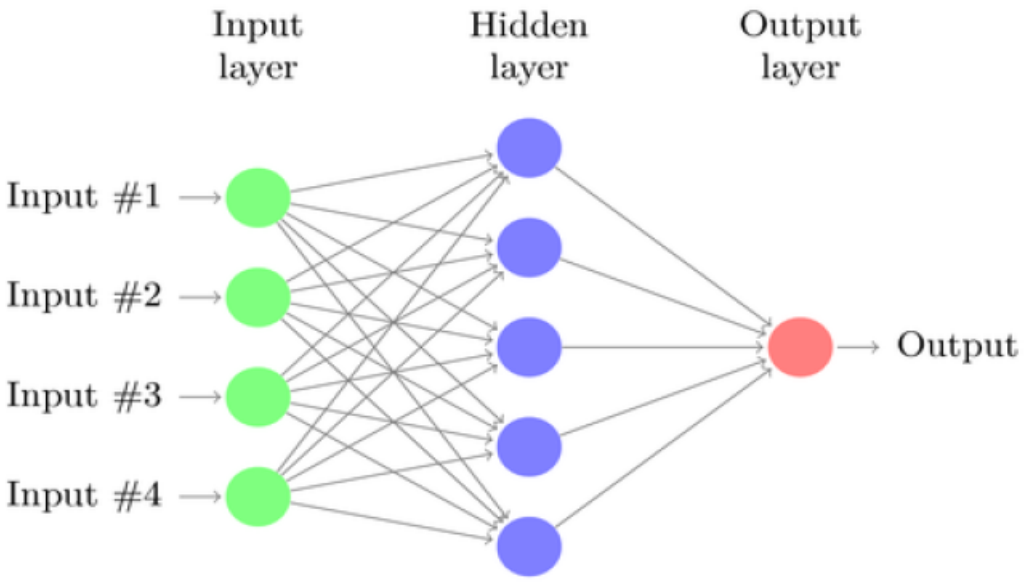

Figura 2 – Diagrama de uma Rede Neural do tipo *Dense Layer*, com uma *hidden layer* . Imagem retirada de (KAPUR, 2017)

A finalidade da função de ativação, presente nos algoritmos de NN, é introduzir termos não-lineares ao modelo e aumentar a sua complexidade, pois caso não houvesse a implementação desse parâmetro, a Rede Neural somente iria relacionar os dados em funções lineares. Dessa forma, o uso desses elementos é de extrema importância para a generalização do problema envolvendo esses modelos de *Machine Learning*.

Além disso, na criação de um programa de Rede Neural, deve-se definir alguns hiper-parâmetros e algumas funções que ajudarão na convergência dos valores obtidos, como o otimizador, a função custo e o número de *epochs*. A função custo, também chamada de $lossfunction$, é um importante medidor da eficácia do modelo, pois dita a distância entre os valores de previsão e os valores esperados, onde quanto menor for essa função, mais eficiente é o programa. Dessa forma, o objetivo principal no treinamento de uma NN é minimizar a *loss function*, que ocorre por intermédio do algoritmo de gradiente descendente estocástico (stochastic grandient descent).

O parâmetro do número de *epochs* está relacionado com a quantidade de vezes que o programa interage com todos os dados de entrada presentes no conjunto de treino. Sendo assim, esse fator é útil para a atualização dos pesos ω do modelo.

Por fim, como já dito anteriormente, o objetivo principal na criação de um modelo por Redes Neurais é a minimização da função custo, pela utilização do método de Gradiente Descendente Estocástico (*Stochastic Gradient Descent* (SGD)). Sendo assim, analiticamente, o gradiente de uma função é um vetor que aponta para a máxima variação em um determinado

ponto no espaço, então com intuito de encontrar os pontos de menor variação (minimização) dessa função, os pesos se movem para o sentido contrário do gradiente. No entanto, a mudança causada nesses pesos, com o intuito de minimizar a *loss function*, é otimizada por intermédio de um método chamado *Backpropagation*.

O uso desse método auxilia na obtenção de resultados mais eficientes para redes mais complexas, e ocorre após obter os resultados das funções de previsão. Ao encontrar esses valores junto aos pesos, o programa aplica esses parâmetros de volta no código, mas seguindo o sentido contrário (*outputs* para *inputs*), com o intuito de verificar se a função custo aumentou ou diminuiu.

2.3 Como uma Rede Neural funciona

Como já dito anteriormente, uma Rede Neural recebe um conjunto de treino, calcula os pesos que minimizam a função custo e retornam *outputs* que serão verificados com o valor esperado, a fim de serem generalizados para um conjunto novo de dados. Sendo assim, ao iniciar esse programa, os pesos são gerados de forma aleatória e é feito o produto escalar entre vetor desses parâmetros com o vetor de *inputs* gerando um resultado que é somado ao vetor de *bias*. Dessa forma, esse *output* encontrado passa para a outra camada, e o processo se repete com os *weights* e *bias* respectivos de cada *layer* até que chegue na camada de saída.

Nesse aspecto, nota-se que os parâmetros de pesos e de viés (*bias*) são referentes à cada camada, e contêm as informações de aprendizagem do modelo para o conjunto de treinamento. Com isso, sabe-se que esses valores não podem ser manuseados pelo programador e são feitos pela máquina, mas existem hiper-parâmetros (parâmetros mutáveis) que auxiliam na otimização e no controle desses pesos. Dessa forma, outro tipo de hiper-parâmetro é a taxa de aprendizagem, também chamada de *learnig rate*, que controla a mudança dos pesos e das *bias*.

A Figura (3) exemplifica para uma Rede Neural Simples, como os parâmetros de *weights* (ω) e de *bias* (b) se relacionam com os dados de entrada, com o intuito de fornecer os dados de saída, ou seja, a função de previsão.

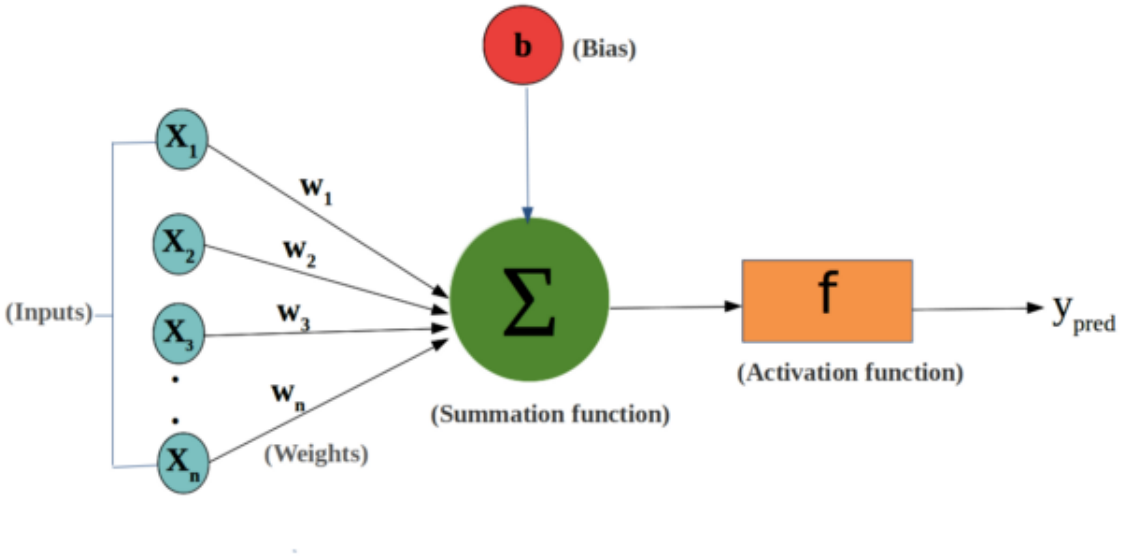

Figura 3 – Diagrama de uma Rede Neural com *weights* e *bias*. Imagem retirada de (KAPUR, 2017).

O hiper-parâmetro *learning rate* (η), geralmente é definido como um número entre 0 e 1, e se relaciona aos parâmetros acima por meio das seguintes equações:

$$\omega_{ij}^{n} = \omega_{ij}^{n-1} + \eta \frac{\partial J}{\partial \omega_{ij}^{n}}, \tag{2.8}$$

$$b^{n} = b^{n-1} + \eta \frac{\partial J}{\partial b^{n}}, \tag{2.9}$$

onde os índices i e j representam as células de ativação (neurônios) que estão conectadas, no caso de ω_{ij}, interpreta-se que a operação esteja sendo passada da célula i pertencente à camada anterior $(n-1)$ para a célula j que pertence à camada atual (n). Além disso, o termo (J) representa a função custo e de forma análoga, a sua derivada parcial em relação a algum dos parâmetros representa o gradiente descendente dessa função.

No entanto, a escolha de um número grande para os hiper-parâmetros relacionados com a aprendizagem, nem sempre permite que os resultados possam convergir para os valores esperados, pois a relação entre esses hiper-parâmetros e a complexidade do problema não é linear. Por isso, a escolha desses valores deve ser testada, com a finalidade de reduzir a função custo, e de forma prudente, conseguir um tempo de resolução e um custo computacional viáveis para a resolução do problema em questão.

2.4 Aprendizado de Máquina utilizando TensorFlow

O *Tensor Flow* (TF) é uma biblioteca gratuita de código aberto ajustada para o desenvolvimento de cálculos numéricos e para o treinamento de redes neurais complexas, na qual foi desenvolvida e disponibilizada pela equipe do Google Brain (INTRODUÇÃO. . . , 2017) em fevereiro de 2017. Desde então, tornou-se bastante utilizada pela comunidade de programadores em *Machine Learning*, por causa da sua similaridade com a linguagem utilizada em *Python* e da sua simplificação nas linhas de código. Portanto, nota-se a importância dessa biblioteca para o avanço dos estudos relacionados à *Deep Learning*.

A implementação de problemas no TensorFlow é feito com a utilização de seu *framework*, no qual possibilita ao programador que não seja necessário implementar o código totalmente do zero. Essa característica é inovadora, pois proporciona a criação de redes neurais em poucas linhas de código, sem a utilização de funções analíticas explícitas, como a função de custo ou a função de ativação.

Ao construir um código nessa biblioteca, deve-se utilizar o termo "tf.Session()" para abrir uma sessão, na qual vai iniciar a propagação do modelo e criar um "grafo computacional". Dessa maneira, esse grafo representa o fluxo de dados presentes no código e gera *insights* que podem ser analisados, por meio do componente chamado de TensorBoard. Dessa forma, esse componente do TF gera gráficos que relacionam características e detalhes sobre o modelo criado.

A criação de uma rede neural pode ser implementada, de forma simples, pelo TensorFlow da seguinte maneira: Define a quantidade de neurônios pertencentes à camada de *input*, o número de camadas internas e a quantidade de neurônios dessas camadas, e a camada de *output* com suas respectivas células de ativação. Após isso, os parâmetros de peso e de viés serão definidos, de forma aleatória, por meio da função "tf.Variable(tf.random_normal())" na qual relaciona as camadas onde esses pesos serão aplicados.

```
# Pesos da camada 1
w1 = tf.Variable(tf.random_normal([n_input, n_hidden_1]))
# Bias da camada 1
b1 = tf.Variable(tf.random_normal([n_hidden_1]))
# Aplicando a função sigmoide na camada 1
camada_1 = tf.nn.sigmoid(tf.add(tf.matmul(x,w1),b1))
```

Figura 4 – Implementação de uma Rede Neural Simples utilizando a linguagem do TensorFlow. Imagem retirada de (O. . . , 2022).

