Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Matemática Aplicada

Redes Neurais aplicadas

Eduardo Galvani Massino

Monografia Final

map2010 — Trabalho de Formatura

Orientador: José Coelho de Pina Junior

São Paulo
Dezembro de 2020

Esta seção é opcional e fica numa página separada;
ela pode ser usada para uma dedicatória ou epígrafe.

Agradecimentos

Do. Or do not. There is no try.

— Mestre Yoda

Texto texto. Texto opcional.

Resumo

Eduardo Galvani Massino. **Redes Neurais aplicadas**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2020.

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

Palavras-chave: redes-neurais. perceptron.

Abstract

Eduardo Galvani Massino. **Redes Neurais aplicadas**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2020.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

Keywords: neural-nets. perceptron.

Lista de Figuras

Lista de Programas

Sumário

Anexos

Capítulo 1

Introdução

De tempos pra cá, ler e ouvir falar de **ciência de dados** tornou-se muito comum, tanto nos meios profissionais e científicos quanto na mídia. Existem atualmente aplicações em praticamente todas as áreas do conhecimento humano, da agricultura à indústria e ao entretenimento.

Uma busca rápida na *Wikipedia* (*Ciência de Dados* 2020) define ciência de dados como um conjunto de ferramentas que extrai informações ou previsões a partir de um grande volume de dados, que podem ser números, textos, áudio, vídeo, entre outros, para ajudar na tomada de decisões de negócios.

Apesar de não ser a única definição para o termo, Pedro A. Morettin e Julio M. Singer (Morettin e Singer, 2020) nos lembram que essa também é uma definição da estatística. Eles comparam o uso dos termos e apontam que o trabalho dos *cientistas de dados* diferem dos *estatísticos* apenas quando eles usam dados de natureza multimídia como áudio e vídeo, por exemplo. Mas que, uma vez que esses dados são processados e tornam-se números, as técnicas e conceitos utilizados pelos primeiros passam a ser basicamente os mesmos utilizados pelos segundos.

Na verdade, Morettin & Singer (Morettin e Singer, 2020) citam que na década de 80 houve uma primeira tentativa de aplicar o rótulo *ciência de dados*, (*Data Science*), ao trabalho feito pelos estatísticos aplicados da época, como uma forma de dar-lhes mais visibilidade. Curiosamente, fato mencionado pelos autores, existem atualmente cursos específicos de ciência de dados em universidades ao redor do mundo, mas a maioria deles situada em institutos de áreas aplicadas como engenharia e economia, e raramente nos institutos de estatística propriamente ditos.

Para entender um pouco mais de seu escopo, David M. Blei e Padhraic Smyth (Blei e Smyth, 2017) discutem ciência de dados sob as visões estatística, computacional e humana. Eles argumentam que é a combinação desses três componentes que formam a essência do que ela é e, assim como, do conhecimento que ela é capaz de produzir.

Em resumo, a estatística guia a coleta e análise dos dados. A computação cria algoritmos, técnicas de processamento e gerenciamento de memória eficazes para que sua execução seja efetiva. E o papel humano é o de avaliar quais tipos de dados, técnicas de análises,

algoritmos e modelos são apropriados para responder ao problema em questão. Este é o papel do *cientista de dados*.

Algoritmos de **aprendizado de máquina** vem sendo utilizados em grande parte dos modelos de ciência de dados. Mas o que é aprendizado de máquina? Ou então, o que significa dizer que o computador, neste caso a "máquina", está *aprendendo*?

Aurélien Géron (Géron, 2019) nos dá uma ideia geral lembrando que uma das primeiras aplicações de sucesso de aprendizado de máquina foi o filtro de *spam*, criado na década de 90. Uma das fases de seu desenvolvimento foi aquela em que os usuários assinalavam que certos e-mails eram *spams* e outros não eram. Hoje em dia, raramente temos que marcar ou desmarcar e-mails, pois a maioria dos filtros já "aprenderam" a fazer seu trabalho de forma muito eficiente, não temos mais nada a "ensiná-lo".

