Universidade Estadual de Maringá
Centro de Ciências Exatas
Departamento de Física

Trabalho de Conclusão de Curso

O Uso do Python no Ensino de Física: proposições para o ensino superior

Acadêmico: Luiz Fernando Menezes Pantaleão

Orientador: Anuar Mincache

Co-orientador: Fernando Carlos Messias Freire

Maringá, 29 de janeiro de 2025

Universidade Estadual de Maringá
Centro de Ciências Exatas
Departamento de Física

Trabalho de Conclusão de Curso

O Uso do Python no Ensino de Física: proposições para o ensino superior

TCC apresentado ao Departamento de Física da Universidade Estadual de Maringá, sob orientação do professor Dr. Fernando Carlos Messias Freire, como parte dos requisitos para obtenção do título de bacharel em Física

Acadêmico: Luiz Fernando Menezes Pantaleão

Orientador: Anuar Mincache

Co-orientador: Fernando Carlos Messias Freire

Maringá, 29 de janeiro de 2025

Sumário

Agradecimentos

Este trabalho é uma composição coletiva. Homenageio, aqui, a Física, que me acompanhou em tantos momentos de solitude e aumentou o poder de minha lucidez quanto às coisas do mundo material, ou seja, do tudo. Agradeço a todos meus companheiros e companheiras que alimentaram minha consciência com palavras positivas, de tal modo que me ajudaram a contrapor todo descrédito que eu dava à minha vontade de exaltar o simples. Este trabalho será simples, e por isso altruísta, porque os motivos de minhas incursões conceptuais em busca de sentido, como num feitiço de ourobouros, me levaram de volta à minha infância, à origem do pensamento sofisticado. Agradeço Rodrigo Azzolini, um pragmático de carteirinha, especialista em desbravar minhas lógicas junto comigo. Agradeço minhas amigas e amigos, e minha *ethernal wife*, Flaviana Bersan, que sempre esteve comigo quando precisei. A minha mãe e meu pai: sem vocês eu não conseguiria. Também agradeço a meu orientador por acompanhar meu conturbado processo criativo. Por fim, agradeço às poesias de Nietzsche, fundamento da real crítica à ciência.

Resumo

A inclusão tecnológica no ensino tem se tornado ainda mais relevante no contexto pós-pandemia de COVID-19. Com base na semiótica social, que sugere que a assimilação de um conceito é facilitada quando apresentado por múltiplos recursos e estímulos, este trabalho propõe o uso da linguagem Python no ensino de Física. Escolhemos o Python por sua sintaxe simples, facilidade de aprendizado e versatilidade na criação de ferramentas interativas, como gráficos, simulações e modelos computacionais. A fundamentação teórica aborda os conceitos de recursos semióticos, enfatizando como o uso de algoritmos pode não apenas criar representações diversificadas, mas também estimular a compreensão científica ao codificar os mecanismos por trás desses recursos.

Neste estudo, apresentamos propostas de uso do Python na produção de recursos didáticos, com foco na criação de redes neurais do zero para resolver problemas físicos, como a regressão para determinação de temperatura a partir de imagens. Concluímos que essa abordagem, ao unir tecnologia e semiótica social, demonstra grande potencial para enriquecer o ensino de Física e promover uma aprendizagem mais dinâmica e significativa.

Palavras-chave: física, computação, python, redes neurais.

Abstract

It's well documented that the covid-19 pandemic brought the necessity to use technology in teaching as a must. It's well understood, based on social semiotics, that programming has the potential to allow multiple views of things. An algorithm can be used to question itself, meaning that programming allows the ability to check something as a scientific approach. We focus our attention on python, based on the fact that is one of the simplest and easiest languages to built examples and explore the potentiality of its use in physical education. Our study, after exploring this potential, confirms the superiority of such language in comparison to traditional methods on certain approaches.

keywords: physics, computational physics, computer vision, neural networks, python, machine learning.

Introdução

Nos últimos anos, a utilização e disponibilidade das Tecnologias da Informação e Comunicação (TIC) têm experimentado um aumento significativo. Esse crescimento foi ainda mais acelerado pela pandemia de COVID-19, que resultou na suspensão das aulas presenciais, obrigando a transição para o ensino remoto. Isso gerou a necessidade de encontrar meios que possibilitassem a continuidade do ensino e aprendizagem de forma remota. Como resultado, as escolas começaram a incorporar diversas ferramentas tecnológicas cibernéticas [1].

Segundo a revisão de [2], que analisou trabalhos publicados, entre 2018 a 2022, em revistas de ensino de Física: é notável a falta de trabalhos que focam na utilização de linguagens de programação, atualmente, no ensino de Física.

A grande maioria dos trabalhos analisados nesta referida pesquisa focam na utilização de softwares e aplicativos prontos para serem utilizados como ferramentas de experimentação digital, ou também, na utilização de ferramentas como o Arduino, caso em que os alunos não atuam na construção ou no desenvolvimento da lógica de programação. Neste referido ínterim, constata-se apenas um trabalho que conduziu seus estudos utilizando a racionalização da linguagem de programação com o propósito final de desenvolver um algoritmo que representasse a experimentação de um fenômeno físico por meio de uma simulação computacional, com ênfase no processo de ensino-aprendizagem.

De acordo com [3], o aprendizado pode ser ainda mais enriquecido quando os alunos assumem a responsabilidade pelo desenvolvimento dos simuladores. Isso fortalece a compreensão aprofundada do tema em questão. Nesse contexto, surge a oportunidade de explorar o uso da linguagem de programação como uma ferramenta algébrica, potencializando a racionalização dos estudantes por meio da lógica de programação para resolver problemas ou conceituar fenômenos físicos.

Por fim, a linguagem de programação em si possui um potencial significativo a ser explorado, uma vez que requer a racionalização de conceitos lógicos que podem impactar diretamente na assimilação de princípios físicos. Este processo não exclui a possibilidade de gerar um produto final, como simulações computacionais. A partir dessas considerações, é possível conjecturar a criação de novas abordagens para atividades e avaliações no ensino de Física, bem como o desenvolvimento de conteúdos interdisciplinares nos quais a linguagem de programação atue como uma ferramenta didática para potencializar o ensino dessa disciplina. Este trabalho focas seus esforços neste tema, propondo o uso do Python no Ensino de Física. A escolha pela linguagem Python se dá justamente por essa ser uma linguagem simplificada, que possui curva de aprendizado favorável a iniciantes em programação [4].

Fundamentação Teórica

O objetivo deste trabalho é, partindo de uma análise da programação como um fenômeno e sua utilidade específica no ensino de Física, propor exemplos de algoritmos que possam ser trabalhados em sala, a nível de Ensino Superior.

A combinação de codificação, visualização e atividades interativas confere à programação a versatilidade para ser utilizada em diversos campos da pesquisa em Física, tais como cosmologia, dinâmica de fluidos e física atômica. Ao estudar não apenas o código, mas também a visualização e a interação com o programa, é possível obter uma compreensão muito mais rica do uso potencial da programação. Essa visão mais abrangente da programação será, neste trabalho, analisada sob a teoria da semiótica social [5].

1.1 Semiótica Social

A semiótica social é um ramo da semiótica construído em torno da compreensão e investigação da criação de significado em grupo e dos recursos utilizados para criar significado por meio da comunicação. Esses recursos são chamados de recursos semióticos e englobam representações, ferramentas e atividades usadas para criar ou derivar significado em grupos especializados.

Utilizando essa definição, podemos analisar a programação como um meio de comunicação entre o estudante e o programa. Um estudante pode fazer perguntas a um programa para obter uma resposta ou como meio de construir novas representações ou ferramentas. No entanto, a programação não é um recurso semiótico; em vez disso, deve ser vista como um sistema semiótico [5], porque a programação pode ser usada para descrever muitos cenários diferentes e extrair respostas variadas para muitas perguntas diferentes.

Um recurso semiótico é usado em um cenário específico para transmitir um significado específico (exemplos: um gráfico tempo-velocidade, um diagrama de um circuito, etc.). Um sistema semiótico é um sistema de comunicação qualitativamente diferente de outros meios de comunicação. O sistema de comunicação “imagem“ é um sistema semiótico qualitativamente diferente de “texto”, que é outro sistema semiótico. No entanto, o texto pode ser usado para transmitir significados diferentes em situações diferentes: quando um sistema semiótico é aplicado em um cenário específico, um recurso semiótico é criado ou extraído dele. Um autor usa texto para escrever um livro; o livro é o recurso semiótico e o texto é o sistema semiótico usado para produzir o recurso semiótico.

Muitos sistemas semióticos diferentes podem ser usados em conjunto para criar um único recurso semiótico, como a ideia deste trabalho, que utiliza os sistemas semióticos texto e imagem para transmitir significado de maneira relevante para a disciplina. A programação pode ser descrita como um sistema semiótico usado para criar ou investigar outros recursos semióticos. Ao usar a programação, é possível mover-se entre diferentes recursos semióticos e entre diferentes sistemas semióticos, como tomar uma longa lista de pontos de dados como entrada e produzir, por exemplo, uma imagem. Neste caso, o programador transformou um recurso semiótico (uma

lista específica de pontos de dados) em um sistema semiótico (lista de números ou dados) para outro recurso semiótico (uma imagem específica) em outro sistema semiótico (imagens). Esse tipo de transformação é chamado de transdução dentro do quadro da multimodalidade [6], ou uma "re-representação".

As transduções são importantes no ensino de Física porque forçam os alunos a discernirem os aspectos relevantes de um determinado tema, mesmo que representados de inúmeras maneiras diferentes. As transduções podem ser complicadas ou difíceis de compreender, e os alunos devem ter tempo para explorá-las e compreendê-las. A programação é adequada para transduções controladas pelos alunos, pois eles realizam a transdução a cada etapa da implementação, desde o modelo matemático inicial até a visualização na tela. A importância do uso de múltiplas representações para aprimorar a aprendizagem foi explorada por, por exemplo, [7] e [8], que descobriram que o uso de múltiplas representações desempenha um papel importante na aprendizagem do aluno.

1.2 Atribuições e Programação

"Atribuição" (tradução livre de "affordance" [5]) é um termo usado para descrever os diferentes significados em potencial que podem ser extraídos de um objeto, ou estímulo, por um agente. Se um estudante interage com uma garrafa, por exemplo, pode, a partir dessa interação, significar a ideia de: "beber", "despejar", ou, se a garrafa estiver vazia, de "jogar fora", ou "reciclar", ou "encher" a garrafa. Esses são exemplos de atribuições de um mesmo objeto. No entanto, se outro estudante interagir com a garrafa, pode extrair outros significados da garrafa. As coisas que uma garrafa pode significar, ou querer dizer, ao segundo estudante diferem em comparação com o primeiro. Essa diferença pode ser explicada pela forma como os dois estudantes discernem diferentes atribuições. As atribuições discernidas por determinado sujeito dependem de seus conhecimentos prévios, personalidade, e muitos outros fatores: humor, ambiente em que se encontram, etc. A garrafa tem uma infinidade de atribuições, e aquelas atribuições que serão discernidas são aquelas que dependerão de um determinado sujeito ao interagir com o objeto em questão. A figura 1.1 abaixo representa como um agente experiencia atribuições a partir de um estímulo externo.

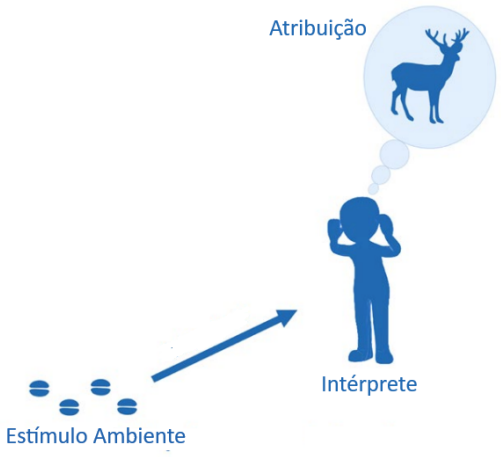


Figura 1.1: Figura que representa a relação entre indivíduo, estímulo ambiente e atribuição

Nesse contexto, a programação oferece ao estudante a oportunidade de modificar o código (representação textual de uma ideia capaz de gerar outras representações da mesma ideia) com a

intenção de aumentar a discernibilidade de diferentes aspectos, trazendo diferentes atribuições. O que um estudante discernirá é baseado em sua capacidade de extrair informações significativas da representação resultante; ao modificar a representação, pode-se discernir aspectos relevantes com mais facilidade. A programação também exige que cada parte da implementação seja explicitada, portanto, requer discernimento de suas diferentes partes, agregando à possibilidade de aprendizado.

A teoria das atribuições tem sido aplicada na ensino de Física por referências [9] [10] [11], sendo até mesmo conceitualizada de forma mais específica à área de Ensino, com referências ao uso de atribuições disciplinares [12] e atribuições pedagógicas [13], descrevendo quão bem um recurso semiótico pode ser usado, ou é usado, na disciplina ou como um recurso pedagógico.

Este trabalho utiliza o termo "atribuição", de acordo com a referência [5], para descrever qualquer coisa que um objeto significa a um agente. Assim, é possível adicionar e remover atribuições, bem como modificar as já existentes ao modificar um objeto. Esse uso de "atribuições" está muito mais próximo de como as comunidades de semiótica social e multimodalidade [29] [14] utilizam o termo, podendo ser interpretado como: "potencial de significado" de um objeto.

1.3 Recursos Semióticos e Atribuições no Ensino

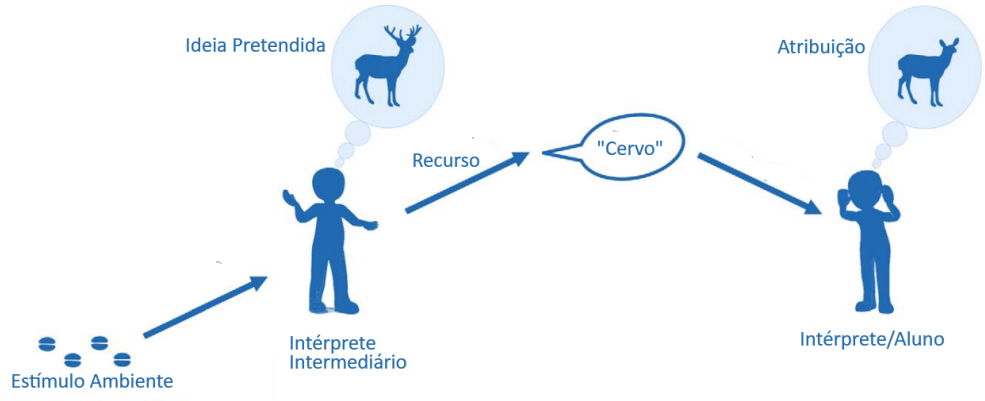


Figura 1.2: A figura ilustra como uma atribuição de um recurso pode estar ou não estar contida na ideia pretendida pelo interlocutor, comunicador do recurso

Recursos semióticos são utilizados no Ensino, e aprendizagem, para "iluminar" um significado específico a ser discernido de um objeto, conceito, ou representação; ou seja, um significado pretendido.