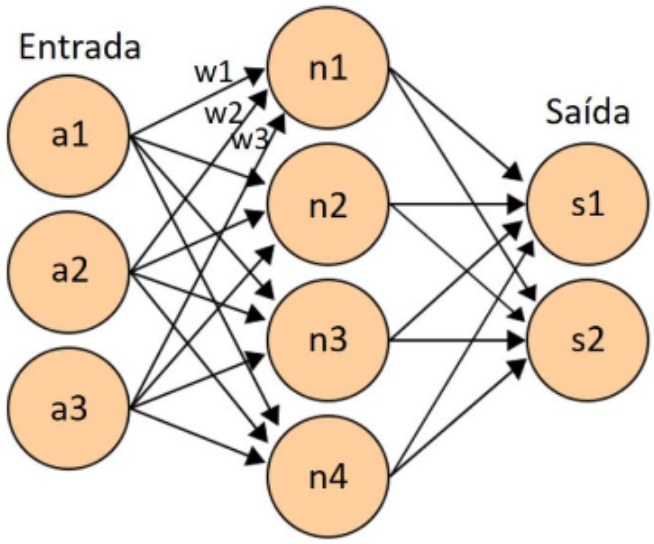

Figura 5 – Representação da Rede Neural Simples implementada pelo código de TensorFlow acima. Imagem retirada de (O. . . , 2022).

Dessa maneira, na Figura (4), ocorre a criação de uma rede neural com uma camada interna, na qual sua função de ativação é definida como a função sigmoide. Com isso, após a implementação do código acima, cria-se um grafo computacional de uma rede com três células na camada de *inputs*, quatro na camada interna e duas células na camada de *outputs*. Esses resultados são divididos em duas classes, pois a função de ativação (σ) retorna valores em 0 ou em 1. Dessa forma, percebe-se que esse algoritmo trabalha sobre o modelo de Regressão Logística, no qual foi estudado anteriormente, e é útil em problemas de classificação.

3 A EQUAÇÃO DE SCHRÖDINGER

No conceito da Física Clássica, sabia-se que o comportamento de uma partícula podia ser analisado verificando sua trajetória ao passar de um determinado intervalo de tempo, e então os parâmetros de estudo eram as posições e a velocidade em quaisquer momentos do tempo. No entanto, em contraste a essa ideia, na formulação da Mecânica Quântica, passou-se a referir ao estado de uma partícula como sendo a representação de sua função de onda, a qual é responsável por possuir todas as informações dessa partícula em um sistema quântico.

Com isso, houve o surgimento de diversas representações a fim de explicar de forma mais clara as propriedades e a identificação dessa função de onda. Dessa forma, tem-se a representação da função de onda por $\psi(\vec{r})$, na qual é descrita no espaço de coordenadas Euclidiano (ε), diferentemente da notação de Dirac, em que define-se a representação do *ket* $|\psi\rangle$ como o vetor de estado, caracterizando cada estado quântico da partícula em um espaço complexo, chamado de espaço de Hilbert (Γ). Logo, é equivalente representar a função de onda $\psi(\vec{r}) \in \varepsilon$ como $|\psi\rangle \in \Gamma$, tal que a sua interpretação física é a amplitude probabilística da posição dessa partícula em um determinado instante de tempo. A função de onda no espaço de Hilbert, ou espaço de auto-estados, é representada como uma combinação linear de um conjunto de bases $|\phi_i\rangle$, tal que se esse conjunto for contínuo em Γ, então ocorre que o quadrado da norma da função de onda fornece a interpretação da densidade de probabilidade da partícula. Além disso, define-se nesse espaço que o vetor de estado $|\psi\rangle$ é o *autoket* do operador A, se a seguinte relação for cumprida: $A|\psi\rangle = \lambda|\psi\rangle$, onde λ é denominado o autovalor de A e a relação descrita é chamada de **equação de autovalor**.

Outra propriedade importante de ser estudada, quando se trata sobre funções de onda é a probabilidade. Assim, seja P a probabilidade de encontrar uma partícula, no conjunto de bases contínuo em Γ, tal que se esse espaço forma um volume τ, então existe um elemento infinitesimal $d\tau$ que compõe a relação definida como: $dP = C \parallel \langle\psi,\psi|\psi,\psi\rangle \parallel^2 d\tau$, em que C é uma constante de normalização. Dessa forma, sabendo que a probabilidade de encontrar uma única partícula ao percorrer todo o espaço é 1, então tem-se que:

$$\int_{\Gamma} dP(\vec{r},t) = \int_{\varepsilon} \psi^* \psi d\tau = 1, \tag{3.1}$$

onde a constante C se torna 1, por conta da função de onda normalizada. Com isso, nota-se que a equação que ao ser interpretada como a probabilidade de encontrar a partícula sobre o volume que inclui possíveis estados de sua localização, é encontrada ao calcular $\parallel \langle\psi,\psi|\psi,\psi\rangle \parallel^2$ ou

$\psi^* \psi$, na qual é chamada de densidade de probabilidade ou condição de normalização.

De acordo com o sexto postulado da Mecânica Quântica, a equação de Schrödinger é responsável por governar as propriedades da evolução temporal da função de onda $\psi(t_0)$, quando definido no tempo fixado t_0, junto ao operador Hamiltoniana $H(t)$, no qual está associado à energia total do sistema. Com isso, a equação de Schrödinger segue a definição da equação de abaixo:

$$H\psi = E\psi, \tag{3.2}$$

onde os resultados das energias em E formam um espectro de autovalores de H, (COHEN-TANNOUDJI *et al.*, 2020). Na Mecânica Clássica, definia-se a função Hamiltoniana ($\mathscr{H}$) como a soma das contribuições de energia cinética (T) e de energia potencial (V), no sistema de partículas. No entanto, ao reescrever a Hamiltoniana para a matemática utilizada na Mecânica Quântica, transforma-se a função em um operador que atua em uma determinada função de onda.

$$\mathscr{H} = \frac{\vec{P}^2}{2m} + V(\vec{r},t) \tag{3.3}$$

A representação da função $\mathscr{H}$ como o operador Hamiltoniana, necessita-se da transformação do *momentum* no operador Momento (**P**) e ao implementar na equação (3.3), tem-se que:

$$\mathbf{P} = P_x + P_y + P_z = -i\hbar \left(\frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z} \right). \tag{3.4}$$

Logo, o operador Hamiltoniana pode ser reescrito da seguinte forma:

$$H = -\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r},t), \tag{3.5}$$

onde ∇^2 é o Laplaciano nas coordenadas x, y e z, que opera como a soma das derivadas de segunda ordem no autovetor na função ψ. Como isso, a equação (3.2) é reescrita adicionando o termo ligado a evolução no tempo.

$$H\psi = i\hbar \frac{d}{dt}\psi \tag{3.6}$$

Por fim, ao acoplar as equações (3.5) e (3.6), obtêm-se a definição completa da **equação de Schrödinger**.

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r},t) \right] \psi = i\hbar \frac{d}{dt}\psi \tag{3.7}$$

Essa equação é linear e homogênea em ψ, ou seja, pelo Princípio de Superposição, existem soluções de funções de onda para essa equação, que podem ser expressas como combinação linear de soluções já conhecidas. A seguir, será iniciada uma revisão sobre as aplicações dessa equação para potenciais clássicos da literatura, a fim de servir como base de comparação para os resultados que serão encontrados com a implementação desses potenciais em uma rede neural.

3.1 Poço de potencial quadrado e infinito

Um caso elementar sobre a aplicação da equação de Schrödinger é o problema do poço quadrado infinito, pois sua finalidade é encontrar os estados estacionários de uma partícula que está contida nesse poço potencial quântico. Dessa forma, sabendo que a função potencial é constante e que a largura do poço é a, tem-se que: $V(x) = 0$, quando $x \in (0, a)$ e $V(x) \to \infty$, quando $x \in]0, a[$. Além disso, define-se que a função de onda é contínua dentro desse intervalo e é nula na parte que está fora, garantindo a aplicação das condições de contorno sem efeito de borda. Com isso, ao reescrever a equação (3.7) para o caso unidimensional, encontra-se que:

$$\left[-\frac{\hbar^2}{2m}\frac{d}{dx^2} + V(x)\right]\psi = E_n\psi \Rightarrow \frac{d\psi}{dx^2} + k^2\psi = 0, \tag{3.8}$$

ao assumir a constante k como $\sqrt{2mE_n/\hbar^2}$ e resolver a equação diferencial subsequente, encontram-se a função de onda estacionária normalizada e os subníveis de energia:

$$\psi = \sqrt{\frac{2}{a}} sen\left(\frac{n\pi x}{a}\right), \tag{3.9}$$

$$E_n = \frac{n^2\pi^2\hbar^2}{2ma^2}. \tag{3.10}$$

Com isso, pode-se encontrar os estados estacionários da partícula em um poço potencial infinito e descrevê-los de forma analítica, utilizando (n) como um número inteiro e positivo.

3.2 Oscilador Harmônico unidimensional

O oscilador harmônico simples é um sistema fundamental para o estudo aproximativo de inúmeros problemas mais complexos, pois a sua formulação física permite que qualquer

potencial que possua um mínimo local, caracterizando um poço potencial, possa ser descrito de forma aproximativa por um oscilador harmônico. Além disso, o estudo de suas propriedades auxilia na descrição de movimentos periódicos moleculares, como a vibração do átomo de uma molécula e a caracterização de partículas idênticas, como os *Bósons*.(SAKURAI; NAPOLITANO, 2013)

A oscilação característica desse sistema é causada quando uma partícula está imersa em um potencial, no qual possui um valor mínimo, e é atraída para esse valor de forma oscilatório, chama-se essa referência de ponto de equilíbrio estável. Então, seja uma partícula de massa m, movendo-se em um potencial $V(x) = Kx^2/2$, no qual possui valor mínimo quando x for igual 0. Dessa forma, essa partícula vai ser atraída para esse valor, que corresponde a sua posição de equilíbrio mais estável, por intermédio da ação de uma força restauradora expressa por $-Kx$, em que a constante K é retirada da Lei de Hooke.

$$\omega = \sqrt{K/m} \Rightarrow K = m\omega^2 \tag{3.11}$$

Para o oscilador harmônico simples, os autovalores e autovetores da equação do movimento podem ser encontrados com a implementação de operadores que criam ou destroem pacotes de energia, pois agora é assumido a transição para maiores ou menores subníveis de energia. Com isso, essa transição da partícula para outro subnível energético somente ocorre com ganho ou perda de pacotes (*quantum*) de energia, relacionado com os operadores de criação ou de destruição.