O conceito de aprendizado de máquina está intimamente ligado à ciência da computação. Porém, no contexto de ciência de dados, é definido por Joel Grus (Grus, 2016) como a "criação e o uso de modelos que são ajustados a partir dos dados". Seu objetivo é usar dados existentes para desenvolver modelos que possamos usar para *prever* possíveis respostas à consultas. Exemplos, além do filtro de *spams* podem ser: detectar transações de crédito fraudulentas, calcular a chance de um cliente clicar em uma propaganda ou então prever qual time de futebol irá vencer o Campeonato Brasileiro.

Como ficará claro ao longo deste trabalho, o aprendizado consiste na utilização de dados já conhecidos para ajustar parâmetros de modelos. Uma vez ajustados os parâmetros, o algoritmo que descreve o modelo passa a ser usado para responder às consultas. Essa fase de ajuste de parâmetros é chamada de aprendizado ou treinamento.

Uma **rede neural** é um exemplo de modelo preditivo de aprendizado de máquina que foi criado com inspiração no funcionamento do cérebro biológico. David Kopec (Kopec, 2019) descreve que apesar de terem sido as primeiras a serem criadas, elas vem ganhando nova importância na última década, graças ao avanço computacional, uma vez que exigem muito processamento, e também porque podem ser usadas para resolver problemas de aprendizagem dos mais variados tipos.

Atualmente existem vários tipos de redes neurais, porém este trabalho lida principalmente com aquele tipo que foi originalmente criado sob a inspiração do funcionamento do cérebro, chamado de **perceptron**, e que portanto tenta imitar o comportamento dos neurônios e suas conexões, aprendendo padrões a partir de dados existentes e tentando prever o comportamento de dados novos a partir do padrão aprendido.

Uma visão geral da arquitetura do aprendizado de máquina, situando a posição das redes neurais e do algoritmo *perceptron* em toda esta estrutura, assim como exemplos de aplicações em cada uma das suas ramificações, estão no Capítulo 2.

Neste trabalho é utilizada como base uma versão simples do algoritmo *perceptron* feita e explicada por Kopec (Kopec, 2019), e a partir desta base, foram criados novos métodos de treinamento e de validação com algumas estratégias, como uma tentativa de automatizar e otimizar o processo de treinamento do algoritmo, que é comumente feito de forma heurística. Detalhes dessa implementação estão no Capítulo 3.

São apresentados os conceitos e usos das séries temporais de dados não-lineares no Capítulo 4, assim como alguns exemplos de aplicações na área financeira. As técnicas tradicionais de análise e de previsão de séries temporais de dados serão brevemente apresentadas neste capítulo.

Para servir de validação e aplicação do algoritmo criado, no Capítulo 5 serão feitas comparações de desempenho entre o *perceptron* e os modelos tradicionais de previsões de séries temporais apresentados no capítulo anterior, como o **SARIMA** (*seasonal auto regressive integrated moving average*). Tais comparações usam como inspiração trabalhos similares como o de Alsmadi, Omar, Noah e Almarashdah (Alsmadi *et al.*, 2009).

Ao final concluo sobre os modelos de previsões aqui estudados e comparados, destacando a qualidade e eficiência dos métodos de redes neurais, tão bons e às vezes melhores do que os métodos tradicionais estatísticos.

Capítulo 2

Redes neurais artificiais

Neste capítulo são apresentados alguns conceitos básicos de **aprendizado de máquina**, com foco nos algoritmos de redes neurais artificiais, em especial o **perceptron**, que teve seu desenvolvimento inspirado nas redes neurais biológicas, ou seja, os neurônios e suas conexões no cérebro, conforme descrito por Kopec (Kopec, 2019).

Pode-se classificar as técnicas de aprendizado de várias formas, de acordo com alguma de suas características. Por exemplo, Géron (Géron, 2019) utiliza o grau de supervisão humana durante o seu funcionamento para classificá-los em aprendizado supervisionado ou não-supervisionado. Durante o aprendizado podem ou não ser fornecidos um conjunto de consultas e de respostas esperadas. Tais respostas foram dadas por humanos, ao menos neste momento, daí o termo "supervisão humana".

Neste texto os termos "algoritmo" e "técnica" serão usados livremente como sinônimos, pois uma técnica de aprendizado de máquina, no contexto atual, é um algoritmo executado no computador que tem por objetivo ajustar parâmetros de modelos estatísticos.