Avalia-se o entendimento de um conceito específico pelo estudante ao analisar o que um recurso semiótico oferece e o que o estudante pode discernir desse recurso semiótico.

Se um recurso semiótico, ao interagir com um agente, não oferecer uma atribuição que especifique o significado pretendido, esse agente não será capaz de discernir o significado pretendido a partir do recurso apresentado. No entanto, caso o recurso semiótico seja modificado, pode este adquirir a atribuição específica necessária para transmitir o significado pretendido, sendo então utilizado na comunicação e compreensão desse significado. Para ilustrar, tomemos o estudo de Ma [15], que destaca o papel distintivo do ábaco na educação chinesa. Essa ferramenta se revela como um fator crucial no ensino de abstração aritmética-matemática no Ensino Básico, apresentando uma abordagem notavelmente diferente da adotada nos Estados Unidos, que é bastante similar à do Brasil. Um recurso diferente, ábaco, em um sistema diferente, jogo

(manipulação mecânica), pode trazer atribuições significativamente diferentes e, muitas vezes, mais abstratas do que recursos de texto ou imagem, comumente usados nos Estados Unidos e no Brasil; explicando o sucesso da educação em matemática básica chinesa nos indicadores internacionais da Educação [16].

Consulte a Figura 1.2, que ilustra um sistema de troca de signos. Nesse sistema, o instrutor interpreta um determinado estímulo externo e, a partir dele, identifica uma ideia pretendida, a qual pode estar associada a uma ou até mesmo a diversas atribuições. Para transmitir essa ideia pretendida, o instrutor deve escolher um meio apropriado, um recurso que, no caso da figura, pode ser um texto verbal (como palavras escritas ou faladas) ou um texto não-verbal (como imagens, gestos ou até mesmo áudio). O estudante, ao receber o estímulo gerado por esse recurso, associará diferentes atribuições à mensagem e, geralmente, escolherá uma delas como foco principal. Como ilustrado no diagrama, mesmo uma mensagem simples transmitida por um recurso explícito, como literalmente dizer "cervo", pode não ser suficiente para comunicar com precisão a ideia pretendida.

Uma modificação em um recurso semiótico pode ser algo tão simples quanto uma pessoa dizer: “esta caneta é um vetor”, em vez de: “imagine um vetor”. Essa abordagem permite ao estudante identificar aspectos relevantes de um vetor com base na caneta.

Ainda que múltiplos recursos semióticos estejam disponíveis para transmitir o mesmo significado (ou ideia), a transição de um recurso para outro pode ocasionar alterações nas atribuições associadas. Essa relação entre recurso, sujeito e suas atribuições derivadas pode ser melhor compreendida por meio do diagrama apresentado na Figura 1.3.

1.4 Modificando Recursos Semióticos

Sempre que um recurso semiótico é modificado – seja pela adição de cor, descrição ou pela transformação em algo completamente diferente –, as atribuições associadas a ele são alteradas ou ajustadas. Essa mudança pode aumentar, diminuir ou até mesmo eliminar a discernibilidade de um significado potencial. A programação, por sua vez, permite que o estudante modifique o que cria de qualquer maneira que considere apropriada, desde que possua as habilidades e o conhecimento necessários. Como efeito secundário, essa modificação permite ao estudante explorar diversas atribuições associadas ao que foi criado, de forma que atenda aos seus objetivos.

Isso permite que o estudante crie recursos que auxiliem na percepção de significados específicos, alinhados tanto às suas próprias aspirações quanto aos objetivos da matéria. Ao interagir com esses recursos, o estudante pode estabelecer atribuições específicas que convergem para o conhecimento pretendido. Caso um recurso não seja suficientemente claro em seu significado, o estudante pode modificá-lo, gerando um novo recurso e, assim, aumentando a discernibilidade de um significado específico, tornando-o mais perceptível.

Como exemplo, pode-se observar a abstração excessiva que estudantes do Ensino Médio frequentemente associam a símbolos matemáticos ou expressões algébricas mais extensas. O significado pretendido não pode ser atribuído se o estudante não estabelecer uma relação significativa com o recurso por meio de suas próprias experiências. Uma estratégia inicial para mitigar esse problema é apresentar a mesma ideia utilizando múltiplos recursos. Contudo, essa abordagem tende a posicionar o estudante como um receptor passivo de conhecimento. Nesse contexto, a programação surge como uma ferramenta para capacitar o estudante a reconhecer e explorar as possibilidades de múltiplas representações de maneira ativa.

Consulte a figura 1.4 para uma demonstração visual de como uma mudança no recurso semiótico também altera a facilidade com que um significado específico pode ser extraído. Na Figura 1.4, um espaço vetorial é representado de duas maneiras, nenhuma informação nova foi

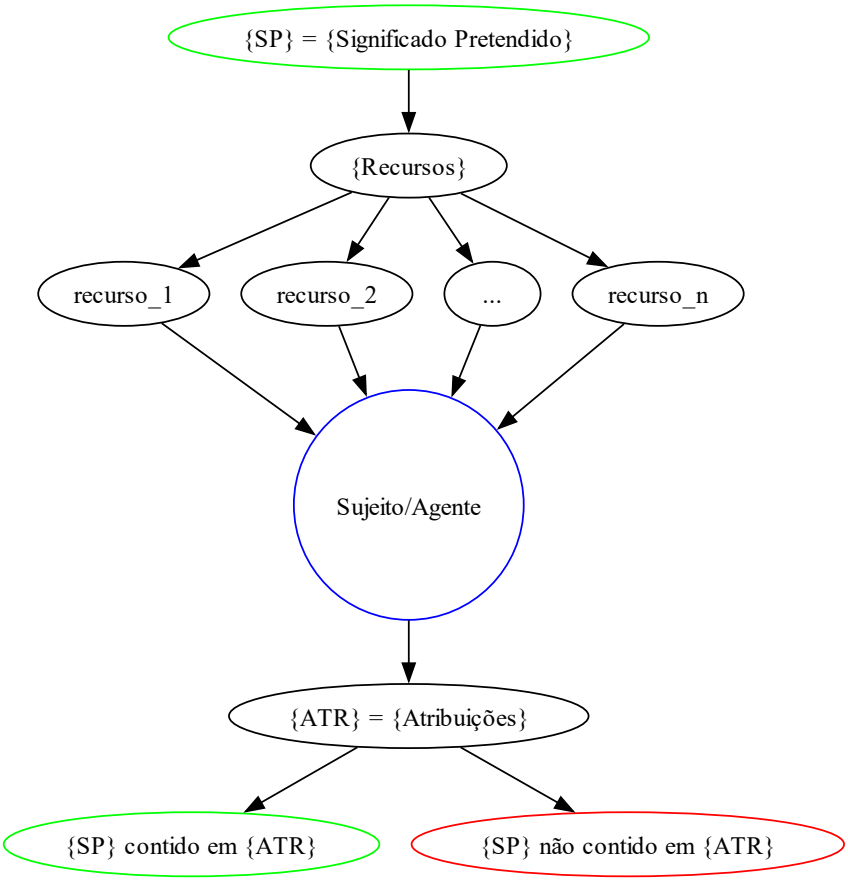
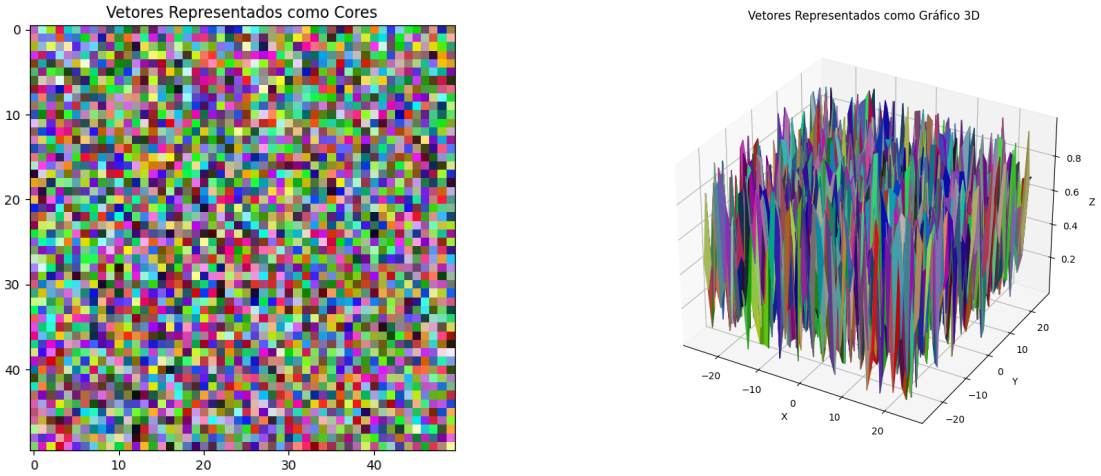


Figura 1.3: Mapa mental ilustrando como um sujeito pode ou não compreender um significado pretendido. Se as atribuições não forem feitas, o recurso não proverá o significado pretendido àquele sujeito

adicionada na transformação, apenas como a informação foi representada. As "atribuições" são distintas, mas relacionadas à mesma informação e significado contidos dentro de um recurso semiótico. No entanto, vale ressaltar que modificar as atribuições de um recurso semiótico não é uma arte precisa e é principalmente orientada por conjecturas e suposições fundamentadas [5].

No que tange a teoria de multimodalidade [29] [14], é sempre possível alterar uma representação modificando-a, ou re-representando-a. Se a mudança ocorrer dentro do mesmo modo, como reescrever um texto, transformando-o em outro texto, essa mudança é denominada transformação [29]. Se a mudança levar a representação de um modo para outro, como mover-se de texto corrido para um diagrama, ou uma animação, essa mudança é denominada transdução [29] [17]. A semiótica social adota esses termos e os utiliza para se referir a diferentes tipos de mudanças nos recursos semióticos. A importância dessas mudanças ou modificações pode ser compreendida a partir da teoria da variação da aprendizagem.



(a) imagem bidimensional

(b) gráfico tridimensional

Figura 1.4: As figuras (a) e (b) representam o mesmo espaço vetorial, ou seja, a mesma informação, mas ambas podem "iluminar" conceitos específicos diferentes sobre, por exemplo, espaço vetorial

1.5 Teoria da Variação da Aprendizagem

A teoria da variação da aprendizagem, proposta por F. Marton, [18] [38,45] afirma que para aprender algo, esse algo deve primeiro ser discernido como seu próprio aspecto, e para discerni-lo, o estudante deve experimentar variações em relação a um fundo estático nesse aspecto.

Marton utiliza um exemplo esclarecedor sobre o aprendizado de cores para ilustrar essa ideia. É por meio da variação, em contraste com um fundo estático, que uma cor específica se destaca e pode ser discernida. Somente ao compará-la com aquilo que ela não é, a cor pode ser identificada como algo que é. Assim, ao variar o aspecto que o estudante deve aprender, esse aspecto torna-se discernível, permitindo o aprendizado. Além disso, múltiplas representações e contrastes possibilitam o desenvolvimento de discernimentos congruentes.

Por exemplo, um estudante daltônico não atribuirá os mesmos significados a recursos visuais que um estudante não daltônico atribuiria. Nesse caso, outros recursos precisam ser empregados para alcançar congruência nas atribuições esperadas. Se uma cor, por exemplo, for definida por um conjunto de valores (a, b, c), em que a representa a porcentagem de vermelho, b de verde e c de azul, surgirão congruências perceptíveis entre estudantes daltônicos e não daltônicos. A modificação do recurso, nesse caso, é capaz de criar atribuições congruentes ao superar limitações individuais de percepção.

A programação, por sua vez, permite a introdução de variações de forma rápida e eficiente em diferentes variáveis e estruturas. Ao alterar, por exemplo, a massa de partículas em uma simulação, o estudante pode discernir diretamente os efeitos dessa mudança: talvez a partícula afunde, talvez flutue. Essa variação é experimentada e, potencialmente, compreendida. Além disso, novas questões podem surgir à medida que antigas são respondidas.

A programação não apenas facilita a modificação de variáveis, mas também transforma a forma como o estudante interage com a simulação e como a simulação é representada. Ela oferece amplas oportunidades e métodos quantificáveis para explorar novas dimensões de variação, permitindo que o estudante investigue e compreenda essas dimensões de forma dinâmica

e interativa.

1.6 Os Recursos Tradicionais

Definir o significado de uma informação é uma tarefa intrincada, especialmente quando consideramos exemplos como a equação:

$$f(x) = x^2 \tag{1.1}$$

A particularidade desta representação matemático-textual reside na sua capacidade de representar, para contextos distintos, diversas ideias.

Isso evidencia que não é possível delimitar isoladamente o que essa função representa, pois a mesma expressão pode adquirir significados variados dependendo do contexto. Além disso, não é razoável presumir que um estudante atribuirá o mesmo significado a 1.1 e a expressões como y(x) = x² ou g(x) = x².

A simples alteração textual de 1.1 para:

$$x(t) = t^2 \tag{1.2}$$

à primeira vista, pode evocar atribuições provenientes de áreas diversas, destacando a sensibilidade do contexto.

O elo entre essas representações é o fato de todas elas traduzirem o mesmo conceito, entretanto, cada uma desencadeará atribuições distintas e resultará em compreensões singulares.

No contexto do Ensino Tradicional e de seus recursos didáticos, observa-se uma utilização notória de elementos textuais e visuais, frequentemente representados por gráficos, na tentativa de conferir atribuições esclarecedoras a conceitos matemáticos. Entretanto, de maneira geral, esses recursos se revelam insuficientes e, muitas vezes, pouco elucidativos para os alunos. A descrição de um conceito, como por exemplo: $f(x) = x^2 = y =' \ parabola'$, não necessariamente resulta no discernimento almejado.