Sendo assim, define-se dois operadores não-Hermitianos, ou seja, o adjunto opera de forma diferente que o não-adjunto, chamados de operador de criação ($a^\dagger$) e de destruição (a), onde atuam sobre os operadores de posição e de momento que se encontram no observável relacionado à energia do sistema. Além disso, ao aplicar a relação de comutação canônica nesses operadores, encontra-se que $[a, a^\dagger]$ é igual a 1.

$$a^\dagger = \sqrt{\frac{m\omega}{2\hbar}} \left(\mathbf{x} - i\frac{\mathbf{P}}{m\omega} \right) \tag{3.12}$$

$$a = \sqrt{\frac{m\omega}{2\hbar}} \left(\mathbf{x} + i\frac{\mathbf{P}}{m\omega} \right) \tag{3.13}$$

Os operadores $\mathbf{x}$ e $\mathbf{P}$, se relacionam com a energia total do sistema, por meio da equação que define a função Hamiltoniana clássica (3.3) aplicada no potencial do oscilador

harmônico simples, no qual foi mencionado anteriormente.

$$\mathscr{H} = \frac{P^2}{2m} + \frac{1}{2}m\omega^2 x^2 \tag{3.14}$$

Outro passo que deve ser feito, ao estudar a equação do movimento do oscilador harmônico simples, é a definição de um operador Hermitiano, no qual relaciona a e $a^\dagger$ entre si, chamado de operador número (**N**), que é definido como: $\mathbf{N} = a^\dagger a$. Esse operador é aplicado em uma equação de autovalor, em que suas componentes de auto-estado são relacionados à energia de **N**, então tem-se que: $\mathbf{N}|n\rangle = n\,|n\rangle$. Ao substituir o operador número no operador Hamiltoniana e igualar com o espectro de energia E_n, encontra-se os subníveis de energia referentes ao oscilador harmônico simples:

$$H\,|n\rangle = (N+1/2)\hbar\omega\,|n\rangle \Rightarrow E_n = (n+1/2)\hbar\omega. \tag{3.15}$$

Quando n é zero, a equação acima resulta no estado fundamental do sistema. Além disso, ao acoplar as definições dos operadores de criação e de destruição com a equação de autovalor para o operador número, obtêm-se duas novas relações: $a\,|n\rangle = \sqrt{n}\,|n-1\rangle$ e $a^\dagger\,|n\rangle = \sqrt{n+1}\,|n+1\rangle$, em que nota-se a função desses operadores na transição entre os níveis de energia n, pois a aplicação de a transfere a partícula de um nível n para $n-1$, ao contrário de $a^\dagger$ que transfere para $n+1$. Essa transição ocorre com o ganho ou a perda de unidades energéticas $\hbar\omega$, visto que os níveis de energia do oscilador harmônico simples são equidistantes.

A equação de Schrödinger, com a representação em termos de $|x\rangle$ para esse sistema quântico, fornece como solução um conjunto de *autokets* $|\phi\rangle$ de H. Esses vetores de estado podem se relacionar com os operadores de criação e de destruição, da seguinte forma: $a\,|\phi_n\rangle = \sqrt{n}\,|\phi_{n-1}\rangle$ e $a^\dagger\,|\phi_n\rangle = \sqrt{n+1}\,|\phi_{n+1}\rangle$. Dessa maneira, as soluções para a função de onda são acopladas entre si, e não podem ser expressas de forma independente para cada n.

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}m\omega^2 x^2\right]|\phi\rangle = E_n\,|\phi\rangle \tag{3.16}$$

Com isso, sabe-se que para n igual a zero, o vetor de estado correspondente é $|\phi_0\rangle$ e possui as seguintes relações: $a\,|\phi_0\rangle = 0$ e $|\phi_1\rangle = a^\dagger\,|\phi_0\rangle$. Da mesma forma, para n igual a 1, o vetor de estado correspondente é $|\phi_1\rangle$, então: $\pm\sqrt{2}\,|\phi_2\rangle = a^\dagger\,|\phi_1\rangle$. Logo, expandindo o vetor de estado $|\phi_2\rangle$, em função do vetor de estado inicial, tem-se que:

$$|\phi_2\rangle = \frac{1}{\sqrt{2}}a^\dagger\,|\phi_1\rangle - \frac{1}{\sqrt{2}}(a^\dagger)^2\,|\phi_0\rangle\,. \tag{3.17}$$

Portanto, seja um vetor de estado $|\phi_{n-1}\rangle$ conhecido e que possa ser normalizado, a generalização da relação entre os *autokets* de H pode ser reescrita, da seguinte forma:

$$|\phi_n\rangle = c_n a^\dagger |\phi_{n-1}\rangle , \tag{3.18}$$

$$|\phi_n\rangle = \frac{1}{\sqrt{n!}}(a^\dagger)^n |\phi_0\rangle , \tag{3.19}$$

onde c_n é uma constante de normalização.

Ao realizar o cálculo da equação diferencial envolvida por $a|\phi_0\rangle = 0$ e relacionar o operador de criação com os operadores de posição e de momento, a equação acima pode ser reescrita por meio de polinômios de *Hermite*, solucionando a equação de autovalor e encontrando o resultado para as suas funções de onda.

$$\phi_n(x) = \left[\frac{1}{2^n n!}\left(\frac{\hbar}{m\omega}\right)^n\right]^{1/2}\left(\frac{m\omega}{\pi\hbar}\right)^{1/4}\left[\frac{m\omega}{\hbar}x - \frac{d}{dx}\right]^n e^{-\frac{1}{2}\frac{m\omega}{\hbar}x^2} \tag{3.20}$$

Logo, ao fazer uma simples conversão dos valores de n, encontra-se a função de onda para o estado fundamental ($n = 0$) e para o primeiro estado de energia do oscilador harmônico ($n = 1$).

$$\phi_0(x) = \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{1}{2}\frac{m\omega}{\hbar}x^2} \tag{3.21}$$

$$\phi_1(x) = \left[\frac{4}{\pi}\left(\frac{m\omega}{\hbar}\right)^3\right]^{1/4} x e^{-\frac{1}{2}\frac{m\omega}{\hbar}x^2} \tag{3.22}$$

Portanto, percebe-se que a função de onda estacionária no estado fundamental, depende somente da forma que a função exponencial junto ao argumento constante se posiciona, formando um pico de máximo local. Diferentemente, da função de onda para o primeiro estado de energia, que está relacionada com o produto da função linear com a função exponencial, criando no gráfico uma oscilação, com cristas de máximo de mínimo locais.

4 IMPLEMENTAÇÃO COMPUTACIONAL

4.1 Equação de Schrödinger por Métodos Computacionais

Nesta seção, será discutida a implementação computacional da Equação de Schrödinger, por intermédio de métodos numéricos computacionais, cuja a finalidade é encontrar resultados que possam ser comparados com as respostas encontradas por meio das Redes Neurais. Dessa forma, inicia-se o programa com a escolha do problema do poço potencial quadrado infinito e depois generaliza para um potencial variável.

Assim, define-se uma função potencial constante, onde as paredes são formadas por valores numéricos muito grandes. Com isso, essa função recebe um conjunto de variáveis x e retorna os valores constantes 0 ou V_0, definindo a forma do poço em um gráfico de representação, com largura L. Dessa forma, utiliza-se a função "np.vectorize()" para transformar a função em um vetor e formar o gráfico desse potencial.

Após isso, utiliza-se o método numérico de Runge-Kutta (Quarta Ordem) ou RK4, no qual resolve numericamente equações diferenciais, com o intuito de encontrar as funções de onda para uma dada energia E. Com isso, define-se uma variável $\phi(x)$ como a derivada em relação x da função de onda $\psi(x)$, tal que a Equação de Schrödinger é reescrita da seguinte forma:

$$\frac{d\phi(x)}{dx} = \frac{2m}{\hbar^2}(V(x) - E)\psi(x). \tag{4.1}$$

Ademais, define-se um vetor $\mathbf{r}(x)$ como sendo a multiplicação de $\phi(x)$ e $\psi(x)$ aplicada à operação transposta, tal que: $\mathbf{r}(x) = [\phi(x)\psi(x)]^T$, e uma função que depende desse vetor e da variável x. Ao aplicar essas definições na equação (4.1), obtêm-se as seguintes relações:

$$f(\mathbf{r},x) = \frac{d\mathbf{r}}{dx}, \tag{4.2}$$

$$f_0(\mathbf{r},x) = r_1, \tag{4.3}$$

$$f_1(\mathbf{r},x) = \frac{2m}{\hbar^2}(V(x) - E)r_0, \tag{4.4}$$

onde f_0 e f_1 são os termos da função vetorial f, que serão utilizados para multiplicar o parâmetro de passos h. A conceituação matemática desse método é mostrada na seção A. Portanto, conclui-se que a utilização dessa técnica retorna a aproximação das funções de onda para o estado fundamental, determinada a seguir:

$$\psi_{n+1} = \psi_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{4.5}$$

O código que define o método de RK4 é uma função que trabalha em um intervalo (a,b) com N pontos, na qual recebe os valores de energias e retorna um vetor (ou *array*) dos valores relacionados à função de onda, onde pode ser encontrado na seção D.

Com a finalidade de encontrar o valor aproximado dos estados de energia fundamentais, utilizou-se o método das secantes, no qual pode ser encontrado na seção B. Esse método computacional é capaz de encontrar as soluções aproximadas de equações diferenciais, ao utilizar a iteração entre pontos fixos, ou seja, pode-se encontrar uma nova solução, ao conhecer uma outra anterior. Dessa forma, ao reestruturar esse método para a equação de autovalor desejada, encontra-se a seguinte relação:

$$E^{n+1} = E^n - \psi^n \frac{E^n - E^{n-1}}{\psi^n - \psi^{n-1}}, \tag{4.6}$$

onde o termo ψ^n é encontrado pela aplicação do método RK4.

A formulação computacional desse método é uma função que recebe valores distintos para o limites das energias, ou seja, o conjunto no qual espera-se que a energia esperada deva participar. Com isso, ocorre o cálculo análogo à equação (4.6), e o valor retornado é a energia do estado fundamental. Sendo assim, os detalhes do código são encontrados na seção E.

Além disso, com o intuito de encontrar a função de onda normalizada, implementa-se primeiramente o método de integração, no qual foi utilizada a Regra de Simpson, que pode ser encontrada na seção C. Com isso, a formulação computacional desse método é definido como uma função que recebe um *array*, no qual seria o vetor formado pela função de onda, e um parâmetro de cálculo h, além de retornar outro *array* do mesmo tamanho do vetor de entrada. Esse resultado é utilizado em outra implementação, que de forma análoga ao método de integração, retorna um vetor com mesmo tamanho, mas que representa a função de onda normalizada. Os detalhes do código são encontrados na seção F.

Com isso, ao implementar os algoritmos descritos acima na Equação de Schrödinger unidimensional, para o problema do poço potencial quadrado, encontraram-se os seguintes resultados.