2.1 Aprendizado de máquina supervisionado

Um algoritmo de **aprendizado supervisionado** é usado quando conhecemos características dos dados que estamos utilizando. De modo geral já temos de antemão as respostas às consultas para os dados utilizados no treinamento. Por exemplo, se estamos classificando fotos de animais, possuímos um conjunto de fotos em que já sabemos quais são de gatos, cachorros, etc.

O ato de rotular previamente os dados que usamos no treinamento é o que designamos de supervisão humana. Uma vez *treinado*, o algoritmo recebe uma foto, ou seja, uma nova consulta e então fornece a resposta, neste caso se essa é a foto de um gato, ou cachorro, ou qualquer outra resposta dentre aquelas que foram dadas como exemplos durante o treinamento.

Dentro do aprendizado supervisionado temos duas técnicas principais. A primeira é a regressão, usada para prever valores, ou seja, fornecer respostas a consultas ainda inéditas,

sejam dados do futuro ou valores de funções em pontos do domínio para os quais ainda não existem respostas.

A segunda técnica é a classificação, usada para rotular ou dividir os dados em classes pré-determinadas, a partir de exemplos, que é exatamente o caso dos exemplos descritos nos parágrafos anteriores. Se as classes não são conhecidas deve-se utilizar um algoritmo de aprendizado não-supervisionado, descrito na próxima seção.

Colocar exemplos...

2.2 Aprendizado não-supervisionado

Nesse tipo de aprendizado de máquina, não sabemos os rótulos dos dados que estamos lidando, assim o algoritmo poderá agrupar os dados de forma automática, por exemplo, se estivermos lidando com problemas de classificação. Aqui, as consultas podem ser coisas como "quantos são os perfis dos clientes" ou "quantas espécies de flores existem nestas fotos", e assim por diante.

Alguns métodos não-supervisionados de aprendizado foram enumeradas por Géron (Géron, 2019). O **agrupamento** de dados similares sob uma inspiração geométrica. Nesse caso os dados são agrupados conforme suas posições num determinado espaço e utiliza-se algoritmos como k-vizinhos, k-means, k-medians, etc. Exemplos de aplicações são agrupamento de produtos em supermercados, interesses comuns de clientes em sites de conteúdo digital, etc.

Outra técnica é a **detecção de anomalias**, cujo objetivo é ter uma descrição de como os dados considerados "normais" se parecem, e usa-se esse agrupamento para detectar se novos dados estariam "fora" desse padrão. Um exemplo é a detecção de fraudes.

Também pode-se citar sobre a técnica de **estimação de densidades**, que tem como objetivo a estimação da função densidade de probabilidade de um conjunto de dados gerados por algum processo aleatório.

Colocar exemplos...

2.3 Técnicas de classificação

É uma das técnicas principais do aprendizado supervisionado, problemas desse tipo buscam aprender com um conjunto de dados previamente rotulados. Consultas do tipo "a qual grupo pertence este cliente?" ou "que animal há nesta foto?" podem ser modeladas por esses algoritmos. De modo geral, ele responde a consultas que dizem respeito às classes dos dados, e atribui para novos dados alguma classe que pertence ao conjunto de classes que usamos para rotular os dados iniciais.

Existem vários tipos de algoritmos de classificação, e dentre os descritos por Géron (Géron, 2019) citamos a máquina de vetor suporte (*support vector machine*, SVM), árvores de decisão, florestas aleatórias que são um conjunto de muitas árvores de decisão aleatoriamente definidas e, finalmente, as redes neurais artificiais.

Todas essas técnicas podem ser usadas para classificação linear ou não-linear, no sentido em que valores eles estão classificando, assim como na forma que está sendo feita essa classificação. Se visualizarmos os dados num espaço bidimensional, um algoritmo de classificação linear irá separar as classes de dados por retas, enquanto que um classificador não-linear poderá usar outra curva qualquer para a separação.

Abstraindo o espaço bidimensional para os espaços multidimensionais dos dados que são comumente analisados, podemos pensar em hiperplanos, estruturas $(n-1)$-dimensionais de espaços n-dimensionais, para o caso dos classificadores lineares, ou subespaços quaisquer para os não-lineares.

Colocar exemplos de aplicações...