Frequentemente, essas múltiplas representações acabam por distanciar o estudante do conhecimento desejado, apresentando-se de maneira externa, sem a devida participação ativa do aluno. A afirmação "isto é x, que também é z, e que também é a; portanto, x = z = a"não surge como resultado de uma investigação autônoma do aluno, mas como uma proposição baseada na autoridade do professor.

Na programação, algo semelhante ocorre. Inicialmente, estamos presos a um sistema textual e, somente após adquirir o entendimento necessário da sintaxe, conseguimos navegar entre múltiplas representações e observar o output. No entanto, diferentemente dos recursos tradicionais, a programação permite validar lógicas diretamente. Por exemplo, uma afirmação como x² = x * x pode ser verificada programaticamente com x2 == x * x. Além disso, o comportamento de uma função pode ser testado ao escrever seu código e verificar como ela responde a diferentes entradas, utilizando algoritmos.

Nesse contexto, esta proposta didática adota o uso básico de Python, incorporando a ideia de notação orientada a objetos. Inicialmente, abordaremos a concepção de algoritmos. Posteriormente, após apresentar a sintaxe da linguagem, introduziremos a necessidade de conceber certas variáveis como objetos dotados de características e atributos. Somente então avançaremos para a codificação de algoritmos, que, nesse estágio, podem ser tanto propostas dos alunos quanto do instrutor.

1.7 A Programação Enquanto Construção de Significado

A programação pode ser utilizada como uma ferramenta poderosa para a construção de significado no ensino de Física, de maneira semelhante à matemática, que também é usada para investigar e compreender conceitos dessa área. Por meio do ato de implementar códigos, as ideias e modelos dos estudantes são tornados explícitos e necessariamente decompostos em partes menores e compreensíveis, que podem ser posteriormente reunidas para formar o modelo ou a ideia completa. Essas partes podem ser modificadas de duas maneiras: interna e externamente. A modificação interna altera como uma parte funciona, como ao modificar a interação entre partículas. Já a modificação externa refere-se a como as diferentes partes se conectam, em que ordem são organizadas e chamadas.

Por exemplo, a afirmação "A energia do sistema é conservada"representa uma parte externa, pois fornece informações sobre como o sistema interage com o ambiente externo, mas não detalha a natureza da energia no interior do sistema. Por outro lado, a parte interna descreve os componentes da energia no próprio sistema, como energia potencial, cinética, térmica ou química, e como essas formas de energia se transformam entre si.

A programação oferece aos estudantes meios de explorar tanto os aspectos externos quanto internos dos conceitos que estão implementando. Dessa forma, é possível investigar quais fenômenos surgem naturalmente e quais informações precisam ser explicitamente inseridas para construir e compreender o modelo de maneira mais completa.

1.8 Introdução ao Python como Ferramenta de Ensino em Física

O Python é uma linguagem de programação amplamente utilizada, especialmente em áreas como a Física, devido à sua simplicidade e à disponibilidade de bibliotecas específicas para cálculos científicos e visualizações gráficas. Nesta seção, apresentaremos os conceitos básicos do Python que serão utilizados ao longo desta proposta.

Executando um Código em Python

Para executar um código Python, pode-se optar por diversas abordagens. Neste trabalho, focamos em duas:

- **Usando o Sublime Text:** Crie um arquivo com a extensão `.py` no Sublime Text. Após escrever o código, salve o arquivo ao pressionar `Ctrl+S`.
- **Usando o Terminal:** Navegue até o diretório onde o arquivo `.py` está salvo e execute o comando `python nome_do_arquivo.py` ou `python3 nome_do_arquivo.py`, dependendo da versão instalada.

Conceitos Fundamentais do Python

Strings

Qualquer texto, em linguagem Python, é representado entre aspas duplas ou aspas simples:

```
texto = "HelloWorld"
texto = 'HelloWorld'
```

Listas

Listas são estruturas de dados que armazenam múltiplos valores em uma única variável. Exemplo de criação e manipulação de uma lista:

```
# Criando uma lista
dados = [1, 2, 3, 4, 5]

# Acessando elementos
o_primeiro_elemento = dados[0]

# Adicionando um elemento
dados.append(6)

# Modificando um elemento
dados[0] = -1
```

Laços de Repetição (*For Loops*)

Os laços `for` permitem executar um bloco de código para cada elemento de uma sequência. Por exemplo:

```
# Iterando sobre uma lista
for numero in dados:
    dados[0]=numero
```

Funções

Python possui funções nativas, como as funções print, type, sum, etc. Além destas funções, pode-se criar uma função própria ou importá-la de alguma biblioteca. Funções customizadas permitem reutilizar blocos de código. Abaixo está um exemplo de uma função customizada que calcula a soma de dois números:

```
# Definindo uma funcao
def soma(a, b):
    return a + b

# Chamando a funcao
resultado = soma(3, 4)
# Chamando funcao nativa
print(resultado)
```

Importando Bibliotecas

Como mencionado, funções podem ser importadas. Bibliotecas são coleções de funções, classes e métodos. Python possui bibliotecas nativas e não-nativas. A grande vantagem desta linguagem é a simplicidade para instalar bibliotecas não nativas.

Numpy

Para instalar uma biblioteca não-nativa no Python, no terminal, escreve-se:

```
pip install biblioteca
```

No caso do Numpy:

```
pip install numpy
```

A biblioteca Numpy é usada para operações matemáticas e manipulação de arrays, e pode ser importada da seguinte maneira:

```
import numpy as np
```

Exemplo de uso:

```
import numpy as np

# Criando um array
array = np.array([1, 2, 3, 4, 5])

# operacoes matematicas
array_dobrado = array * 2
```

Matplotlib

A biblioteca Matplotlib é não-nativa e muito usada para visualização de dados. No terminal, instalamos-a desta maneira:

```
pip install matplotlib
```

E o importamos:

```
import matplotlib.pyplot as plt

# Dados para o grafico
x = np.array([0, 1, 2, 3, 4])
y = x**2

# Criando o grafico
plt.plot(x, y, label='y = x^2')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Grafico Exemplo')
plt.legend()
plt.show()
```

Objetos

Um exemplo de construção de significado físico em Python pode ser observado em cinemática.Nos recursos tradicionais, o conceito de massa pontual é apresentado como o "sujeito"das equações de movimento, enquanto as variáveis posição, velocidade e aceleração são tratadas separadamente como elementos que "regulam"o funcionamento dessas equações. Como os recursos utilizados geralmente se limitam a textos e imagens (principalmente gráficos), é difícil

fugir dessa estrutura tradicional de apresentação. Nesse contexto, o movimento de uma massa pontual é descrito em torno da equação de movimento (representada textualmente) e das variáveis textuais (x, v, a).

Na programação, o conceito de "objeto"oferece uma abordagem alternativa para iniciar uma investigação da cinemática, que precede a apresentação das equações de movimento. Por exemplo, podemos instanciar um objeto utilizando o seguinte pseudo-código:

```
massa_pontual -> objeto(massa, posicao, velocidade):
    objeto.massa = massa
    objeto.posicao = posicao
    objeto.velocidade = velocidade
```

E construir diversos objetos que possuam atributos distintos:

```
massa_pontual_1 = massa_pontual(1, 0, 1)
massa_pontual_2 = massa_pontual(0, 0, 0)
massa_pontual_3 = massa_pontual(1, 1, 1)
```

Esses objetos nos levam a várias perguntas relevantes: O que esses objetos representam? Como podemos escolher uma forma de "enxergar"o comportamento deles? Se eles possuem velocidade, então eles se movem; mas se movem em relação a quê? Além disso, como os atributos que definimos interagem entre si? Por exemplo:

A posição está relacionada a qual referência? A velocidade é medida em relação a quê?

Essas questões colocam em debate aspectos da cinemática que, frequentemente, não são explorados pelos recursos visuais tradicionais [5].

Este trabalho utiliza apenas os conceitos e bibliotecas mencionados anteriormente para propor estratégias didáticas no ensino com programação. No capítulo seguinte, apresentaremos a metodologia adotada como guia para essas estratégias, visando explorar o potencial da programação como um valioso instrumento didático.

1.9 Álgebra

Álgebra é o ramo da matemática que estuda estruturas, relações e operações, utilizando símbolos para representar números e conceitos, permitindo generalizações e resoluções de problemas em contextos abstratos e concretos. De origem árabe, significa ”redução”, e é o cerne da ligação que desejamos estabelecer entre semiótica social, programação e método científico.

Um programa em Python interpreta o código sequencialmente, analisando cada linha para identificar variáveis. Essas variáveis são associadas a endereços de memória onde os valores correspondentes são armazenados. Durante a execução, o interpretador consulta uma estrutura chamada tabela de símbolos para localizar essas associações, aplica as operações definidas no código aos valores correspondentes e, em seguida, retorna o resultado, seja exibindo-o na saída padrão, armazenando-o em outra variável ou realizando outras ações definidas pelo programa. De certa forma, um algoritmo é um método algébrico, e por isso sua ligação com o processo científico é feita: se um algoritmo descreve relações cientificamente comprovadas, então é uma simulação. Uma simulação é um ambiente que obedece a regras determinadas. As regras são o algoritmo, e a concretude do código em execução é a simulação do sistema descrito pelo algoritmo.

Incremento

Posteriormente, desenvolvemos um raciocínio algébrico que envolve a ideia de incremento. Embora anacrônico, há de se pensar na possibilidade do conceito de limite ter existido antes da sua formalização por Newton e Leibniz. Arquimedes utilizava o conceito de exaustão, que envolve imaginar um número alcançando um valor muito próximo de um valor limite.

Considere uma função de uma variável:

$$y = f(x)$$

Definimos um incremento, s, como a resposta de uma função para uma mudança ínfima na sua variável, estabelecendo a razão:

$$s = \frac{\Delta y}{\Delta x}$$

Seja f uma reta:

$$f(x) = m * x + b$$

então seu incremento é

$$s = \frac{\Delta y}{\Delta x} \tag{1.3}$$

$$s = \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{1.4}$$

$$s = \frac{m * (x + \Delta x) + b - m * x - b}{\Delta x} \tag{1.5}$$

$$s = \frac{m * \Delta x}{\Delta x} \tag{1.6}$$

Em álgebra, qualquer divisão que não for por zero é permitida. Seja Δx um incremento ínfimo, então

$$0 < \Delta x << 1$$

portanto s é definido para a reta:

$$s = m \tag{1.7}$$

Utilizaremos esta técnica para obter incrementos de funções, baseada em álgebra, com o intuito de manter os temas acessíveis para, também, alunos iniciantes no Ensino Superior.

Metodologia

A presente pesquisa tem como objetivo o desenvolvimento de algoritmos em linguagem Python para enriquecer o ensino de Física. Esse esforço fundamenta-se em diversas obras que, por meio da aplicação da semiótica social e da multimodalidade, destacam a programação como uma ferramenta essencial para o aprimoramento do ensino das ciências. Ao possibilitar a criação de algoritmos, a programação emerge como um instrumento singular de avaliação, permitindo a formulação de perguntas variadas ao algoritmo e o discernimento de diferentes nuances de um determinado conceito.

Neste estudo, emprega-se o termo "atribuições"para denotar as interpretações que um sujeito pode extrair de um recurso semiótico, o qual pode incluir imagens, textos, animações, música ou qualquer outro meio capaz de transmitir uma ideia. A metodologia adota a concepção de um "sujeito ideal", representado, neste contexto, por um estudante do Ensino Superior dedicado ao aprendizado de Física. A partir dessa construção imaginativa, busca-se identificar os tipos de recursos semióticos que possam suscitar atribuições benéficas para o objetivo principal do ensino: a construção de um raciocínio investigativo.

Com o sujeito ideal como referência, são desenvolvidos algoritmos que exemplificam recursos semióticos úteis. Ao apresentar esses algoritmos, destaca-se a natureza multimodal da programação e sua capacidade de gerar múltiplas saídas sobre um mesmo conceito. Vale ressaltar que a programação vai além dessa função, podendo inclusive ser utilizada para investigar sua própria estrutura e funcionamento.

O embasamento teórico, delineado no capítulo anterior, desempenha um papel central na estruturação desta pesquisa. Fundamentado na busca por múltiplas representações até que a atribuição desejada seja alcançada, esse alicerce teórico posiciona o professor ou instrutor mais como um facilitador de estímulos e conexões do que como um avaliador convencional. Sob essa perspectiva, a abordagem proposta assume uma natureza interdisciplinar, incorporando e valorizando diversas formas de recursos semióticos. No contexto da programação, a geração de um output exige um conhecimento detalhado dos inputs, incentivando o aluno a realizar re-representações e transduções sem se afastar do tema em análise, seja uma equação expressa em linguagem textual ou um gráfico bidimensional.

Nesse sentido, propõe-se a criação de algoritmos que, a partir de recursos provenientes de manuais tradicionais de ensino, busquem gerar e explorar múltiplas representações desses recursos. Tal abordagem fomenta a diversidade de perspectivas e se alinha ao contexto educacional, permitindo aos alunos desenvolverem habilidades de reinterpretação, adaptação e transposição de conhecimentos sem se desviar do tema principal.

A interseção entre a riqueza semântica dos materiais didáticos convencionais e a versatilidade da programação configura-se como um terreno fértil para o desenvolvimento de estratégias pedagógicas inovadoras. Essa abordagem promove a conexão entre diferentes formas de representação e fortalece o aprendizado ativo e investigativo.

2.1 A Busca por Múltiplas Representações

A exploração de múltiplas representações exige um sólido domínio por parte do estudante, tanto em lógica de programação quanto nos conteúdos programáticos tradicionais. Nesse sentido, propõe-se uma abordagem educacional que integra essas duas vertentes de forma complementar. A escolha da linguagem Python revela-se estratégica, pois sua sintaxe simplificada facilita a transição para a compreensão dos fundamentos da programação. Inicialmente, o uso de algoritmos em pseudocódigo promove uma familiarização rápida com conceitos básicos, preparando o estudante para adquirir fluência na linguagem Python.

Com o domínio inicial da linguagem estabelecido, o próximo passo é explorar diversas representações dos problemas apresentados. Este trabalho não busca impor uma estrutura didática rígida, mas sim propor um esqueleto flexível que sirva como base para estratégias pedagógicas versáteis. Essas estratégias têm como objetivo explorar integralmente o potencial da programação em sala de aula, permitindo que os estudantes desenvolvam suas próprias abordagens ao lidar com os desafios propostos.