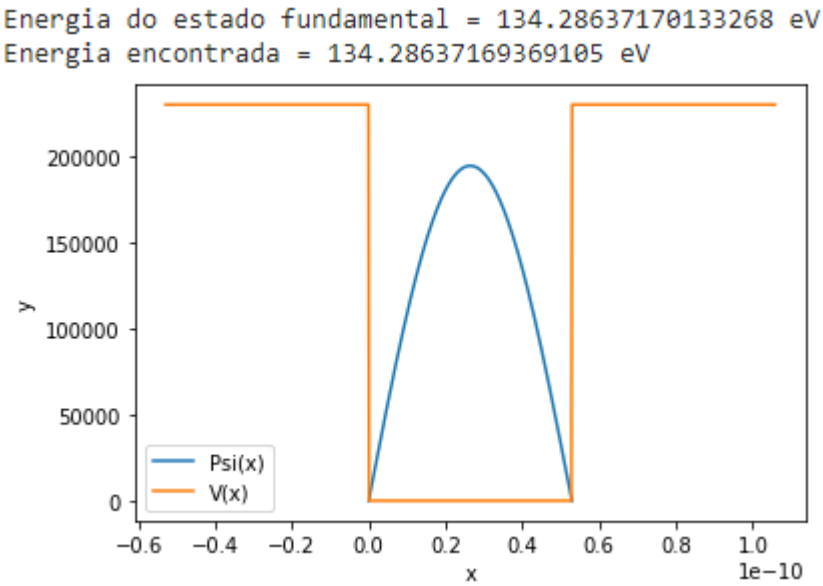

Figura 6 – Visualização da função de onda dentro do poço potencial quadrado infinito, para o primeiro nível de energia (n = 1).

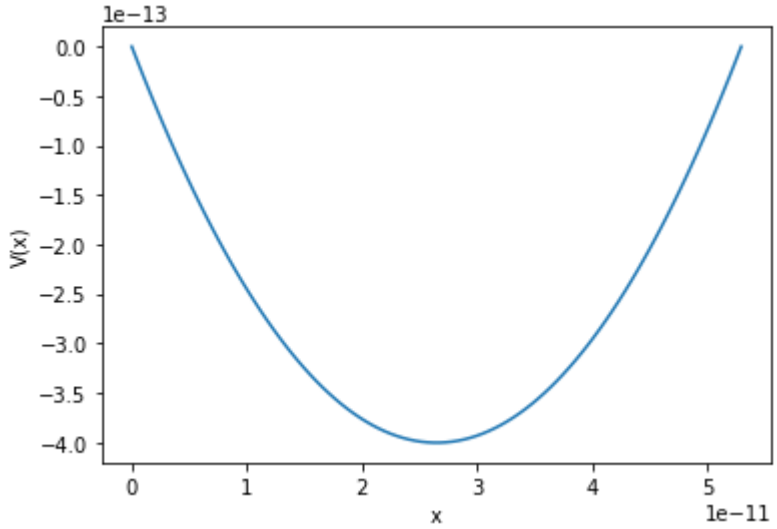

Figura 7 – Visualização da função potencial para o Oscilador Harmônico Simples (OHS).

Além disso, ao aplicar os métodos para o potencial do oscilador harmônico simples, encontrou-se a sua respectiva função de onda normalizada. Portanto, nota-se a eficiência desses métodos em descrever de forma numérica das funções de onda para os determinados potenciais.

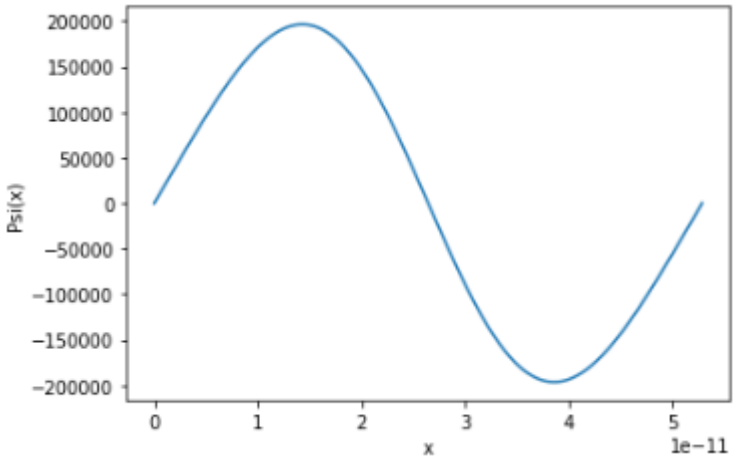

Figura 8 – Visualização da função de onda para o potencial que representa o Oscilador Harmônico Simples (OHS).

4.2 Implementação da Rede Neural

Nesta seção, será mostrada a implementação de uma rede neural para a resolução aproximada da Equação de Schrödinger unidimensional, na qual busca encontrar a representação das funções de onda nos estados fundamentais de energia. Para isso, houve a preparação do código com a utilização da biblioteca de software TensorFlow, visto que a sua linguagem favorece a criação de modelos que utilizam redes neurais mais complexas. Dessa forma, a simplificação e a modelagem no formalismo dessa biblioteca, por serem de fácil reconhecimento, são capazes de auxiliar na identificação das estruturas de uma rede neural.

```
# Modelo da Rede Neural
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='softplus', input_dim=len(inputs[0])),
    tf.keras.layers.Dense(512, activation='softplus'),
    tf.keras.layers.Dense(760, activation='softplus'),
    tf.keras.layers.Dense(512, activation='softplus'),
    tf.keras.layers.Dense(256, activation='softplus'),
    tf.keras.layers.Dense(len(outputs[0]), activation='linear')
])

# Otimizadores e Erro
model.compile(optimizer='adamax',     # otimizador ADAMAX
              loss='mse',             # erro por MSE
              metrics=['accuracy'])
```

Figura 9 – Código de implementação de uma Rede Neural Densa, por meio da biblioteca TensorFlow.

A construção do modelo de NN foi caraterizada pelo uso da Rede Neural Densa, a qual é chamada pela função “tf.keras.layers.Dense”, e utilizou uma arquitetura simétrica

em relação a camada intermediária. Com isso, no exemplo de implementação da Figura (9), a primeira e última camadas (*input layer* e *output layer*) são representadas por "input_dim", presente na segunda linha de código, e por "len(outputs)", na quinta linha do programa. Dessa maneira, esses objetos foram definidas, cada um, com 100 neurônios, diferentemente das camadas internas (*hidden layer*) que seguiram com 256,512,760,512 e 256, respectivamente para *hidden layer 1*, *hidden layer 2*, *hidden layer 3*, *hidden layer 4* e *hidden layer 5* . Dessa forma, como foi visto anteriormente, a escolha de uma Rede Neural Densa faz com que todos os neurônios de uma camada se conectem com todas as outras células das camadas adjacentes, fazendo que o número de parâmetros de peso aumentem.

Além disso, percebe-se a utilização da mesma função de ativação em todas as camadas internas, chamada de função *softplus*, na qual é uma variação da função ReLu e sua derivada retorna uma função logística, e a função "linear" na camada de *output*. O gerador utilizado na otimização, foi o operador *Adam*, o qual é um dos mais eficientes para a otimização do gradiente descendente, auxiliando na diminuição da função custo. Essa função, definida por *loss*, foi definida por meio do Erro Quadrático Médio MSE e, por fim, foram definidas a quantidade de *epochs* que a máquina vai utilizar.

Concluída a montagem da Rede Neural, o próximo passo é treinar o modelo para receber valores de potenciais como amostra e seus estados fundamentais como alvo, a fim de fazê-la aprender com esses dados e aplicar em um novo conjunto de potenciais. Desse modo, o objetivo desse novo estágio é gerar potenciais aleatórios e obter suas respectivas funções de onda. Dessa forma, foi utilizada a função "generate_potential()", na qual recebe dois tipos de parâmetros:

- *potential type*: essa variável recebe um texto específico (*string*), no qual seleciona o tipo de potencial que será formado. O termo *step* seleciona potenciais do tipo "degrau"; o termo *linear* seleciona potenciais do tipo "linear" e o termo *fourier* seleciona potenciais do tipo "senoidal";
- Característica do potencial: número que varia de 0 a 1, e modifica a característica do potencial, como o número de cristas ou o número de descontinuidades, aumentando ou diminuindo a sua complexidade.

Essa função retorna os potenciais gerados de forma aleatória, os quais são utilizados para encontrar as respectivas funções de onda, ao aplicar esses resultados no método numérico de resolução da Equação de Schrödinger, analisado na subseção anterior.

Portanto, ao encontrar esse conjunto de potenciais aleatórios e funções de onda respectivas, ocorre o treinamento da Rede Neural, a fim de otimizar os *weights* e *bias*. Com isso, após treinar esses parâmetros e analisar os *insights* do modelo, percebe-se que a função custo minimiza com o aumento do número de *epochs* e que todos os parâmetros de peso da Rede Neural foram otimizados.

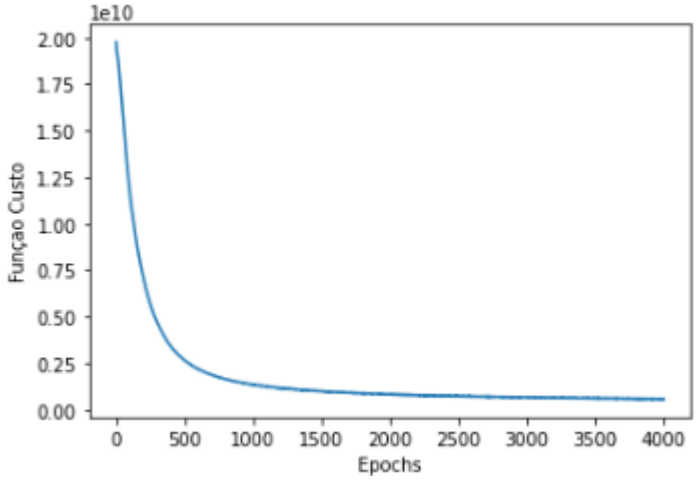

Figura 10 – Visualização da função custo em relação ao número de epochs, para os parâmetros treinados.

```
Dimensão de Input: 100
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_63 (Dense)            (None, 256)               25856

 dense_64 (Dense)            (None, 512)               131584

 dense_65 (Dense)            (None, 760)               389880

 dense_66 (Dense)            (None, 512)               389632

 dense_67 (Dense)            (None, 256)               131328

 dense_68 (Dense)            (None, 100)               25700

=================================================================
Total params: 1,093,980
Trainable params: 1,093,980
Non-trainable params: 0
```

Figura 11 – Visualização do total de parâmetros que foram treinados pela Rede Neural Densa, com 7 camadas totais e 4000 epochs.

Assim, de forma resumida, inicialmente houve a criação estrutural da Rede Neural, com a escolha do tipo de utilização, das funções de ativação e otimização, e dos hiper-parâmetros (número de camadas, número de células de ativação, número de *epochs*). Em seguida, houve a criação de um código para gerar potenciais aleatórios que seriam utilizados no treinamento da Rede Neural, os quais foram aplicados na resolução numérica da Equação de Schrödinger, com o intuito de encontrar as funções de onda no estado fundamental. Dessa maneira, ao obter o conjunto de amostras (potenciais) e o conjunto de alvos (funções de onda), treinou-se a Rede Neural, a fim de minimizar a função custo e otimizar os parâmetros internos. Nesse aspecto, ao término do treinamento, houve a fase de teste para diferentes potenciais.

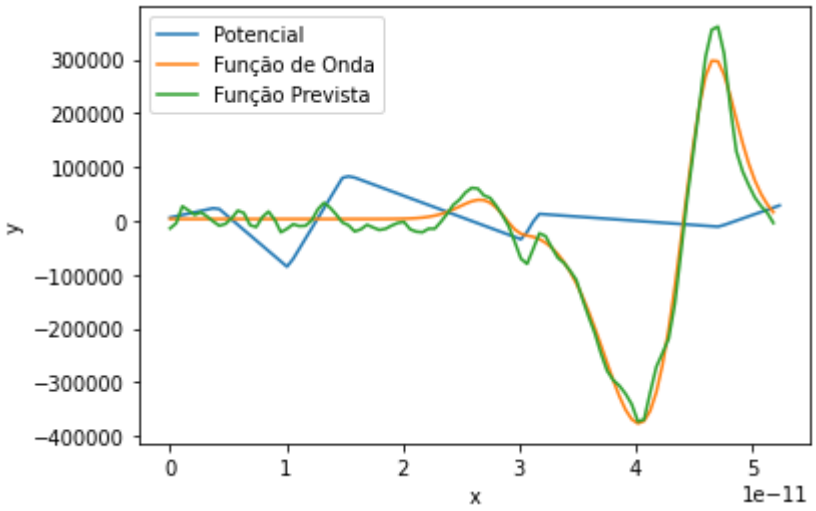

Figura 12 – Visualização de uma função potencial randômica, junto às funções de onda encontradas de forma numérica e pelo uso da Rede Neural.

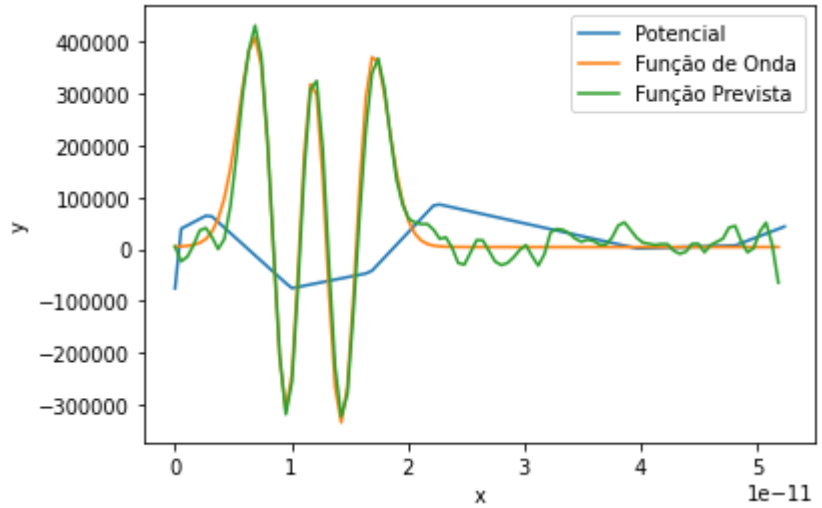

Figura 13 – Visualização da segunda função potencial randômica, junto às funções de onda encontradas de forma numérica e pelo uso da Rede Neural.

As Figuras (12) e (13) foram introduzidos por funções potenciais mescladas entre os tipos “degrau” e “linear”, contudo, percebe-se que a função de onda prevista (calculada por NN) possui uma acurácia boa em relação ao resultado calculado de forma numérica.

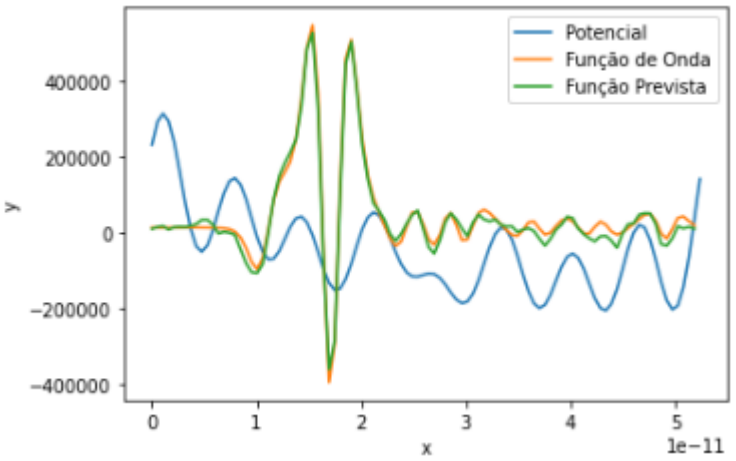

Figura 14 – Visualização da terceira função potencial randômica, junto às funções de onda encontradas de forma numérica e pelo uso da Rede Neural.

Além disso, na Figura (14), em que o potencial se expressa como uma função senoidal, a Rede Neural conseguiu convergir os valores previstos para os valores esperados, garantindo também a minimização da função custo para esses potenciais.

Com isso, após os resultados encontrados para os potenciais aleatórios acima, foi implementada a mesma Rede Neural na resolução das funções de onda para os potenciais estudados na seção 4.1. Dessa maneira, a utilização dessa técnica teve o intuito de comparar os resultados por métodos numéricos e por Redes Neurais, para potenciais previamente conhecidos. Portanto, os seguintes resultados foram encontrados para esse novo conjunto de potenciais.

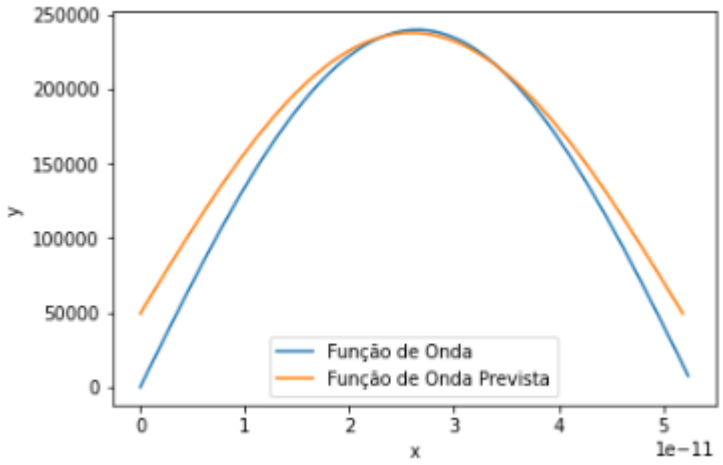

Figura 15 – Visualização da função de onda para o poço potencial quadrado, na qual foi encontrada por métodos computacionais e pela implementação de Redes Neurais.

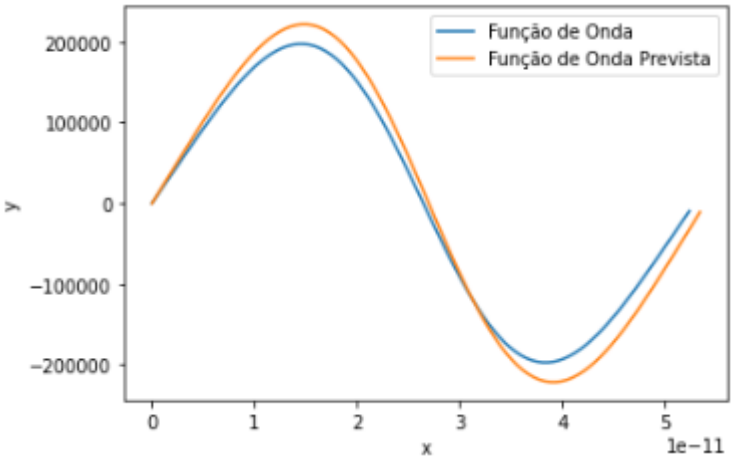

Figura 16 – Visualização da função de onda para o potencial do oscilador harmônico simples, na qual foi encontrada por métodos computacionais e pela implementação de Redes Neurais.

4.2.1 Testes da Rede Neural para diferentes funções de aplicação

Nesta seção, o trabalho será voltado para a análise dos resultados encontrados por meio da Rede Neural, ao serem aplicadas diversas funções de ativação. Antes disso, é válido explicar as diferenças entre as principais funções de ativação utilizadas em programas de Redes Neurais. Dessa forma, a função mais básica é a **linear**, pois trata-se apenas de um fator multiplicativo aplicado no valor que a célula recebe, a qual é bastante útil em problemas de regressão ou em aplicações lineares.

Outra função importante é a **sigmoide**, usada em problemas de regressão logística e de classificação binária, a qual seus valores resultam entre 0 e 1. Além disso, cita-se a função **softmax**, a qual é uma generalização da função linear e sigmoide para problemas de classificação multi-classes, e geralmente é utilizada nas camadas de saída da rede neural.

Ademais, outras duas funções bastante utilizadas no âmbito da programação por *Machine Learning* são as funções **tanh** (tangente hiperbólica) e **ReLU** (*rectified linear unit*). Dessa maneira, a função **tanh** é bastante útil em problemas não-lineares, e seus resultados retornam no intervalo de -1 a 1, caracterizando-se como uma função centrada em zero, que calcula em células que receberam valores negativos. Por fim, cita-se a função **ReLU**, a qual retorna valores entre $[0, +\infty[$ e não costuma ser utilizada em camadas de saída por conta dessa divergência. Porém, essa função possui um problema, pois o seu formalismo matemático não recebe valores negativos, implicando na exclusão da atuação de células que possuam esse valor, ao encontrar o resultado final.