Este enfoque metodológico, centrado na dualidade entre lógica de programação e conteúdo disciplinar, vislumbra catalisar uma sinergia entre os aspectos técnicos da programação e a compreensão conceitual da matéria específica, proporcionando aos estudantes uma experiência de aprendizado enriquecedora e integral. A linguagem Python, ao ser adotada como veículo dessa integração, proporciona um ambiente propício ao desenvolvimento cognitivo, incentivando a abstração e a resolução de problemas de forma eficaz.

A Proposta

3.1 Técnicas de Investigação em Programação

Neste trabalho, utilizaremos apenas duas bibliotecas do Python para desenvolver os raciocínios e investigações. Numpy: uma biblioteca que permite trabalhar com vetores. Qualquer numpy.array é considerado um vetor, e sua dimensão é definida pela quantidade de elementos fornecidos como entrada. Matplotlib: uma biblioteca gráfica que possibilita a criação de gráficos, facilitando a investigação e visualização de conceitos.

Python, devido à sua sintaxe simplificada, apresenta grande potencial como ferramenta de aprendizado. Essa simplicidade permite que estudantes se concentrem nos conceitos e na lógica, sem serem sobrecarregados por complexidades técnicas da linguagem.

Como mencionado nos capítulos anteriores, os recursos semióticos são múltiplos, e a programação se destaca por permitir transposições rápidas entre esses recursos. Em um ambiente de sala de aula, é possível propor múltiplas representações para uma mesma ideia, estimulando diferentes percepções nos alunos e facilitando a assimilação de conteúdos.

Adicionalmente, as propostas didáticas apresentadas neste trabalho são intencionalmente flexíveis, permitindo ajustes de acordo com as interações em sala, dificuldades conceituais encontradas ou outros fatores contextuais. Com essa abordagem, introduziremos exemplos que demonstram como usar código para investigar conceitos.

Exemplo: Investigação de um Problema

Suponha o seguinte problema:

"Se um objeto se move em linha reta e sua posição em função do tempo é descrita por uma função seno no intervalo de 0 a 90 graus, qual seria a forma da velocidade desse objeto?"

A programação permite abordar questões como essa de maneira prática e visual, utilizando código para explorar conceitos de forma interativa e experimental. Nos próximos exemplos, exploraremos como usar Python, com o auxílio das bibliotecas Numpy e Matplotlib, para investigar esse e outros problemas de forma didática.

Através de um código, podemos transpor o problema para um problema em código. Em verdade, ao codificar um problema, escrevendo seu funcionamento e suas partes, aparece a possibilidade de interagir com este, e ramificá-lo a outros problemas, transpondo um evento a outro através da coesão da evolução do próprio código, customizando-o ou introduzindo lógicas adicionais.

Se buscamos codificar a situação do problema, ou seja, escrever o estado de movimento do objeto enquanto um algoritmo, pode-se fazê-lo do seguinte modo:

```
import numpy as np
import matplotlib.pyplot as plt

angulos = np.linspace(0, 3.14, 100)
pos = np.sin(angulos)
```

```
x = angulos # vals do eixo x
y = pos # vals do eixo y
plt.plot(x, y)
plt.xlabel('angulo')
plt.ylabel('posicao')
plt.show()
```

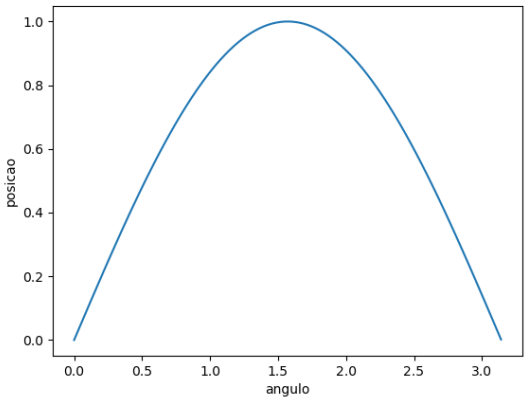


Figura 3.1: Gráfico produzido pelo código acima. No eixo x, ângulo; e no eixo y, posição

Quando um problema é transformado em algoritmo, suas partes são definidas em uma sequência lógica que, quando executada, resulta em um sistema definido por variáveis iniciais. Devido a esse fato, pode-se concluir que, se de um sistema são derivadas outras variáveis, então codificar o sistema e depois codificar uma função, ou método, que nos leve a estas variáveis, é um processo científico. Se nosso algoritmo é respaldado por definições concretas, então nosso sistema caracteriza uma simulação.

Podemos investigar a velocidade do objeto lembrando que, para cada pequeno deslocamento que o objeto realiza em um intervalo de tempo, sua velocidade naquele instante é definida por:

$$v(t) = \frac{\Delta x}{\Delta t} = \frac{\Delta x(t) - \Delta x(t - \delta t)}{\Delta t}$$

A ideia de comparar um pequeno intervalo no espaço da posição com um pequeno intervalo no espaço do tempo é concreta. Se temos definido o espaço como uma sequência de passos, então o cálculo da velocidade em cada instante de tempo é possível, desde que respeitemos a escala que criamos quando definimos nosso espaço. Ao definir o espaço de posição como composto por 100 pontos; e ao definir o espaço do tempo como um intervalo de 0 a 3.14, então satisfazemos a equação da velocidade instantânea definindo:

$$\Delta t = 3.14/100$$

A programação permite que tentemos construir v a partir das nossas definições anteriores de posição e tempo:

```
todos_dx = []
for i in range(len(pos)):
    if i+1 == len(pos):
        break
    else:
```

```
            i = i+1
            dx = pos[i] - pos[i-1]
            todos_dx.append(dx)

dt = 3.14/100
V = np.array(todos_dx)/dt
plt.plot(range(len(V)), V)
plt.xlabel('tempo')
plt.ylabel('velocidade')
plt.show()
```

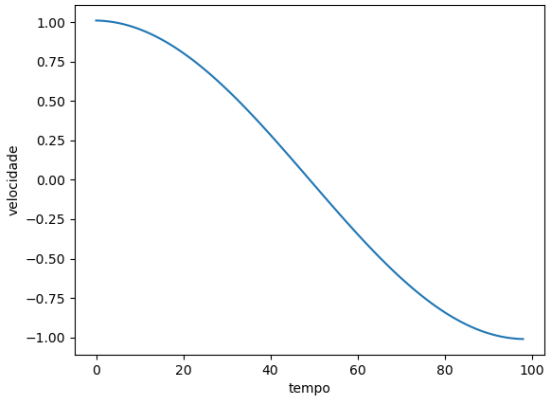


Figura 3.2: Gráfico da velocidade com relação ao tempo

Obtemos todas as velocidades ao construir cada velocidade instantêna. Como podemos perceber, nossa velocidade começa em um máximo, chega a zero, quando o objeto muda o sentido do movimento, e vai para um mínimo com mesmo valor absoluto que o máximo. O objeto se move até um certo ponto e volta próximo do ponto orígina. Sua velocidade muda ao longo do tempo, pois o sentido do movimento muda. Com o algoritmo citado, o problema é respondido, obtemos a forma da velocidade em função do tempo. É notório que a codificação de um sistema requer o entendimento de suas partes, e, portanto, codificando a velocidade partindo do espaço das posições e do tempo, compreende-se o que quer dizer por adquirir a velocidade a partir da posição.

Poderíamos investigar a aceleração, também, caso quiséssemos. Se esta a derivada da velocidade, então obtemos a aceleração para cada instante de tempo partindo das velocidades. Lembremos que:

$$a(t) = \frac{\Delta v}{\Delta t} = \frac{\Delta v(t) - \Delta v(t - \delta t)}{\Delta t}$$

Então:

```
todos_dv = []
for i in range(len(V)):
    if i+1 == len(V):
        break
    else:
        i = i+1
```

```
        dv = V[i] - V[i-1]
        todos_dv.append(dv)

dt = 3.14/100
A = np.array(todos_dv)/dt
plt.plot(range(len(A)), A)
plt.xlabel('tempo')
plt.ylabel('aceleracao')
plt.show()
```

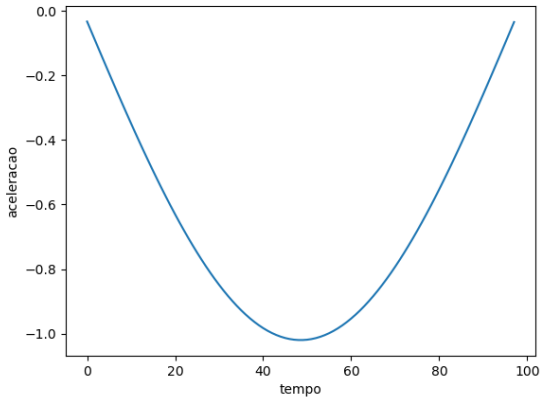


Figura 3.3: Gráfico da aceleração com relação ao tempo

É notório como python é útil nesses casos, pois muitas vezes para desenvolver o raciocínio da relação entre o gráfico velocidade e posição, gasta-se muito tempo e isso ocorre em detrimento da produtividade em sala.

Com esse exemplo, buscamos solidificar o potencial do Python para investigação científica em sala de aula. A multiplicidade de representações permite melhor assimilação, e portanto é muito vantajosa, ainda mais levando em consideração que podemos apresentar uma maior quantidade de recursos em um menor tempo.

3.2 Redes Neurais

Propomos um algoritmo que une assuntos muito relevantes tanto no aspecto científico quando no dia a dia das pessoas. Muitas vezes, um recurso mal utilizado é justamente aquele que não parece ser importante no presente (mesmo que a falta de importância seja um equívoco...). Partimos desse ímpeto por propor algo novo explorando redes neurais. Nossa proposta consiste em, utilizando conceitos aritméticos e algébricos, construir uma rede neural que resolva um problema conhecido. No nosso caso, optamos por resolver o dataset MNIST, que consiste em imagens de números de 0 a 9. O objetivo da rede neural é estipular quais são os valores dos dígitos de acordo com a imagem que recebe. Mais especificamente, em termos de código, a rede neural é um mecanismo que transforma um imagem, representada por uma matriz de 3 dimensões, em um digito, 0 ou 1, representado por uma matriz de uma dimensão.

Podemos utilizar numpy para compreender como um input se torna unidimensional por multiplicação de matrizes:

```
import numpy as np
input = np.random.rand(1, 100)
interagente = np.random.rand(100,1)
output_esperado = np.random.rand(1,1)
output = input@interagente
assert output.shape == output_esperado.shape
```

podemos também realizer uma operação de soma, levando em conta que as dimensões devem ser as mesmas:

```
import numpy as np
input = np.random.rand(1, 100)
interagente1 = np.random.rand(100,1)
interagente2 = np.random.rand(1,1)
output_esperado = np.random.rand(1,1)
output = input@interagente1 + interagente2
assert output.shape == output_esperado.shape
```

Uma Rede Neural executa esse tipo de operação. Recebe uma entrada e, através de operações de multiplicação matricial, gera uma saída que é então recebida por outro agente posterior. Podemos abstrair essa noção criando um objeto CamadaLinear.

```
import numpy as np

class CamadaLinear:
    def __init__(self, canais_entrada, canais_saida):
        self.pesos = np.random.randn(canais_entrada,
        canais_saida)
        self.propensoes = np.random.randn(1, canais_saida)
    def processar(self, x):
        y = x@self.pesos + self.propensoes
        return x, y
```

A rede linear é uma rede bidimensional que recebe entradas em um canal e retorna uma saída em outro canal. Canais são nada menos que as dimensões de saída e entrada de uma matriz em numpy. Podemos processar um input da seguinte maneira:

```
camada_linear = CamadaLinear(100, 1)
entrada = np.random.rand(16, 100)
entrada, saida = camada_linear.processar(entrada)
entrada.shape, saida.shape
# >>> (16, 100), (16, 1)
```

Podemos ver que uma camada linear recebe 16 amostras de dimensão 100 e retorna 16 amostras de dimensão 1. Isso permite que treinemos uma rede com múltiplas amostras. Podemos criar uma rede que contém diversas camadas, desde que levemos sempre em consideração que as dimensões deve ser continuadas. Criamos um processo de uma rede com múltiplas camadas da seguinte maneira:

```
class RedeNeural:
    def __init__(self, camadas):
        self.camadas = camadas
        self.agenda = {}
```

```
        for n, camada in enumerate(self.camadas):
            self.agenda[f'camada{n}'] = {
            'x': None, 'y':None}
    def processar(self, x):
        for n, camada in enumerate(self.camadas):
            x, y = camada.processar(x)
            self.agenda[f'camada{n}']['x'] = x
            self.agenda[f'camada{n}']['y'] = y
            x = y
        return y

rede = RedeNeural(
[CamadaLinear(100, 50),
CamadaLinear(50, 25),
CamadaLinear(25, 10),
CamadaLinear(10, 1)]
)
num_amostras = 16
entrada = np.random.rand(num_amostras, 100)
saida = rede.processar(entrada)
print(saida.shape)
# >>> (16,1)
```

A rede neural processa uma entrada passando-a por cada camada sequencialmente. A camada, ao processar sua entrada, retorna a entrada que recebeu e a sua saída. Aqui já definimos uma rede, mas ainda não é neural, pois não aprende com exposição a amostras. Para que esta faça isso, precisamos compreender como os pesos e propensões da rede afetam seus resultados. Para isso, devemos sempre ter um gabarito de cada pacote de amostras, e precisamos de uma função que estime o quão distante a resposta da rede está da resposta esperada. Aqui, nos manteremos no ambiente algébrico, e responderemos como atualizar a rede da forma apropriada.

3.3 Funções de erro

As funções de erro são uma das partes mais importantes no aprendizado de uma rede neural, e muitas vezes a troca de uma função erro para outra pode mudar radicalmente a performance do aprendizado da máquina. Introduziremos, inicialmente, uma função muito utilizada na resolução de problemas de regressão, a função de erro quadrático médio. Chamaremos nossa função de ”erro_medio”, por motivos de notação, definindo-a como:

```
def erro_medio(saida, resultado_esperado):
    N = resultado_esperado.shape[0]
    return (1/N)*(saida - resultado_esperado)**2
```

Essa função é simples, somente executa a operação da diferença entre as saídas da rede e os valores esperados e a eleva ao quadrado. Como veremos a seguir, o fato de elevarmos a diferença ao quadrado fará uma grande diferença no aprendizado.