A função de ativação **softplus**, que foi utilizada na implementação da Rede Neural, é uma generalização da função ReLU, e a sua derivada gera uma função logística similar à função sigmoide. Com isso, é criada uma Rede Neural com três camadas internas e as mesmas funções de ativação da Figura (9). No entanto, para esse teste, troca-se a função **softplus** pela função **sigmoide** na primeira camada da *hidden layer*.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='sigmoid', input_dim=len(inputs[0])),
    tf.keras.layers.Dense(512, activation='softplus'),
    tf.keras.layers.Dense(256, activation='softplus'),
    tf.keras.layers.Dense(len(outputs[0]), activation='linear')
])
```

Figura 17 – Modelo da Rede Neural Densa, com a função de ativação sigmoide na primeira camada interna.

Dessa forma, o resultado gráfico obtido foi comparado com a função de onda numérica, junto à função de onda encontrada pelo modelo original.

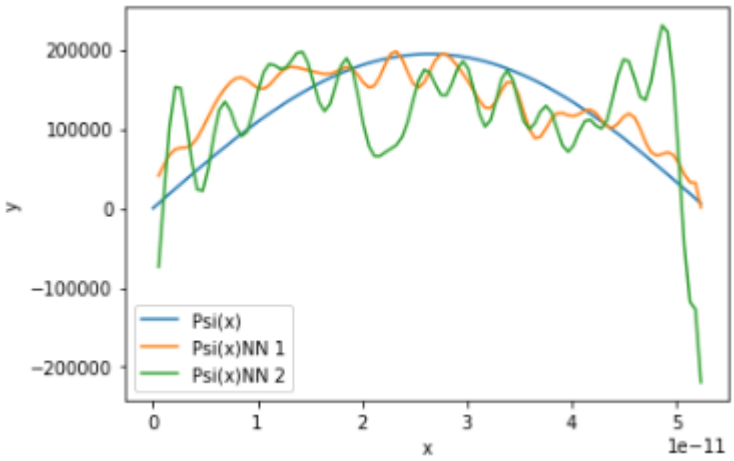

Figura 18 – Visualização da função de onda prevista, por meio da função sigmoide (cor verde) e da função softplus (cor laranja), comparada com a função de onda numérica (cor azul).

Após isso, ao refazer o mesmo processo de troca nas funções de ativação, utilizou-se a função **softmax** também na primeira camada interna, e manteve-se as mesmas funções de ativação das camadas posteriores. Portanto, o resultado encontrado na Figura (19) foi comparado com o modelo original e a função de onda numérica.

O teste com a utilização da função **ReLu** não foi possível, pois a sua peculiaridade de aceitar somente valores positivos fez com que a função de onda não fosse gerada, visto que haviam células que apresentavam valores negativos. Com isso, a última função de ativação

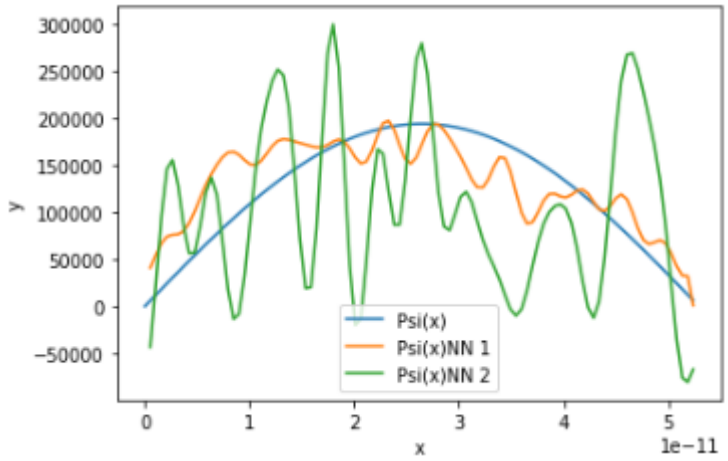

Figura 19 – Visualização da função de onda prevista, por meio da função softmax (cor verde) e da função softplus (cor laranja), comparada com a função de onda numérica (cor azul).

testada foi a função **tanh**, mostrada na Figura (20), a qual apresentou uma resposta similar à Figura (18), pois seu conjunto de imagem complementa o da função **sigmoide**.

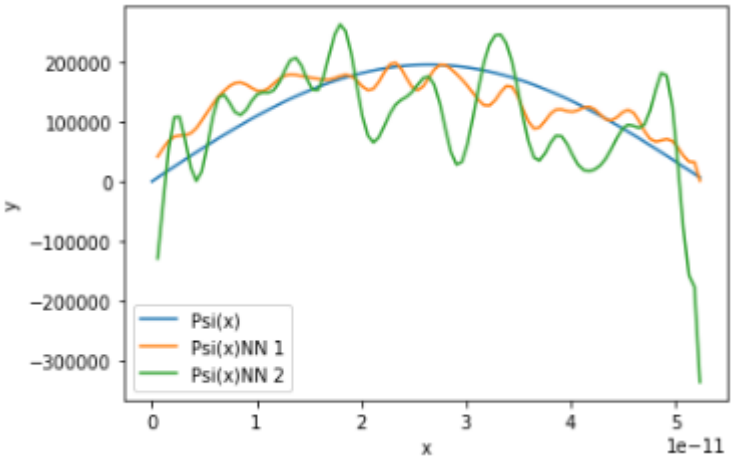

Figura 20 – Visualização da função de onda prevista, por meio da função tanh (cor verde) e da função softplus (cor laranja), comparada com a função de onda numérica (cor azul).

Portanto, tendo em vista os resultados encontrados, percebe-se que a função de ativação **softplus** adequou melhor os valores de *output* com os valores esperados pela função de onda numérica. Dessa forma, a sua escolha é útil para transformar a Rede Neural em um modelo gerador generalizável, ou seja, seja capaz de gerar funções de onda aproximadas para qualquer tipo de potencial.

4.2.2 Testes da Rede Neural para mudanças de hiper-parâmetros

Nesta seção, ocorrerá os testes da Rede Neural para variações os hiper-parâmetros, com o intuito de discutir acerca dos resultados encontrados. Com isso, houveram mudanças nas seguintes características: quantidades de camadas internas, números de células de ativação, quantidade de *epochs* e tamanho de *batch size*. Essa última propriedades, indica a quantidade de *inputs* que serão fornecidos para cada *epoch*, visto que se seu valor for definido como 1, então todo o conjunto de dados será fornecido por vez.

Ademais, é válido observar a não linearidade entre esses hiper-parâmetros, ou seja, não há relação linear entre o aumento desses aspectos e a eficiência do modelo. Dessa maneira, o objetivo principal é encontrar um modelo computacional que minimize a função custo, e tenha consumos operacional e temporal viáveis para a obtenção dos resultados.

4.2.2.1 Teste 1: Rede Neural para 3 camadas internas

Dessa forma, a primeira fase de teste ocorreu para uma Rede Neural Densa, com as mesmas funções de ativação, função custo e otimizadores da Figura (9). Com isso, implementou-se três camadas internas, totalizando cinco camadas com entrada e saída. Ademais, essa rede foi instruída a realizar quinhentas voltas (*epochs*) com *batch size* igual a dez, ou seja, cada volta analisava um décimo do tamanho total de dados de entrada.

Nesse aspecto, cita-se que a quantidade de dados de entrada foram 5990 pontos de potenciais, gerados de forma aleatória. Logo, para essa rede, cada *epoch* analisava 599 potenciais por vez. Portanto, nota-se que o aumento no valor do *batch size* diminui o tempo de realização do problema, mas aumenta o erro entre a previsão e o esperado.

A modelagem da Rede Neural para os aspectos discutidos é mostrada na Figura (21). Posto isso, houve a análise da função custo pela quantidade de *epochs*, e a solução das funções de onda para o potencial do poço quadrado e para outro potencial aleatório.

```
# Modelo da Rede Neural
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(190, activation='softplus', input_dim=len(inputs[0])),
    tf.keras.layers.Dense(350, activation='softplus'),
    tf.keras.layers.Dense(190, activation='softplus'),
    tf.keras.layers.Dense(len(outputs[0]), activation='linear')
])

# Otimizadores e Erro
model.compile(optimizer='adamax',     # otimizador ADAMAX
              loss='mse',             # erro por MSE
              metrics=['accuracy'])