3.4 Propagação Recursiva

Com uma função para estimar o erro, podemos estabelecer a relação entre os erros e os pesos e propensões de cada camada. Primeiro pensemos numa rede composta por uma única camada. Podemos notar que a função de erro é uma função quadrática, que é zero quando a saída e o valor esperado são iguais, e infinito caso a a saída e o valor esperado estejam muito distantes. Se pudéssemos criar um algoritmo que investiga, numericamente, a relação entre o erro da saída e do valor esperados, e os pesos e propensoes da camada que gerou a saída, deveriamos investigar

$$\frac{\Delta Erro}{\Delta pesos}$$

A pergunta a ser respondida é: quando meu erro muda se eu fizer uma pequena mudança nos pesos da camada? Quando mudamos o peso, mudamos por consequência a saída da rede, que por consequência é a entrada da função erro. Portanto, podemos também investigar:

$$\frac{\Delta Erro}{\Delta pesos} = \frac{\Delta Erro}{\Delta saida}\frac{\Delta saida}{\Delta pesos}$$

Poderíamos colocar um leve incremento nos pesos e verificar o quanto estes alteram as saídas, que por sua vez também alteram o resultado do erro.

Se temos uma função

$$f = f(x)$$

A variação dessa função com relação ao seu argumento é simplesmente:

$$\frac{\Delta y}{\Delta x}$$

Se a variação é muito alta, então uma ínfima alteração em x altera demasiadamente y. Se a variação é baixa, então uma considerável variação em x causa uma ínfima variação em y.

Em geometria, dois segmentos perpendiculares formam um triângulo retângulo. A relação entre a altura e a largura de um triângulo constitui um ângulo, ângulo este limitado, pois por definição, a soma dos ângulos de um triângulo constituem 180 graus. Se aplicarmos esse raciocínio na equação da variação de y com relação a x, podemos perceber que se esta razão é 0, então nenhum incremento em x altera y. Se a razão é 1, então y responde diretamente ao incremento de x sem nenhuma alteração (ou seja, a função f(x) retorna x). Se a razão é -1, então y responde diretamente ao incremento x mas de forma inversa (aumentar x em w diminui y em w, e vice-versa).

Lembrando que o Erro é uma função quadrática, investiguemos a relação:

$$f(x) = x^2$$

$$\frac{\Delta f(x)}{\Delta x} = \frac{\Delta x^2}{\Delta x}$$

Vamos supor que o incremento em x seja tão pequeno, que este constitui o próprio incremento mínimo possível, ou seja, qualquer incremento é derivado desse incremento mínimo, sendo todos os incrementos possíveis $1 * dx, 2 * dx, 3 * dx, ...$ Seja $\epsilon > 0$ e $\epsilon \ll 1$.

$$\frac{\Delta f(x)}{\Delta x} = \frac{f(x+\epsilon) - f(x)}{\epsilon} = \frac{(x+\epsilon)^2 - x^2}{\epsilon}$$

$$\frac{(x+\epsilon)^2 - x^2}{\epsilon} = \frac{x^2 + 2\epsilon x + \epsilon^2 - x^2}{\epsilon}$$

Se dividirmos o numerador e o denominador de uma fração pela mesma razão (desde que evitemos denominador nulo), então a razao permanece inalterada. Ao dividir, neste caso, numerador e denominador por ϵ, temos:

$$\frac{\frac{x^2+2\epsilon x+\epsilon^2-x^2}{\epsilon}}{\frac{\epsilon}{\epsilon}} = \frac{2x+\epsilon}{1}$$

Como ϵ é um incremento mínimo, então $2x >> \epsilon$ e

$$\frac{\frac{x^2+2\epsilon x+\epsilon^2-x^2}{\epsilon}}{\frac{\epsilon}{\epsilon}} = \frac{2x+\epsilon}{1} = 2x$$

Podemos perceber isso pois uma variação de $x = 1$ causa uma variação de $y = 2$ pois:
Erro(0,x=1) = 1 Erro(0,x=2) = 4
Quando x dobra, y quadriplica, seguindo a razão 2x.
Se o erro é uma função da saída da camada, então:

$$\frac{\Delta E}{\Delta y}\frac{\Delta y}{\Delta pesos}$$

$$\frac{\Delta E}{\Delta y} = 2y$$

$$\frac{\Delta y}{\Delta pesos} = ?$$

Suponha que todas as entradas das camadas fossem 1, então se dobrássemos os pesos, a camada retornaria o dobro do que retornaria se estivesse com os pesos inalterados. Ou seja, no caso onde

$$y = 1@pesos + propesoes$$

Dobrar o peso causa o dobro da saída, numa relação $\frac{y}{x} = 1$
Como a entrada é multiplicada pela saída, então a variação é direta e dependendo da entrada, sendo assim:

$$\frac{\Delta y}{\Delta pesos} = entradas$$

Definimos anteriormente:

$$\frac{\Delta Erro}{\Delta pesos} = \frac{\Delta Erro}{\Delta saida}\frac{\Delta saida}{\Delta pesos}$$

$$\frac{\Delta Erro}{\Delta saida}\frac{\Delta saida}{\Delta pesos} = \frac{\Delta E}{\Delta y}\frac{\Delta y}{\Delta pesos}$$

3.5 Funções de Ativação

Anteriormente, definimos as saídas das nossas camadas como $y = x@pesos + propensoes$. Existe algo nessa relação que não é muito favorável: nossas saídas são retas formadas pelas entradas, ou seja, as saídas não conseguem se desacoplar das entradas propriamente, e portanto são incapazes de prever algo 'geral'.

Uma função de ativação muito simples que pode ser utilizada aqui é a ativação ReLU [19], ilustrada na figura 3.4. Ela retorna y se a saída for > 0 e retorna 0 se a saída for < 0.

```
def relu(y):
    return np.where(y>0, y, 0)
```

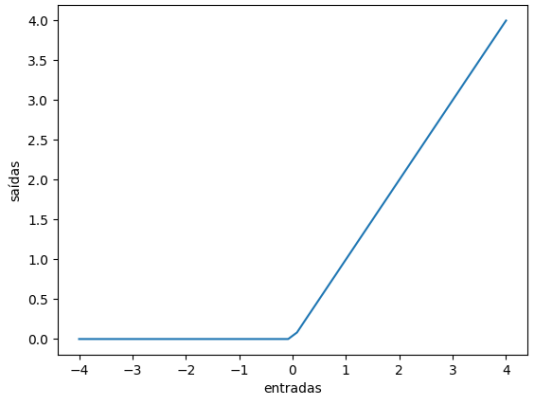


Figura 3.4: Ativação ReLU. Entradas no eixo horizontal; saídas no eixo vertical

Essa simples mudança tem grande impacto no cálculo dos erros, pois agora uma camada ativada retorna z = relu(y).

De volta ao código:

```
import numpy as np

class CamadaLinear:
    def __init__(self, canais_entrada, canais_saida, ativacao):
        self.pesos = np.random.randn(
        canais_entrada, canais_saida)
        self.propensoes = np.random.randn(1, canais_saida)
        self.ativacao = ativacao
    def processar(self, x):
        y = x@self.pesos + self.propensoes
        z = self.ativacao(y) if self.ativacao else y
        return x, y, z

class RedeNeural:
    def __init__(self, camadas):
        self.camadas = camadas
        self.agenda = {}
        for n, camada in enumerate(self.camadas):
            self.agenda[f'camada{n}'] = {
```

```
            'x': None, 'y':None, 'z':None
            }
    def processar(self, x):
        for n, camada in enumerate(self.camadas):
            x, y, z = camada.processar(x)
            self.agenda[f'camada{n}']['x'] = x
            self.agenda[f'camada{n}']['y'] = y
            self.agenda[f'camada{n}']['z'] = z
            x = z
        return z
```

Como a rede retorna duas saídas, uma ativada e outra desativada, a relação entre o erro e os pesos muda:

$$\frac{\Delta Erro}{\Delta pesos} = \frac{\Delta Erro}{\Delta z}\frac{\Delta z}{\Delta y}\frac{\Delta y}{\Delta pesos} \tag{3.1}$$

Perceba que o Erro é função direta de z; z é função direta de y; e y é função direta dos pesos.

Nesse caso é muito fácil investigar como a relu muda com relação a y. Se y < 0, então relu(y) = 0; se y>=0, então relu(y)=y. Sabemos que a função f(x)=x possui incremento $\frac{y}{x} = 1$. Portanto relu possui incremento 1 se y>=0 e incremento 0 se y<0.

Vamos introduzir o conceito incremento=True or False nas nossas funções de Erro e ativação:

```
def erro_medio(saida, val_esperado, incremento=False):
    N = val_esperado.shape[0]
    if incremento:
        return (2/N)*(saida - val_esperado)
    else:
        return (1/N)*(saida - val_esperado)**2

def relu(y, incremento=False):
    if incremento:
        return np.where(y>=0, 1, 0)
    else:
        return np.where(y>=0, y, 0)
```

Com isso, podemos especificar os incrementos dos pesos com relação ao Erro de acordo com nosso código:

$$\frac{\Delta Erro}{\Delta pesos} = (Erro(z, incremento = True) * relu(y, incremento = True))@x.T \tag{3.2}$$

Caso acoplemos mais de uma camada, devemos notar que o Erro com relação a camadas anteriores ainda pode ser obtido se associarmos incrementos e suas entradas diretas.

Por exemplo, o Erro com relação aos pesos de uma camada anterior a última saída depende das entradas e saídas da camada anterior, que possuem dimensão diferente da última camada. Podemos pensar no incremento dos pesos como resultando na entrada da última camada, e assim associar ao Erro, da seguinte maneira:

$$\frac{\Delta Erro}{\Delta pesos_{anterior}} = \frac{\Delta Erro}{\Delta z}\frac{\Delta z}{\Delta y}\frac{\Delta y}{\Delta z_{anterior}}\frac{\Delta z_{anterior}}{\Delta y_{anterior}}\frac{\Delta y_{anterior}}{\Delta pesos_{anterior}} \tag{3.3}$$

As saídas inativadas anteriores são funções diretas dos pesos anteriores; as saídas ativadas anteriores são funcões diretas das saídas inativadas anteriores; e as saídas da última camada são funções diretas das saídas ativadas da camada anterior. Essa expressão revela que cada camada anterior possui incrementos das camadas posteriores.

Em verdade, se tivermos múltiplas camadas, a mesma regra segue:

$$\frac{\Delta E}{\Delta pesos_j} = \frac{\Delta E}{\Delta z_n}\frac{\Delta z_n}{\Delta y_n}\frac{\Delta y_n}{\Delta z_{n-1}}\frac{\Delta z_{n-1}}{\Delta y_{n-1}}\frac{\Delta y_{n-1}}{\Delta z_{n-2}}\frac{\Delta z_{n-2}}{\Delta y_{n-2}}\frac{\Delta y_{n-2}}{\Delta z_{n-3}}...\frac{\Delta y_j}{\Delta pesos_j}$$

Os pesos da última camada recebem correções de incremento da função erro e de sua entrada. Os pesos de todas as outras camadas recebem correções de incremento das saídas das camadas anteriores e seus pesos.

Seja Z o incremento de todas as camadas posteriores à camada j. Então:

$$Z_j = ((Z_{j+1}@W_{j+1}.T) * ativacao_j(y_j, incremento = True)).T@X_j \tag{3.4}$$

Perceba que Z_{j+1} possui mesma dimensão que as saídas da camada $j + 1$; e W_{j+1} mesma dimensão que os pesos da camada $j + 1$. Como a saída de uma camada linear sempre possuirá dimensão $(num\_amostras, pesos.shape[1])$, os pesos de uma camada sempre possuirão dimensao $(pesos.shape[0], pesos.shape[1])$, fica evidente que Z@W.T satisfaz

$$(Z_{j+1}@W_{j+1}.T).shape = (num\_amostras, W_{j+1}.shape[1])@(W_{j+1}.shape[1], W_{j+1}.shape[0])$$

$$(Z_{j+1}@W_{j+1}.T).shape = (num\_amostras, W_{j+1}.shape[0])$$

O incremento da ativação j possui as dimensoes de saída de j:

$$(ativacao_j(y_j, incremento = True)).shape = (num\_amostras, W_j.shape[1])$$

Por definição:

$$Pesos_{j+1}.shape[0] = Pesos_j.shape[1] \tag{3.5}$$

Então

$$(Z_{j+1}@W_{j+1}.T).shape = ativacao_j(y_j, incremento = True).shape$$

E, portanto:

$$((Z_{j+1}@W_{j+1}.T) * ativacao_j(y_j, incremento = True)).T.shape = (W_j.shape[1], num\_amostras)$$

E, por fim, a entrada de j, que possui dimensão da saída de j-1. Assim:

$$X_j.shape = (num\_amostras, W_j.shape[0])$$

Então

$$Z_j.shape = ((W_j.shape[1], num\_amostras)@(num\_amostras, W_j.shape[0])).shape \tag{3.6}$$

$$Z_j.shape.T = (W_j.shape[1], W_j.shape[0]).T = (W_j.shape[0], W_j.shape[1]) \quad (3.7)$$

Ao buscar encontrar como diminuir o erro com relação aos pesos de cada camada, encontramos a expressão de incremento de peso da camada com relação ao erro da saída da rede. Já encontramos a expressão dos pesos, agora podemos usar a mesma lógica para as propensões. O incremento de propensão é mais simples de ser derivado, pois diferente do peso, a relação da propensão e a saída é direta, pois a propensão é uma mera soma sobre a saída.