# model.summary()

# Número de epochs
epochs = 500
```

Figura 21 – Modelo da Rede Neural Densa, com 3 camadas internas, 500 *epochs* e *batch size* igual a 10.

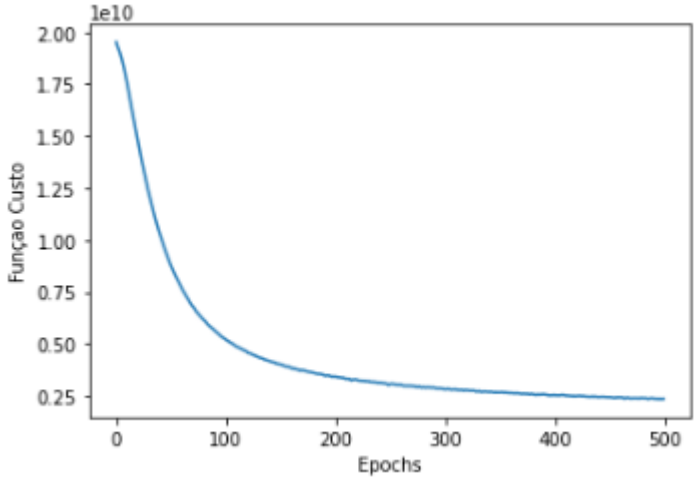

Figura 22 – Visualização da função custo, em relação à quantidade de *epochs* para a Rede Neural da Figura (21).

Com o intuito de verificar o fator de generalização do modelo criado, é analisada a resposta da rede para os potenciais implementados. Com isso, percebe-se que a escolha desses parâmetros possui menor consumo operacional e temporal, mas é ineficaz para descrever as soluções esperadas.

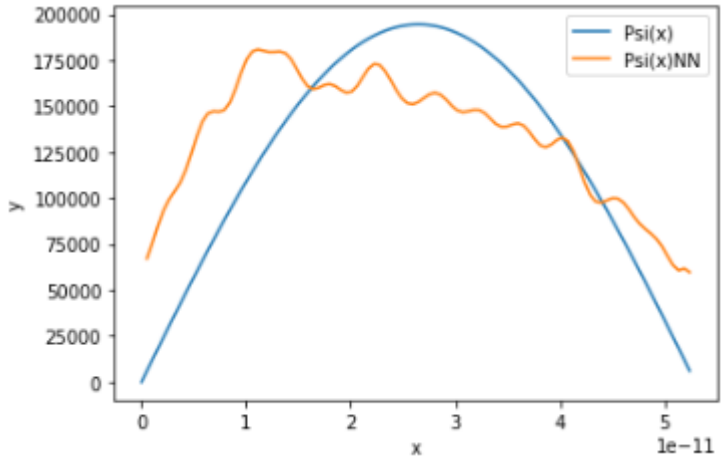

Figura 23 – Visualização da função de onda encontrada pela Rede Neural (Psi(x)NN), em comparação com a função numérica (Psi(x)), para o poço potencial quadrado.

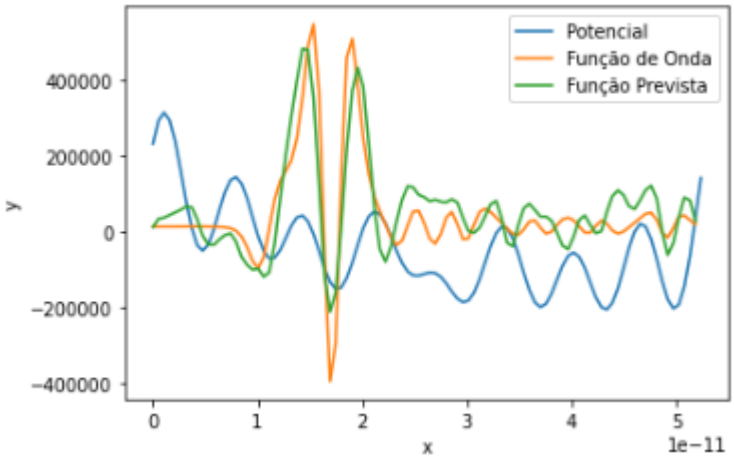

Figura 24 – Visualização da função de onda encontrada pela Rede Neural, em comparação com a função numérica, para o potencial aleatório.

Após isso, ocorreu outro teste, em que foi mantido o conjunto de hiper-parâmetros utilizados no modelo anterior, exceto a quantidade de *epochs*. Com isso, nessa nova verificação, foram utilizadas quatro mil *epochs*, e a obtenção dos respectivos resultados estão a seguir.

Dessa maneira, nota-se que a função custo dessa nova Rede Neural obteve sua minimização de forma melhorada, em comparação com o modelo anterior, pois utilizou um intervalo de tempo oito vezes maior. Além disso, a representação das funções de onda para os mesmos potenciais seguem abaixo.

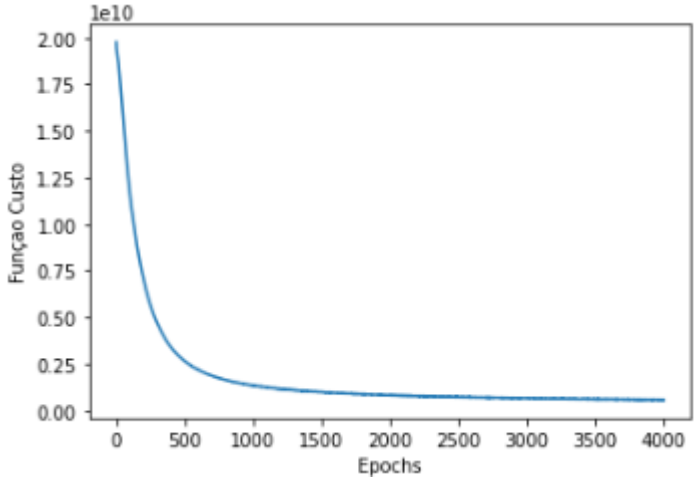

Figura 25 – Visualização da função custo, em relação à quantidade de *epochs* para a Rede Neural com 4000 *epochs* e 3 camadas.

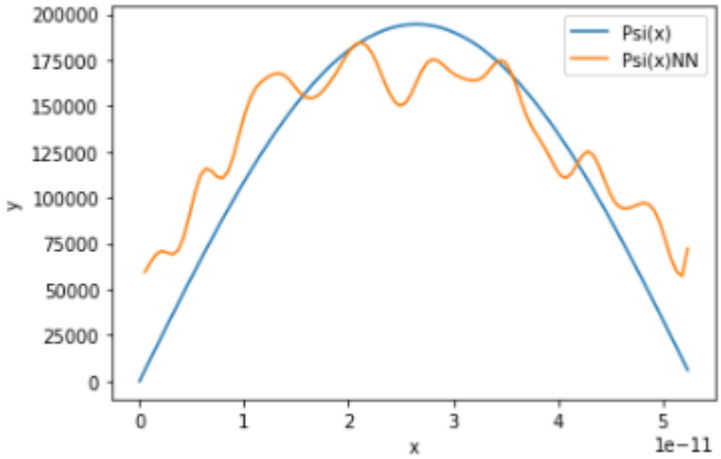

Figura 26 – Visualização da função de onda encontrada pela Rede Neural (Psi(x)NN) com 4000 *epochs*, em comparação com a função numérica (Psi(x)), para o poço potencial quadrado.

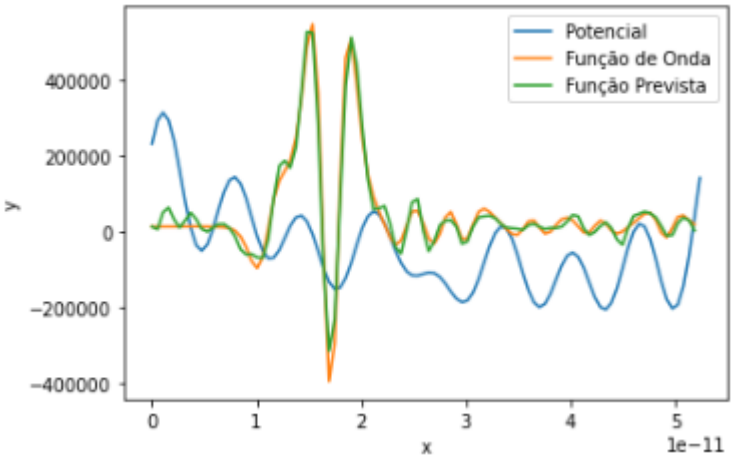

Figura 27 – Visualização da função de onda encontrada pela Rede Neural com 4000 *epochs*, em comparação com a função numérica, para o potencial aleatório.

Portanto, com a obtenção desses resultados, constata-se que o aumento no número de *epochs* foi eficaz para solução da função de onda para o potencial aleatório. No entanto, a melhoria do resultado para o poço potencial quadrado não foi expressiva, e o tempo de resolução total da rede foi extenso. Dessa forma, conclui-se que o aumento no número de *epochs* não foi o suficiente para a obtenção de resultados esperados em um intervalo de tempo viável. Com isso, são necessárias mudanças em outros hiper-parâmetros, a fim de ajustar o resultado encontrado.

4.2.2.2 Teste 2: Rede Neural para 5 camadas internas

A segunda fase de teste consiste com as mesmas funções de ativação, função custo e otimizadores da Figura (9). Porém, nesse caso, implementou-se cinco camadas internas, totalizando sete camadas com entrada e saída. Ademais, essa rede foi instruída a realizar quinhentas voltas (*epochs*) com *batch size* igual a vinte, ou seja, cada volta analisava o tamanho total de dados de entrada dividido por vinte.

O aumento dessa última quantidade foi necessária para diminuir o tempo operacional, pois o custo temporal para a resolução dessa rede neural é alto. Dessa forma, houve a análise da função custo e das soluções obtidas, tal como nos testes anteriores.

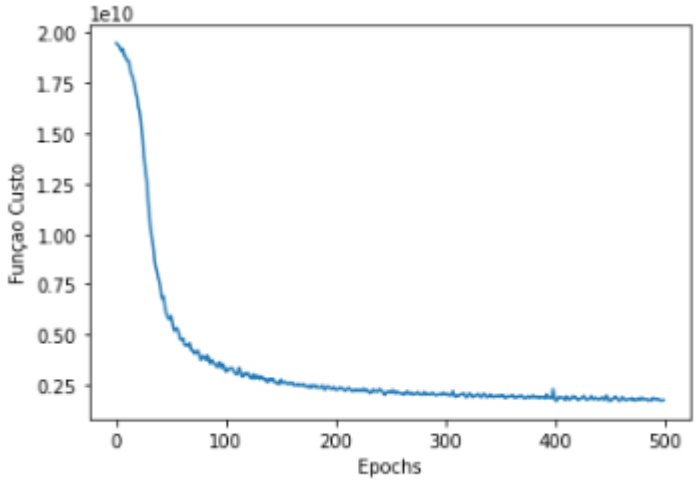

Figura 28 – Visualização da função custo, em relação à quantidade de *epochs* para a Rede Neural com 5 camadas e 500 *epochs*.

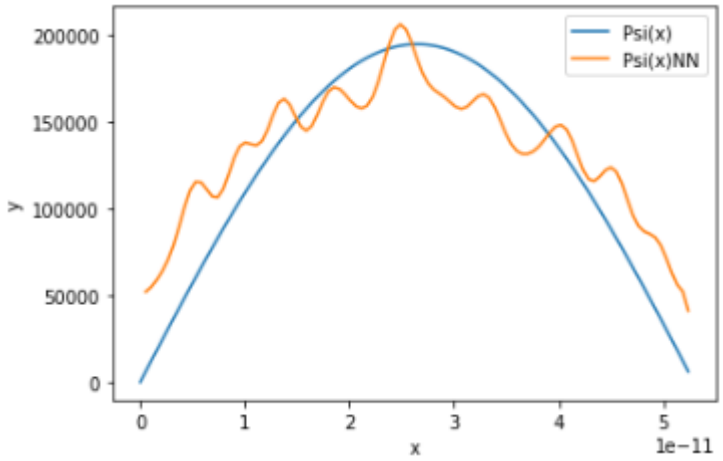

Figura 29 – Visualização da função de onda encontrada pela Rede Neural (Psi(x)NN) com 5 camadas e 500 *epochs*, em comparação com a função numérica (Psi(x)), para o poço potencial quadrado.

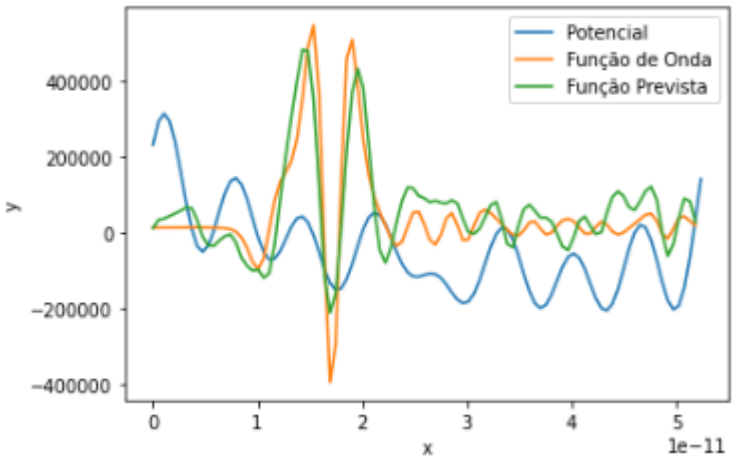

Figura 30 – Visualização da função de onda encontrada pela Rede Neural com 5 camadas e 500 *epochs*, em comparação com a função numérica, para o potencial aleatório.

Com isso, conclui-se que a utilização de uma Rede Neural com cinco camadas não é eficaz quando a quantidade de *epochs* for pequena, visto que a quantidade de parâmetros aumenta. Dessa forma, o melhor modelo encontrado para solucionar a equação de Schrödinger está representado na Figura (9), no qual utiliza de uma Rede Neural Densa com cinco camadas internas e com quatro mil *epochs*. Portanto, esse modelo possui ótima generalização, junto à minimização função custo e custos operacional e temporal viáveis para a resolução em vários potenciais.

5 CONCLUSÕES

Ao final do trabalho, conclui-se que a utilização de técnicas de *Machine Learning* é favorável para a descoberta de informações acerca de soluções que se tornariam difíceis de serem encontradas de forma analítica. A partir dessa análise, o uso crescente das redes neurais foi induzido pela facilidade de manuseio de suas ferramentas para obter as soluções, anteriormente dificultadas pelos métodos tradicionais.

Ademais, na primeira parte do experimento, foram utilizados métodos numéricos computacionais para calcular as funções de onda específicas de cada potencial, e observou-se a eficiência desses métodos ao descrever a energia do estado fundamental, na Figura 6, além da forma da função de onda para o potencial do oscilador harmônico simples, na Figura 7. Logo após, tais métodos foram utilizados para encontrar as funções de onda numéricas para outros potenciais, com o intuito de comparar com o aprendizado gerado pela rede neural.

Na segunda parte do experimento, houve a implementação de uma rede neural do tipo *dense*, caracterizando um algoritmo de *Deep Learning*, por conta da alta dimensionalidade dentro das camadas. Nesse parte do código, foram gerados potenciais aleatórios para um conjunto de treino, os quais foram aplicados nos métodos numéricos, a fim de obter as respectivas funções de onda. Dessa forma, o conjunto formado por esses potenciais e funções de onda foi utilizado no treinamento da rede neural.

Após a fase de treino, obteve-se *insights* sobre os parâmetros da rede neural, e por meio do Figura 10, conclui-se que os termos de peso e de viés, junto à escolha do número de *epochs*, foram eficazes na minimização da função de custo. Por fim, ao testar a rede neural para diferentes tipos de potenciais, observou-se que os métodos numéricos foram superiores para a formação da função de onda em comparação à rede neural. No entanto, o objetivo principal da implementação de uma rede neural não é a exatidão, e sim a possibilidade de generalização do código. Tendo isso em vista, percebe-se que o modelo de NN consegue ser generalizado para a formação da função de onda aproximada.

REFERÊNCIAS

CAMPONOGARA, E. **Integração Numérica.** 2022. Diponível em: http://docplayer.com.br/15392604-Regra-de-simpson-eduardo-camponogara.html. Acesso em: 8 jul. 2022.

COHEN-TANNOUDJI, C.; DIU, B.; LALOë, F. **Quantum Mechanics**. [S.l.]: Wiley-Vch, 2020. v. 1.

ESTEVA, A. R. A.; RAMSUNDAR, B. A guide to deep learning in healthcare. **Nature Medicine**, v. 25, p. 24–29, 2019.

GERON, A. **Mãos à Obra: Aprendizado de Máquina com Scikit-Learn e TensorFlow**. Rio de Janeiro: Alta Books Editora, 2019.

IMPORTANCE of Deep Learning. **Universidad de Alcalá**. 2022. Diponível em: https://master-artificialintelligence.com/importance-of-deep-learning/. Acesso em: 19 jul. 2022.

INTRODUÇÃO ao TensorFlow. 2017. Diponível em: https://www.tensorflow.org/. Acesso em: 10 jul. 2022.

JIN, L. Y. Y.; CHIANG, C.-E. Identifying exoplanets with machine learning methods: A preliminary study. **International Journal on Cybernetics & Informatics (IJCI)**, v. 11, p. 31–42, 2022.

KAPUR, R. **Recurrent Neural Networks and LSTMs.** 2017. Diponível em: https://ayearofai.com/rohan-lenny-3-recurrent-neural-networks-10300100899b. Acesso em: 10 jul. 2022.

O que é TensorFlow? Para que serve na prática?. **Didática Tech**. 2022. Diponível em: https://didatica.tech/o-que-e-tensorflow-para-que-serve/. Acesso em: 12 jul. 2022.

SAKURAI, J. J.; NAPOLITANO, J. **Mecânica Quântica Moderna**. Porto Alegre: Bookman Companhia Editora Ltda., 2013.

SCHLEDER, G. R.; FAZZIO, A. Machine learning na física, química, e ciências de materiais: Descoberta e design de materiais. **Revista Brasileira de Ensino de Física**, v. 43, 2021.

SOUZA, N. V. de Lima e. **Aplicação de técnicas elementares de machine learning à física.** 2020. Monografia (Bacharel em Física) — Universidade Federal do Rio Grande do Norte.

APÊNDICE A – MÉTODO DE RUNGE-KUTTA PARA ORDEM 4 (RK4)

É um método computacional que fornece a solução aproximada no cálculo de equações diferenciais em um determinado intervalo. Dessa forma, seja a solução de uma equação diferencial $y(x)$, então define-se uma função f que dependa dos valores de x e de y, e um parâmetro h que indica o tamanho dos passos que ocorrerá o cálculo numérico. Além disso, definem-se as funções de dependência k_i, em que o índice i representa a ordem do método utilizado. Nesse caso, o índice será (1,2,3 e 4).

$$k_1 = hf(x,y) \tag{A.1}$$

$$k_2 = hf(x+h/2, y+k_1/2) \tag{A.2}$$

$$k_3 = hf(x+h/2, y+k_2/2) \tag{A.3}$$

$$k_4 = hf(x+h, y+k_3) \tag{A.4}$$

As expressões acima são dependentes entre si, pois utilizam do resultado da equação anterior para prosseguir com a aproximação. O parâmetro h pode ser expresso como a distância entre os pontos inicial e final sobre a quantidade de "passos" dados, e é uma variável ajustável no código.

Por fim, ao juntar as expressões de k_i por meio de uma combinação linear e encontrar as constantes presentes nesse cálculo, a solução numérica pode ser calculada da seguinte forma:

$$y(x+h) = y(x) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{A.5}$$

Diferentemente das aproximações por série de Taylor, em que deve ser feita a diferenciação da função desejada, a aproximação por $RK4$ não possui essa necessidade, mas utiliza de um maior trabalho computacional.

APÊNDICE B – MÉTODO DAS SECANTES

È um método computacional que também fornece a solução aproximada no cálculo de equações diferenciais não lineares, cujo a ideia principal é aproximar a derivada de uma função pela razão fundamental utilizada no cálculo de limites. Dessa maneira, seja a função diferenciável $f(x)$ e sua derivada aproximada $f'(x)$:

$$f'(x) \approx \frac{f(x) - f(x_0)}{x - x_0}, \tag{B.1}$$

para x no limite de x_0.

Além disso, define-se no método aproximativo de Newton, que a iteração entre pontos fixos é dado da seguinte forma:

$$x^{(n+1)} = x^{(n)} - \alpha(x^{(n)}) f(x^{(n)}), \tag{B.2}$$

em que $\alpha(x^{(n)}) = 1/f'(x^{(n)})$ e $n \geq 1$. Esses pontos fixos são o conjunto de valores pertencentes ao domínio da função a ser aproximada, e com isso pode-se aproximar o resultado de α, por meio da equação (B.1), assim tem-se que:

$$\alpha(x^{(n)}) \approx \frac{x^{(n)} - x^{(n-1)}}{f(x^{(n)}) - f(x^{(n-1)})}, \tag{B.3}$$

onde ao reescrever o método aproximativo de Newton, obtêm-se a equação que dita a iteração por método de secantes, para valores de $n \geq 2$, tal que:

$$x^{(n+1)} = x^{(n)} - \frac{x^{(n)} - x^{(n-1)}}{f(x^{(n)}) - f(x^{(n-1)})} f(x^{(n)}). \tag{B.4}$$

APÊNDICE C – REGRA DE SIMPSON

É um método que fornece a solução aproximada no cálculo de integrais definidas em um determinado intervalo. Dessa forma, seja uma função $F(x)$ definida em um intervalo $[a,b]$, tal que:

$$F(x) = \int_a^b f(x)dx, \tag{C.1}$$

em que y_m é o ponto médio entre as coordenadas (a, b) e $f(x)$ é a função que será aproximada por um polinômio de grau 2, definida como $f^*(x)$. Com isso, utilizam-se o *polinômio de Gregory-Newton* e a mudança de variável de x para a dependência de um peso α, assim tem-se a formulação da função aproximativa $f^*(x)$:

$$f^*(x) = f(a) + (x-a)\frac{\Delta f(a)}{h} + (x-a)(x-y_m)\frac{\Delta^2 f(a)}{2h^2}, \tag{C.2}$$

onde $\Delta f(a)$ = $f(y_m)$ - $f(a)$ e $\Delta^2 f(a)$ = $f(b)$ - $2f(y_m)$ + $f(a)$. Faz-se então, a mudança de variável $x(\alpha) = a + \alpha h$, em que α pode tomar os valores inteiros (0, 1 e 2), e encontra-se as seguintes relações: $x(0) = a$; $x(1) = y_m$; $x(2) = b$ e $h = (b-a)/2$. Além disso, percebe-se também que ao encontrar a seguinte relação: $(x-a)(x-y_m) = \alpha(\alpha-1)h^2$, é possível substituir na equação (C.2) e encontrar a integral aproximada para a função $f^*(x)$:(CAMPONOGARA, 2022)

$$F(x) \approx \int_a^b f^*(x)dx = \frac{h}{3}[f(a) + 4f(y_m) + f(b)]. \tag{C.3}$$

Essa equação é chamada de **Regra de Simpson Simples**.(CAMPONOGARA, 2022)

A construção da versão composta dessa equação segue o mesmo raciocínio, mas agora implementa-se uma função chamada de integrador de Simpson (S), na qual é definida como:

$$S(h_j) = \sum_{j=1}^{n} S_j(h) = \sum_{j=1}^{n} \frac{h}{3}[f(x_j) + 4f(y_j) + f(x_{j+1})], \tag{C.4}$$

onde y_j e h_j podem ser reescritos de forma simples, ao trocar a e b, respectivamente por x_j e x_{j+1}. Com isso, o integrador de Simpson utilizado no cálculo numérico, de forma expandida, torna-se:

$$S(h) = \frac{h}{3}[f(x_1) + 4f(y_1) + 2f(x_2) + 4f(y_2) + ... + 2f(x_n) + 4f(y_n) + f(x_{n+1})], \tag{C.5}$$

em que o índice n é um parâmetro que pode ser modificado na implementação do código.

APÊNDICE D – CODIFICAÇÃO DO MÉTODO RK4

O anexo a seguir representa a formulação do código para o método RK4, com o intuito de encontrar os valores numéricos da função de onda para determinada energia.