Podemos atualizar nosso código para permitir que nossa rede corriga os pesos com relação aos erros, considerando os incrementos necessários para cada cálculo.

```
import numpy as np

def erro_medio(saida, val_esperado, incremento=False):
    N = val_esperado.shape[0]
    if incremento:
        return (2/N)*(saida - val_esperado)
    else:
        return (1/N)*(saida - val_esperado)**2

def relu(y, incremento=False):
    if incremento:
        return np.where(y>0, 1, 0)
    else:
        return np.where(y>0, y, 0)

class CamadaLinear:
    def __init__(self, canais_entrada, canais_saida, ativacao):
        self.pesos = np.random.randn(
        canais_entrada, canais_saida)
        self.propensoes = np.random.randn(1, canais_saida)
        self.ativacao = ativacao
    def processar(self, x):
        y = x@self.pesos + self.propensoes
        z = self.ativacao(y) if self.ativacao else y
        return x, y, z
    def corrigir(self, dPeso, dProp):
        self.pesos -= dPeso
        self.propensoes -= dProp


class RedeNeural:
    def __init__(self, camadas):
        self.camadas = camadas
        self.agenda = {}
        for n, camada in enumerate(self.camadas):
            self.agenda[f'camada{n}'] = {
            'x': None, 'y':None, 'z':None
            }
    def processar(self, x):
```

```
        for n, camada in enumerate(self.camadas):
            x, y, z = camada.processar(x)
            self.agenda[f'camada{n}']['x'] = x
            self.agenda[f'camada{n}']['y'] = y
            self.agenda[f'camada{n}']['z'] = z
            x = z
        return z
    def corrigir(self, erro_):
        N = len(self.camadas)
        for n in reversed(range(N)):
            if n+1==N:
                if self.camadas[n].ativacao:
                    Z_ = erro_*self.camadas[n].ativacao(
                    self.agenda[f'camada{n}']['y'],
                    incremento=True)
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T@self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso, dProp)
                else:
                    Z_ = erro_
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T @ self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso, dProp)
            else:
                if self.camadas[n].ativacao:
                    Z_ = Z_@(self.camadas[n+1].pesos).T
                    Z_ = Z_*self.camadas[n].ativacao(
                    self.agenda[f'camada{n}']['y'],
                    incremento=True
                    )
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T@self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso, dProp)
                else:
                    Z_ = Z_@(self.camadas[n+1].pesos).T
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T@self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso,dProp)
```

Definimos o funcionamento de uma rede. A primeira camada gera, de saída, a entrada da próxima camada. Após todas as camadas operarem, a saída da última camada é comparada com

os valores esperados. Em estudos de redes neurais, é uma abordagem comum testar se uma rede consegue alcançar precisão ínfima para uma mesma entrada. Esse problema chama-se ”overfitting”, e descreve uma rede memorizando uma entrada. Se nossa rede é capaz de memorizar suas entradas, então é porque ela é capaz de diminuir seu erro, caracterizando assim uma rede capaz de aprender. Nomearemos esse arquivo python de ”RedeNeural.py”, e salvaremos este arquivo no fichário de nosso projeto.

Nossas classes e funções estão definidas em um arquivo python, e utilizaremos daqui em diante esse arquivo para importar a rede definida, utilizando-a como biblioteca para outros projetos. Contanto que o arquivo que descreve a biblioteca esteja no mesmo fichário que os códigos que o importarão, podemos sempre fazer:

```
# importa classes e funcoes individualmente
from RedeNeural import CamadaLinear, RedeNeural, relu, erro_medio
# import todas as classes e funcoes
from RedeNeural import *
```

O símbolo * descreve a ação de importar todas as outras variáveis, funções e classes que estejam presentes no arquivo importado.

Para testar o funcionamento da rede, resolve-se o problema de ”overfitting”. Fazemos-o importando as classes e funções da rede neural e as utilizando na tarefa de memorizar uma entrada fixa:

```
import numpy as np
from RedeNeural import CamadaLinear, RedeNeural, relu, erro_medio


C = CamadaLinear
camadas = [
C(100,15, ativacao=relu),
C(15,10, ativacao=relu),
C(10,5,ativacao=relu),
C(5,1,ativacao=None),
]
rede = RedeNeural(camadas)
num_amostras=16
entrada = np.random.rand(num_amostras, 100)
n = num_amostras
r_esperado = np.array(
[np.random.rand(1) for _ in range(n)]
).reshape(-1,1)

for t in range(3):
    saida = rede.processar(entrada)
    erro = erro_medio(saida, r_esperado)
    erro_ = erro_medio(saida,r_esperado,incremento=True)
    rede.corrigir(erro_)
    print(f'erro com {t} correcoes: {erro.mean()}')
```

Ao executar o código, observa-se um problema que precisa ser corrigido:

- Os erros são grandes demais, causando correções que prejudicam a estabilidade do treino

Este erro ocorre pelos seguintes motivos:

- Os pesos, quando criados de forma randomizada, às vezes são grandes demais e resultam em saídas muito distantes dos resultados esperados;
- A função erro retorna erros grandes demais quando os resultados estão distantes, causando correções dos pesos desproporcionais;

Introduzindo uma variável que diminua os incrementos de correção em cada camada é um bom começo. Chamando de ”ta” a variável que regula a taxa de aprendizado, introduzimos-a no arquivo RedeNeural dentro das classes RedeNeural e CamadaLinear da seguinte maneira:

```
class CamadaLinear:
    def __init__(self, canais_entrada, canais_saida, ativacao):
        self.pesos = np.random.randn(
        canais_entrada, canais_saida)
        self.propensoes = np.random.randn(1, canais_saida)
        self.ativacao = ativacao
    def processar(self, x):
        y = x@self.pesos + self.propensoes
        z = self.ativacao(y) if self.ativacao else y
        return x, y, z
    def corrigir(self, dPeso, dProp, ta):
        self.pesos -= ta*dPeso
        self.propensoes -= ta*dProp

class RedeNeural:
    def __init__(self, camadas, ta):
        self.ta = ta
        self.camadas = camadas
        self.agenda = {}
        for n, camada in enumerate(self.camadas):
            self.agenda[f'camada{n}'] = {
            'x': None, 'y':None, 'z':None
            }
    def processar(self, x):
        for n, camada in enumerate(self.camadas):
            x, y, z = camada.processar(x)
            self.agenda[f'camada{n}']['x'] = x
            self.agenda[f'camada{n}']['y'] = y
            self.agenda[f'camada{n}']['z'] = z
            x = z
        return z
    def corrigir(self, erro_):
        N = len(self.camadas)
        for n in reversed(range(N)):
            if n+1==N:
                if self.camadas[n].ativacao:
                    Z_ = erro_*self.camadas[n].ativacao(
                    self.agenda[f'camada{n}']['y'],
```

```
                    incremento=True)
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T@self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso, dProp,
                    self.ta)
                else:
                    Z_ = erro_
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T @ self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso, dProp,
                    self.ta)
            else:
                if self.camadas[n].ativacao:
                    Z_ = Z_@(self.camadas[n+1].pesos).T
                    Z_ = Z_*self.camadas[n].ativacao(
                    self.agenda[f'camada{n}']['y'],
                    incremento=True
                    )
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T@self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso, dProp,
                    self.ta)
                else:
                    Z_ = Z_@(self.camadas[n+1].pesos).T
                    dProp = Z_.mean(axis=0, keepdims=True)
                    dPeso = (
                    Z_.T@self.agenda[f'camada{n}']['x']
                    ).T
                    self.camadas[n].corrigir(dPeso,dProp,
                    self.ta)
```

Isso garante estabilidade ao treino, e o erro diminui para uma mesma entrada, como esperado. De fato, este deve ficar próximo de zero para uma alta exposição à mesma entrada:

```
import numpy as np
import matplotlib.pyplot as plt
from RedeNeural import CamadaLinear,RedeNeural,relu,erro_medio

C = CamadaLinear
camadas = [
C(100,15, ativacao=relu),
C(15,10, ativacao=relu),
C(10,5,ativacao=relu),
C(5,1,ativacao=None),
```

```
]
rede = RedeNeural(camadas, ta=0.0001)
num_amostras=16
entrada = np.random.rand(num_amostras, 100)
n = num_amostras
r_esperado = np.array(
[np.random.rand(1) for _ in range(n)]
).reshape(-1,1)
epocas=19
erros_lista=[]
for t in range(epocas):
    saida = rede.processar(entrada)
    erro = erro_medio(saida, r_esperado)
    erros_lista.append(erro.mean())
    erro_ = erro_medio(saida,r_esperado,incremento=True)
    rede.corrigir(erro_)
    print("epoca ", t+1, "erro ", erro.mean())

plt.plot(range(len(erros_lista)), erros_lista)
plt.show()
```

O treino está estável, e o erro diminui consideravelmente, caracterizando um ”overfitting”. Sem corrigir a inicialização dos pesos, já obtivemos um bom aprendizado. Atentando-se ao fato de que erros grandes são consequência de saídas grandes, podemos regular os pesos com simples funções, oferecendo-as na inicialização da classe de cada camada. Editando novamente o código RedeNeural.py:

```
def regulador(matriz, val=0.9):
    return matriz * val

class CamadaLinear:
    def __init__(self,
    canais_entrada,
    canais_saida,
    ativacao,
    regulador=None):
        self.pesos = np.random.randn(
        canais_entrada, canais_saida)
        self.propensoes = np.random.randn(1, canais_saida)
        if regulador:
            self.pesos = regulador(self.pesos)
            self.propensoes = regulador(self.propensoes)
        self.ativacao = ativacao
    def processar(self, x):
        y = x@self.pesos + self.propensoes
        z = self.ativacao(y) if self.ativacao else y
        return x, y, z
    def corrigir(self, dPeso, dProp, ta):
        self.pesos -= ta*dPeso
        self.propensoes -= ta*dProp
```

De fato, pode-se constatar a diferença de uma inicialização de pesos com e sem regularizador:

```
from RedeNeural import *

C = CamadaLinear
nao_regulado = C(100,100,
ativacao=None,regulador=None).pesos
regulado = C(100,100,
ativacao=None, regulador=regulador).pesos
print(f'COM:{regulado.max()}|SEM:␣{nao_regulado.max()}')
```

O teste do overfitting sugere um problema da rede neural, o erro pode diminuir pelos motivos errados! Se a rede aprende e abstrai as características necessárias e se a rede apena memoriza as amostras, o erro diminui da mesma maneira! A estratégia para garantir que a rede não memoriza os dados é sempre dividir as amostras em duas parte, uma parte para o treino e outra para o teste. Durante o treino, a rede recebe as amostras de treino; e durante o teste, a rede avalia saídas para entradas que nunca viu antes. Se o erro da rede permanece mínimo nas amostras de teste, então ela aprendeu!

3.6 Problemas Binários

Como vimos em seções anteriores, a função erro é um dos elementos mais importantes de uma rede neural, pois dita a forma como seus pesos e propensões serão corrigidos. A priori, utilizamos uma função de erro médio por ser uma função muito simples de definir. Essa função, porém, não é a mais adequada para resolver problemas de respostas binárias: zero ou um.

Uma rede neural é nada menos do que múltiplas matrizes acopladas que processam uma entrada e geram uma saída, matrizes essas que se auto corrigem ao longo do tempo pela técnica de propagação recursiva. No caso de problemas binários, faz sentido que utilizemos uma função de ativação que restrinja (sem ser detrimental ao processo de propagação) as saídas para dentro do intervalo $[0, 1]$. Uma função de ativação muito conhecida e que possui esse propósito é a função sigmoide [20], ilustrada na figura 3.5:

```
def sigmoid(z, incremento=False):
    s = 1 / (1+np.exp(-z))
    if incremento:
        return s*(1-s)
    else:
        return s
```

Com essa ativação podemos garantir que a rede 'foque' no intervalo $[0, 1]$ para suas saídas, permitindo um treino mais estável (consultar apêndice A para mais informações).

No caso da função erro, vamos utilizar uma chamada "entropia binária cruzada".

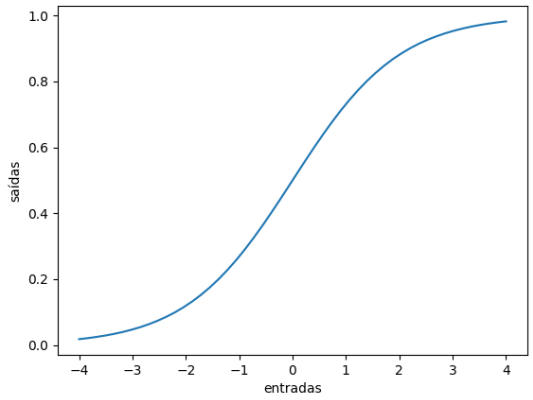


Figura 3.5: Ativação Sigmoide. Entradas no eixo horizontal; saídas no eixo vertical

```
def entropia_binaria_cruzada(saida, val_esperado, incremento=False):
    out = saida
    gt = val_esperado
    epsilon = 1e-8
    out = np.clip(out, epsilon, 1 - epsilon)
    if incremento:
        return (out - gt) / (len(gt) * out * (1 - out))
    else:
        return -np.mean(gt * np.log(out)
        + (1 - gt) * np.log(1 - out))
```

Estabelecendo essa função de erro para as correções e uma ativação sigmoide na última camada, indicamos para a rede de forma explícita que estamos tratando de um problema binário. Vamos testar o desempenho das funções de ativação e erro mencionada para reconhecimento de números. Importamos o banco de dados MNIST [21], que possui digitos de 0 a 9 (portanto, 10 categorias), utilizando a bibliotenca tensorflow, e selecionamos apenas os digitos 0 e 1 (ou qualquer outro par de digitos), transformando o problema em classificação binária; e treinamos a rede neural:

```
import numpy as np
from RedeNeural import CamadaLinear, RedeNeural, relu
from tensorflow.keras.datasets import mnist

def entropia_binaria_cruzada(
    saida, val_esperado, incremento=False
    ):
    out = saida
    gt = val_esperado
    epsilon = 1e-8
    out = np.clip(out, epsilon, 1 - epsilon)
    if incremento:
        return (out - gt) / (len(gt) * out * (1 - out))
    else:
        return -np.mean(gt * np.log(out)
```

```
        + (1 - gt) * np.log(1 - out))

def sigmoid(z, incremento=False):
    s = 1 / (1 + np.exp(-z))
    if incremento:
        return s * (1 - s)
    else:
        return s


(x_treino, y_treino), (x_teste, y_teste) = mnist.load_data()
treino_filtro = (y_treino==0) | (y_treino==1)
teste_filtro = (y_teste==0) | (y_teste==1)
x_treino = x_treino[treino_filtro]
y_treino = y_treino[treino_filtro]
x_teste = x_teste[teste_filtro]
y_teste =  y_teste[teste_filtro]
x_treino = x_treino.reshape(-1, 28 * 28) / 255.0
x_teste = x_teste.reshape(-1, 28 * 28) / 255.0
y_treino = y_treino.astype(float).reshape(-1, 1)
y_teste = y_teste.astype(float).reshape(-1, 1)
num_amostras = 64
N = num_amostras
epocas = 10
camadas = [
    CamadaLinear(28 * 28, 128, ativacao=relu),
    CamadaLinear(128, 64, ativacao=relu),
    CamadaLinear(64, 1, ativacao=sigmoid)
]
rede = RedeNeural(camadas, ta=0.0001)

# Treino
for epoc in range(epocas):
    indices = np.random.permutation(len(x_treino))
    x_treino, y_treino = x_treino[indices], y_treino[indices]
    for i in range(0, len(x_treino), num_amostras):
        x_batch = x_treino[i:i + num_amostras]
        y_batch = y_treino[i:i + num_amostras]
        saida = rede.processar(x_batch)
        erro = entropia_binaria_cruzada(
        saida,
        y_batch)
        erro_ = entropia_binaria_cruzada(
        saida,
        y_batch, incremento=True)
        rede.corrigir(erro_)
    print(f"Epoca {epoc + 1} Erro: {erro}")
```

```
# Avalie Treino
teste_saida = rede.processar(x_teste)
teste_erro = entropia_binaria_cruzada(teste_saida, y_teste)
teste_acuracia = np.mean((teste_saida > 0.5) == y_teste)
print(f"Erro Teste: {teste_erro},
Teste Acuracia {teste_acuracia}")
```

Atingimos acurácia de 99% nos testes!

Esse modelo performa muito bem para essas tarefas, e, como podemos ver, a troca de função de erro muda completamente a eficácia do aprendizado. Em verdade, estaríamos mentindo se não disséssemos que o erro quadrático médio performa tão bem quanto o erro de entropia binária cruzada. Contanto que estejamos interessados em resolver um problema de regressão (preveja qualquer valer em um determinado intervalo arbitrário, desde que performe bem com o banco de dados).

3.7 Problema Físico

Como conclusão, propomos o uso da rede neural codificada para resolver problemas de Física. Como explicitado anteriormente, a rede neural é um mecanismo que diminui o erro comparando suas respostas com valores esperados. Em Física existem problemas assim: quando sabemos um valor esperado e queremos encontrar como extrair informação do sistema que conclua aquele valor esperado. A temperatura, por exemplo, já era medida muito antes de se saber seu significado físico! Ao utilizar a rede neural criada para resolver problemas de regressão em Física, extendemos o problema de forma aprofundada: simulamos uma rede neural confirmando sua eficácia através de análises numéricas, e, depois, com a natureza científica dessa simulação confirmada, fazemos perguntas à simulação: ”Qual a temperatura desta imagem?”.

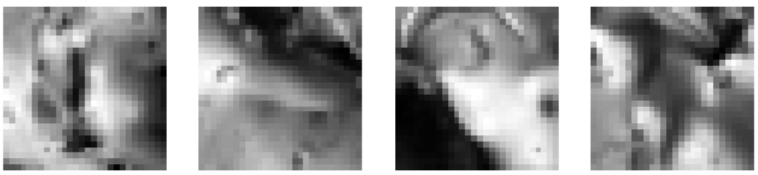


Figura 3.6: Amostras para o problema de regressão

Como já mencionado, a função de erro quadrático médio performa muito bem em problemas de regressão. Utilizando um banco de dados de imagens de materiais e suas temperaturas, buscaremos resolver o problema de regressão: estime a temperatura do material baseada em sua temperatura.

Com a biblioteca RedeNeural já construída, basta escrever o algoritmo de treino:

```
from RedeNeural import CamadaLinear,RedeNeural,relu,erro_medio
from PIL import Image # Biblioteca para carregar imagens
import os # Biblioteca para navegar em pastas
import numpy as np
from random import shuffle #embaralhar dados


def imagem(caminho):
        try:
                img = Image.open(caminho)
                img.verify()
                return True
        except (IOError, SyntaxError):
                return False

def dividir_em_12(img, dim=(28,28)):
        largura = 510
        altura = 511
        partes = []
        for linha in range(3):
                for coluna in range(4):
                        esquerda = coluna*largura
                        topo = linha*altura
                        direita = esquerda + largura
                        baixo = topo + altura
                        pedaco = img.crop(
                        (esquerda, topo, direita, baixo)
                        ).resize(dim)
                        partes.append(pedaco)
        return partes

def funcao_temperatura(fotoNum):
        if fotoNum < 62:
                return (fotoNum-5)*0.3 + 40
        else:
                return (fotoNum-62)*0.05 + 57

def gerar_amostras(caminho, modo='L'):
        amostras = []
        for root, dirs, files in os.walk(caminho):
                for file in files:
                        caminho_img = f'{root}\\{file}'
                        if '.' in file and imagem(caminho_img):
```

```
                                fotoNum = int(
                    file.split('.')[0])
                                temp = funcao_temperatura(
                    fotoNum)
                                for img in dividir_em_12(
                                        Image.open(caminho_img)
                                        ):
                                        amostras.append(
                                        [np.array(
                                        img.convert(modo)
                                        )/255.0,
                                        temp])
            return amostras


amostras = gerar_amostras('data')
shuffle(amostras)

div=0.7
num_amostras = 64
N = num_amostras
amostras = amostras[:int(div*len(amostras))]
teste = amostras[int(div*len(amostras)):]
amostras = amostras[:(len(amostras)//N)*N]
print("amostras ", len(amostras))

ta = 0.0001
N = num_amostras
epocas = 19
camadas = [
    CamadaLinear(28 * 28, 128, ativacao=relu),
    CamadaLinear(128, 64, ativacao=relu),
    CamadaLinear(64, 1, ativacao=None)
]
rede = RedeNeural(camadas, ta=ta)

# Treino
for epoc in range(epocas):
    erro_epoca = []
    shuffle(amostras)
    for i in range(0, len(amostras), N):
        x = np.array([z[0] for z in amostras[i:i+N]])
        y = np.array([z[1] for z in amostras[i:i+N]])
        x = x.reshape(N, 28*28)
        y = y.reshape(N, 1)
        saida = rede.processar(x)
        erro = erro_medio(saida, y)
        erro_ = erro_medio(saida, y, incremento=True)
```

```
            rede.corrigir(erro_)
            erro_epoca.append(erro)
        print(f"Epoca {epoc + 1} Erro: {np.mean(erro_epoca)}")

#Teste
x = np.array([z[0] for z in teste])
y = np.array([z[1] for z in teste])
x = x.reshape(-1, 28*28)
y = y.reshape(-1, 1)
saida = rede.processar(x)
erro = erro_medio(saida, y)
print(f"Erro no teste: {erro.mean()}")
```

Os erros plotados são instáveis e sugerem o mesmo problema do caso binário, os pesos iniciais geram saídas grandes que ocasionam erros muito grandes. Fazendo uma simples mudança na inicialização das camadas lineares resolve este problema. De volta no arquivo RedeNeural.py:

```
def regulador(matriz):
    canais_entrada = matriz.shape[0]
    return matriz*np.sqrt(2/canais_entrada)

class CamadaLinear:
    def __init__(self,
    canais_entrada,
    canais_saida,
    ativacao,
    regulador=regulador):
        self.pesos = np.random.randn(
        canais_entrada, canais_saida)
        self.propensoes = np.zeros((1, canais_saida))
        if regulador:
            self.pesos = regulador(self.pesos)
            self.propensoes = regulador(self.propensoes)
        self.ativacao = ativacao
    def processar(self, x):
        y = x@self.pesos + self.propensoes
        z = self.ativacao(y) if self.ativacao else y
        return x, y, z
    def corrigir(self, dPeso, dProp, ta):
        self.pesos -= ta*dPeso
        self.propensoes -= ta*dProp
```

O regulador adicionado é conhecido como ”inicialização He”, e a troca de numpy.random.rand para numpy.zeros nas propensões ocasiona saídas iniciais menores, tornando o treino mais estável.

O erro observado não é animador, embora diminua com o tempo, permanece maior que 1. De fato, os valores adivinhados pela rede não estão tão distantes das temperaturas reais, pois, supondo uma temperatura de 50 graus, e um erro quadrático médio igual a 4, segue que:

$$(50 - x)^2 = 4$$

$$|50 - x| = 2$$

o valor adivinhado pela rede, x, é 2 graus a menos ou a mais que o valor verdadeiro!

Inserindo um cálculo de acurácia na fase de teste:

```
#Teste
x = np.array([z[0] for z in teste])
y = np.array([z[1] for z in teste])
x = x.reshape(-1, 28*28)
y = y.reshape(-1, 1)
saida = rede.processar(x)
erro = erro_medio(saida, y)
acc = sum(
        abs(saida-y) < 0.5
        ) / y.shape[0]
print(f"Erro no teste: {erro.mean()}")
print(f"Acuracia no teste: {acc}")
```

É curioso comparar erro e acurácia no teste. ambos destoam muito! O erro no teste é ínfimo, caracterizando que a rede eneraliza bem, mas a acurácia é muito baixa, ou seja, individualmente, cada resposta sempre está errada.

Durante o treinamento, a rede corrige seus pesos comparando pacotes de amostras entre si. Muitas vezes, a rede pode conseguir diminuir o erro aprendendo a sempre ”chutar” a média das amostras. Isso não é bom, pois o modelo generaliza com baixa acurácia.

Se o banco de dados não é extenso, ao usar um alto número de amostras, a variância de cada amostra tende a ser pequena e a rede generaliza mal. Alternativamente, se o número de amostras é pequeno, a variância de cada amostra é alta e a rede tende a generalizar melhor. Um número baixo de amostras nunca é recomendado, pois alta variância dificulta generalização. Deve-se encontrar um número meio-termo, com alta variância e número de amostras alto, mas não tão alto que generalize mal.

Aumentando a quantidade de amostras para treino, separando apenas 8 imagens para teste. Com um número de amostras igual 8, garantimos que a variância seja alta mantendo uma quantidade considerável de amostras por pacote de correção. De fato, a rede generaliza bem e, embora erre alguns resultados, chuta os valores de forma coesa! Em verdade, alcançamos um Erro < 0.5 durante o treino para 200 épocas. Editando o código do treino novamente:

```
from RedeNeural import CamadaLinear, RedeNeural, relu, erro_medio
from PIL import Image # Biblioteca para carregar imagens
import os # Biblioteca para navegar em pastas
import numpy as np
import matplotlib.pyplot as plt
from random import shuffle #embaralhar dados


def imagem(caminho):
        try:
                img = Image.open(caminho)
```

```
                img.verify()
                return True
        except (IOError, SyntaxError):
                return False

def dividir_em_12(img, dim=(28,28)):
        largura = 510
        altura = 511
        partes = []
        for linha in range(3):
                for coluna in range(4):
                        esquerda = coluna*largura
                        topo = linha*altura
                        direita = esquerda + largura
                        baixo = topo + altura
                        pedaco = img.crop(
                        (esquerda, topo, direita, baixo)
                        ).resize(dim)
                        partes.append(pedaco)
        return partes

def funcao_temperatura(fotoNum):
        if fotoNum < 62:
                return (fotoNum-5)*0.3 + 40
        else:
                return (fotoNum-62)*0.05 + 57

def gerar_amostras(caminho, modo='L'):
        amostras = []
        for root, dirs, files in os.walk(caminho):
                for file in files:
                        caminho_img = f'{root}\\{file}'
                        if '.' in file and imagem(caminho_img):
                                fotoNum = int(
                file.split('.')[0])
                                temp = funcao_temperatura(
                fotoNum)
                                for img in dividir_em_12(
                                        Image.open(caminho_img)
                                        ):
                                        amostras.append(
                                        [np.array(
                                        img.convert(modo)
                                        )/255.0,
                                        temp])
        return amostras
```

```
amostras = gerar_amostras('data')
div=0.9975
num_amostras = 8
N = num_amostras

amostras = amostras[:int(div*len(amostras))]
shuffle(amostras)

teste = amostras[int(div*len(amostras)):]
amostras = amostras[:(len(amostras)//N)*N]
print("amostras ", len(amostras))
print("teste ", len(teste))

ta = 0.00001
N = num_amostras
epocas = 200
camadas = [
    CamadaLinear(28 * 28, 128, ativacao=relu),
    CamadaLinear(128, 64, ativacao=relu),
    CamadaLinear(64, 1, ativacao=None)
]
rede = RedeNeural(camadas, ta=ta)

# Treino
for epoc in range(epocas):
    erro_epoca = []
    shuffle(amostras)
    for i in range(0, len(amostras), N):
        x = np.array([z[0] for z in amostras[i:i+N]])
        y = np.array([z[1] for z in amostras[i:i+N]])
        x = x.reshape(N, 28*28)
        y = y.reshape(N, 1)
        saida = rede.processar(x)
        erro = erro_medio(saida, y)
        erro_ = erro_medio(saida, y, incremento=True)
        rede.corrigir(erro_)
        erro_epoca.append(erro)
    print(f"Epoca {epoc + 1} Erro: {np.mean(erro_epoca)}")

#Teste
x = np.array([z[0] for z in teste])
y = np.array([z[1] for z in teste])
x = x.reshape(-1, 28*28)
y = y.reshape(-1, 1)
saida = rede.processar(x)
erro = erro_medio(saida, y)
acc = sum(
        abs(saida-y) < 0.5
```

```
        ) / y.shape[0]
print(f"Erro no teste: {erro.mean()}")
print(f"Acuracia no teste: {acc}")


# Visualizar Resultados
fig, axs = plt.subplots(2, 4)
n=0
x = x.reshape(-1,28,28,1)
for linha in range(2):
        for coluna in range(4):
                axs[linha][coluna].imshow(x[n], cmap='gray')
                axs[linha][coluna].axis('off')
                axs[linha][coluna].set_title(
                        f'{y[n].round(1)}|{saida[n].round(1)}'
                        )
                n+=1
plt.show()
```

A rede é capaz de, com poucas amostras, generalizar. E mesmo quando erra, chuta valores muito próximos dos valores reais! Ainda sim, a acurácia de teste é baixa, como pode-se observar na figura 3.7.