```
#Método RK4
#Para encontrar a função de onda, dada uma determinada energia E
def solve(E, a = 0, b = L, N = 1000): # a, b: início e fim do conjunto de pontos
                                      # N: números de pontos dentro do conjunto (a,b)
    h = (b - a) / N                   # h: parâmetro que dita o tamanho dos passos


    psi = 0.0                         # condição inicial para psi
    phi = 1.0                         # condição inicial para psi
    r = np.array([psi, phi], float)   # definição do vetor r

    x_points = np.arange(a, b, h)     # pontos de x expressos no gráfico
    psi_points = []                   # array dos pontos de psi

# Inicialização do Método RK4
    for x in x_points:
        psi_points.append(r[0])
        k1 = h * f(r, x, E)
        k2 = h * f(r + 0.5 * k1, x + 0.5 * h, E)
        k3 = h * f(r + 0.5 * k2, x + 0.5 * h, E)
        k4 = h * f(r + k3, x + h, E)
        r += (k1 + 2 * k2 + 2 * k3 + k4) / 6

    x_points = np.append(x_points, b) # redimensiona os valores de x
    psi_points.append(r[0])           # adiciona os resultados em psi

    return x_points, np.array(psi_points), r[0]
```

Figura 31 – Visualização do código utilizado na formulação do método RK4.

APÊNDICE E – CODIFICAÇÃO DO MÉTODO DAS SECANTES

O anexo a seguir representa a formulação do código para o método das secantes, com o intuito de encontrar o autovalor aproximado da energia no estado fundamental.

```
# Método das Secantes
# Finalidade: Encontrar os autovalores da energia no estado fundamental
def find_energy_eigenvalue(E1, E2):   # E1, E2: conjunto inicial (E1, E2)
    psi2 = solve(E1)[2]               # Função de onda para energia E1
    epsilon = e / 1000                # Parâmetro para o "loop"

# Cálculo para encontrar a autovalor da energia
    while abs(E1 - E2) > epsilon:
        psi1, psi2 = psi2, solve(E2)[2] # Transformação da Função de onda para energia E2
        E1, E2 = E2, E2 - psi2 * (E2 - E1) / (psi2 - psi1) # Transformação de E2

    return E2
```

Figura 32 – Visualização do código utilizado na formulação do método das secantes.

APÊNDICE F – CODIFICAÇÃO DA NORMALIZAÇÃO DAS AUTOFUNÇÕES

O anexo a seguir representa a formulação do código para a Regra de Simpson e para a normalização da função de onda, com o intuito de encontrar a função de onda normalizada.

```
# Método de Integração
# Regra de Simpson
def integrate(arr, h): # arr: "array", h: parâmetro de cálculo
    N = len(arr)       # N: número igual ao tamanho do "array" de entrada
# Cálculo de integração
    return 1/3 * h * (arr[0] + arr[N - 1] +
                      4 * sum([arr[2*k - 1] for k in range(1, int(N/2) + 1)]) +
                      2 * sum([arr[2*k] for k in range(1, int(N/2))]))

# Normalização da função de onda
def normalize(psi_points):
    return 1/math.sqrt(integrate([psi ** 2 for psi in psi_points],
                                 L / len(psi_points))) * np.array(psi_points)
```

Figura 33 – Visualização do código utilizado na formulação do método de integração.