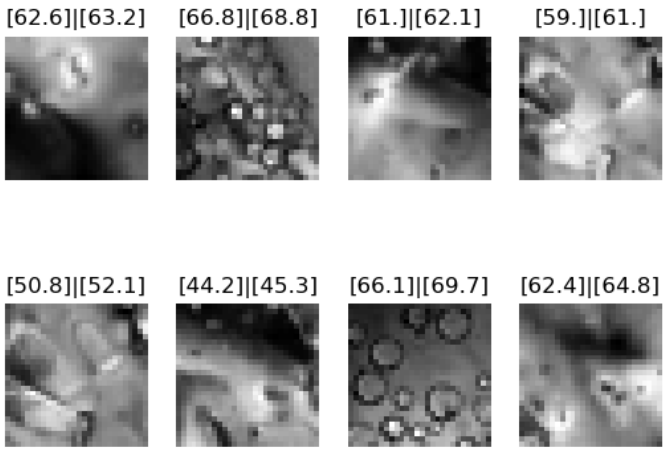


Figura 3.7: Resultado da rede com as amostras de teste. À esquerda, o valor esperado; à direita, o valor previsto pela rede

Como pode-se constatar, na imagem 3.8, a rede prevê muito mal temperaturas que estão próximas do mínimo ou do máximo do banco de dados:

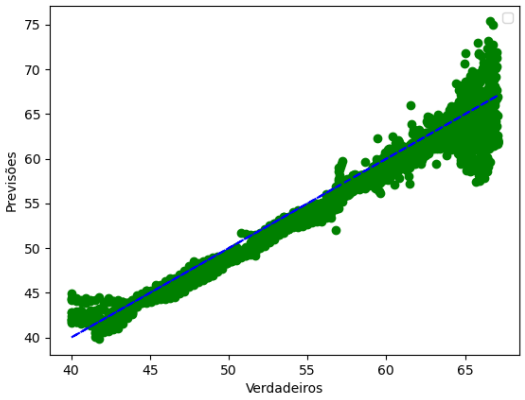


Figura 3.8: Previsões da rede com relação aos valores esperados

Devido ao fato do modelo generalizar bem sem memorizar o banco de dados, pode-se treiná-lo por muitas épocas sem causar saturação no treino. Treinando-o por mil épocas, observa-se acurácia considerável:

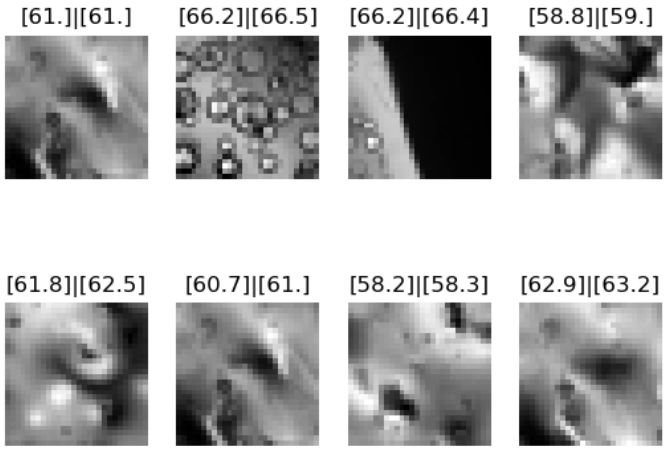


Figura 3.9: Resultados do treino de mil épocas. À esquerda, os valores esperados; à direita, os valores previstos pela rede

A rede prevê com maior certeza temperaturas que estão nas bordas, de acordo com o gráfico da figura 3.10:

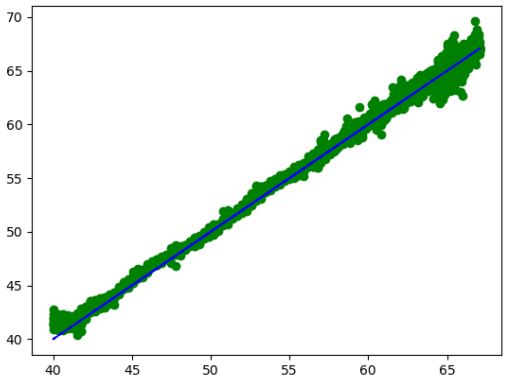


Figura 3.10: Previsões da rede com relação aos valores esperados. No eixo horizontal, os valores verdadeiros; no eixo vertical, os valores previstos

Resolvemos o problema Físico. O modelo generaliza bem e é capaz de prever com precisão temperaturas de amostras.

Todo o desenvolvimento da proposta seguiu a simples ideia de explorar a programação como um meio de investigação e compreensão do processo científico. Sob o óculo da semiótica social, exploramos uma proposta avançada de Ensino, explorando os múltiplos potenciais da programação para o Ensino de Física. Constatamos que a escolha do python é, certamente, perspicaz sobre esta ótica, pois a simplicidade da sintaxe e a facilidade em se usar bibliotecas não nativas garante utilidade muito abrangente. Literalmente, qualquer processo de investigação Física, a nível do Ensino Básico, pode ser proposto e investigado em Python, assegurando o potencial desta linguagem.

Esse estudo buscou ser o mais atualizado possível, conjecturando propostas que, no presente, são temas muito pertinentes. Todos os códigos propostos podem ser encontrados no github.

Conclusão

O presente trabalho propôs a integração da linguagem Python no ensino de Física como uma ferramenta poderosa para promover múltiplas representações, com base nos fundamentos teóricos da semiótica social. A pesquisa demonstrou que o uso de recursos variados, como visualizações gráficas, simulações computacionais e modelagem matemática, favorece a compreensão dos conceitos físicos, alinhando-se à ideia de que a diversificação dos meios de representação potencializa a assimilação do conhecimento.

O Python, por sua facilidade de aprendizado, ampla aplicabilidade e ferramentas robustas como numpy e matplotlib, mostrou-se um instrumento acessível e eficiente para fomentar o ensino baseado em múltiplas representações. Por meio da implementação de exemplos práticos, como a construção de uma rede neural do zero e sua aplicação na resolução de um problema de regressão — a determinação da temperatura de um material a partir de sua imagem —, foi possível evidenciar como essa abordagem pode não apenas reforçar a compreensão conceitual, mas também introduzir os estudantes a tecnologias modernas e relevantes.

Ao longo deste estudo, argumentou-se que a inclusão de ferramentas computacionais no ensino de Física transcende a simples transmissão de conteúdo, permitindo que os estudantes se envolvam de maneira mais ativa e exploratória com os problemas apresentados. Essa prática não apenas estimula o desenvolvimento do pensamento crítico e computacional, mas também aproxima o ensino de Física das demandas contemporâneas do mercado e da ciência.

Por fim, a proposta apresentada ressalta o potencial de uma abordagem pedagógica que integra a programação à semiótica social, contribuindo para tornar o aprendizado mais dinâmico, interativo e significativo. Este trabalho, portanto, serve como ponto de partida para futuros estudos que visem ampliar e diversificar as aplicações dessa metodologia no ensino de Ciências, evidenciando a importância de alinhar práticas educacionais às tecnologias emergentes e aos fundamentos teóricos que sustentam o aprendizado efetivo.

Apêndice

Código

Versão completa do código da rede neural produzida neste trabalho:

```
import numpy as np

def erro_medio(saida, val_esperado, incremento=False):
    N = val_esperado.shape[0]
    if incremento:
        return (2/N)*(saida - val_esperado)
    else:
        return (1/N)*(saida - val_esperado)**2

def relu(y, incremento=False):
    if incremento:
        return np.where(y>0, 1, 0)
    else:
        return np.where(y>0, y, 0)

def regulador(matriz):
    canais_entrada = matriz.shape[0]
    return matriz * np.sqrt(2 / canais_entrada)

class CamadaLinear:
    def __init__(
    self, canais_entrada, canais_saida,
    ativacao, regulador=regulador):
        self.pesos = np.random.randn(
        canais_entrada, canais_saida)
        self.propensoes = np.zeros((1, canais_saida))
        if regulador:
            self.pesos = regulador(self.pesos)
            self.propensoes = regulador(self.propensoes)
        self.ativacao = ativacao

    def processar(self, x):
        y = x @ self.pesos + self.propensoes
        z = self.ativacao(y) if self.ativacao else y
        return x, y, z
```

```
    def corrigir(self, dPeso, dProp, ta):
        self.pesos -= ta * dPeso
        self.propensoes -= ta * dProp

class RedeNeural:
    def __init__(self, camadas, ta):
        self.ta = ta
        self.camadas = camadas
        self.agenda = {
        f'camada{n}': {
        'x': None, 'y': None, 'z': None,
        } for n in range(len(self.camadas))}

    def processar(self, x):
        for n, camada in enumerate(self.camadas):
            x, y, z = camada.processar(x)
            self.agenda[f'camada{n}']['x'] = x
            self.agenda[f'camada{n}']['y'] = y
            self.agenda[f'camada{n}']['z'] = z
            x = z
        return z

    def corrigir(self, erro_):
        Z_ = erro_
        N = len(self.camadas)

        for n in reversed(range(N)):
            if n + 1 < N:
                Z_ = Z_ @ self.camadas[n + 1].pesos.T

            if self.camadas[n].ativacao:
                Z_ *= self.camadas[n].ativacao(
                    self.agenda[f'camada{n}']['y'],
                    incremento=True)

            dProp = Z_.mean(axis=0, keepdims=True)
            dPeso = (Z_.T @ self.agenda[f'camada{n}']['x']).T
            self.camadas[n].corrigir(dPeso, dProp, self.ta)
```

Ativação sigmoide

O comportamento da função sigmoide, A, é definida pela expressão, com x sendo a entrada:

$$A(x) = \frac{1}{1 + e^{-x}}$$

Investigando os extremos dessa função A, temos:

$$A(x = \infty) = \frac{1}{1 + 0} = 1$$

$$A(x = -\infty) = \frac{1}{1 + \infty} = 0$$

$$A(0) = 0.5$$

A função sigmoide, descrevendo uma curva suave por todo o domínio x, é exatamente uma ativação binária. Seu incremento, A' é [22]:

$$A' = A(1 - A)$$

Com extremos:

$$A'(\infty) = 1(1 - 1) = 0$$

$$A'(-\infty) = 0(1 - 0) = 0$$

$$A'(0) = 0.5(1 - 0.5) = 0.25$$

As contribuições para saídas muito grandes são nulas, de modo que a função sigmoide gera correções significativas somente pelas redondezas de $x = 0$. Como $x = 0$ resulta em $A = 0.5$, não ocorre a explosão de correções se a rede prevê zero.

Referências Bibliográficas

[1] D. dos Santos Silva, L. A. P. Andrade, and S. M. P. dos Santos, "Alternativas de ensino em tempo de pandemia," *Research, Society and development*, vol. 9, no. 9, pp. e424997177–e424997177, 2020.

[2] M. X. Yamaguti and J. T. de Carvalho Neto, "O uso de linguagem de programação no ensino de física: Uma revisão da literatura,"

[3] C. dos Santos Cruz, L. Q. Galvão, S. Rosa, and W. S. Santana, "O uso do python na construção de simuladores computacionais: proposições e potencialidades para o ensino de física," *Caderno Brasileiro de Ensino de Física*, vol. 39, no. 1, pp. 204–237, 2022.

[4] K. Gunnarsson and O. Herber, "The most popular programming languages of github's trending repositories," 2020.

[5] K. Svensson, U. Eriksson, and A.-M. Pendrill, "Programming and its affordances for physics education: A social semiotic and variation theory approach to learning physics," *Physical Review Physics Education Research*, vol. 16, no. 1, p. 010127, 2020.

[6] E. Adami, "Introducing multimodality," *GARCIA, O.; FLORES, N.; SPOTTI, M.(Orgs.)*, 2013.

[7] S. Ainsworth, "The educational value of multiple-representations when learning complex scientific concepts," in *Visualization: Theory and practice in science education*, pp. 191–208, Springer, 2008.

[8] M. A. Rau, "Conditions for the effectiveness of multiple visual representations in enhancing stem learning," *Educational Psychology Review*, vol. 29, pp. 717–761, 2017.

[9] J. Airey and C. Linder, "Multiple representations in physics education," *Multiple Representations in Physics Education*, vol. 10, pp. 95–122, 2017.

[10] T. Fredlund, C. Linder, J. Airey, and A. Linder, "Unpacking physics representations: Towards an appreciation of disciplinary affordance," *Physical Review Special Topics-Physics Education Research*, vol. 10, no. 2, p. 020129, 2014.

[11] U. Eriksson, C. Linder, J. Airey, and A. Redfors, "Introducing the anatomy of disciplinary discernment: an example from astronomy," *arXiv preprint arXiv:1703.00269*, 2017.

[12] J. Airey, U. Eriksson, T. Fredlund, and C. Linder, "The concept of disciplinary affordance," in *The 5th International 360 Conference. Encompassing the multimodality of knowledge, May 8-10 2014, Aarhus University, Denmark*, p. 20, 2014.

[13] J. Airey and C. Linder, "Social semiotics in university physics education: leveraging critical constellations of disciplinary representations," in *The 11th Conference of the European Science Education Research Association (ESERA), Aug. 31-4 Sept., 2015 Helsinki*, European Science Education Research Association, 2015.

[14] J. Bezemer and G. Kress, "Writing in multimodal texts: A social semiotic account of designs for learning," *Written communication*, vol. 25, no. 2, pp. 166–195, 2008.

[15] L. Ma, "Knowing and teaching elementary mathematics: Teachers' understanding of fundamental mathematics in china and the united states (studies in mathematical thinking and learning series)," 1999.

[16] S. Zhu and P. Meyer, "A comparative study of mathematics self-beliefs between students in shanghai-china and the usa," *The Asia-Pacific Education Researcher*, pp. 1–11, 2020.

[17] T. S. Volkwyn, J. Airey, B. Gregorcic, and F. Heijkenskjöld, "Transduction and science learning: Multimodality in the physics laboratory.," *Designs for Learning*, vol. 11, no. 1, pp. 16–29, 2019.

[18] F. Marton and S. Booth, *Learning and awareness*. Routledge, 2013.

[19] E. Grigsby, K. Lindsey, and D. Rolnick, "Hidden symmetries of relu networks," in *International Conference on Machine Learning*, pp. 11734–11760, PMLR, 2023.

[20] A. C. Marreiros, J. Daunizeau, S. J. Kiebel, and K. J. Friston, "Population dynamics: variance and the sigmoid activation function," *Neuroimage*, vol. 42, no. 1, pp. 147–157, 2008.

[21] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[22] A. A. Minai and R. D. Williams, "On the derivatives of the sigmoid," *Neural Networks*, vol. 6, no. 6, pp. 845–853, 1993.