2.4 A rede neural perceptron

Uma rede neural artificial é um dentre vários métodos de classificação, ou seja, de aprendizado supervisionado, embora ela também possa ser usada para aplicaçõesd de aprendizado não supervisionado. De acordo com Kopec (Kopec, 2019), ele é utilizado como um classificador não-linear, e por isso pode ser utilizado para classificar ou prever quaisquer tipos de funções, que podem ou não ter uma relação linear com o tempo ou com qualquer outro domínio no qual estejam definidas.

Uma definição para uma rede neural artificial dada por Rosangela Ballini (Ballini, 2000) é a de um sistema de processamento paralelo e distribuído baseado no sistema nervoso biológico, sendo compostos por elementos computacionais chamados neurônios, arranjados em padrões semelhantes às redes biológicas.

Na figura 2.1 está uma representação de um neurônio biológico. Ele recebe impulsos elétricos de entrada através dos dentritos, que são transmitidos ou não através do núcleo, caso sejam ativados por ele, para os terminais de saída dos axônios. Os neurônios se comunicam através de sinapses, que são ligações entre os dentritos de um e os axônios de outro que realizam a transmissão dos sinais.

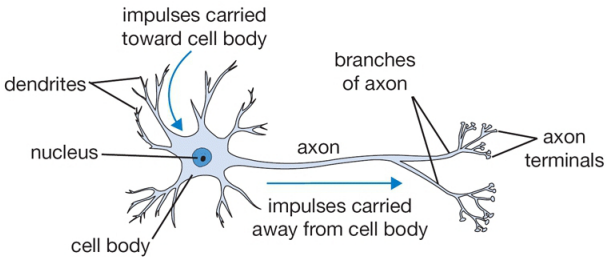

Figura 2.1: *Representação de um neurônio biológico.*[a]

[a]Extraído de https://cs231n.github.io/neural-networks-1/

Dá-se o nome de *perceptron* de camada única (*single-layer perceptron*) ou simplesmente *perceptron* a uma das primeiras e mais simples rede neural artificial a ser criada. Uma ilustração conceitual dela está na figura 2.2. Mais recentemente foram criadas várias

outras versões dessa rede, dentre as quais podemos citar o *perceptron* de multi-camadas (*multi-layer perceptron*).

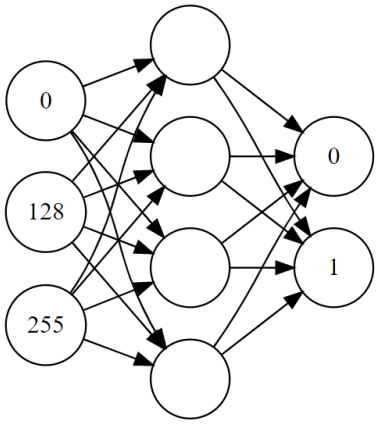

Figura 2.2: *Rede neural simples, o perceptron de camada única.*

Os neurônios são representados por círculos, dentro deles há um valor numérico que intuitivamente podemos atribuir ao nível ou grau de ativação do neurônio, mesmo que no caso biológico se restrinja aos valores 0 e 1, ou seja, ativados ou não. Cada coluna de neurônios representa uma camada, nesse caso, da esquerda para a direita temos a camada de entrada, a camada oculta e a camada de saída. As linhas representam as ligações entre os neurônios, sendo que cada neurônio de uma camada está ligado a todos da camada anterior.

O perceptron de camada única consiste de uma camada de neurônios de entrada, uma camada oculta de neurônios usados na otimização, e uma camada de saída, que irá conter os dados previstos, ou ainda as probabilidades do dado pertencer a alguma das classes que a rede poderá classificá-lo. E é o fato de haver uma camada oculta nesta rede que a define como sendo de "camada única". Caso houvessem mais do que uma camada oculta, ela seria do tipo "multi-camadas" mencionada acima.

De modo a entendermos as bases matemáticas do algoritmo, podemos começar de uma rede ainda mais básica, a partir um *perceptron* que seja constituído de apenas 1 neurônio na única camada oculta. Esta rede super simplificada, que está na figura 2.3, pode ser útil para para o entendimento uma vez que neste caso será possível acompanhar graficamente o resultado da execução do algoritmo.

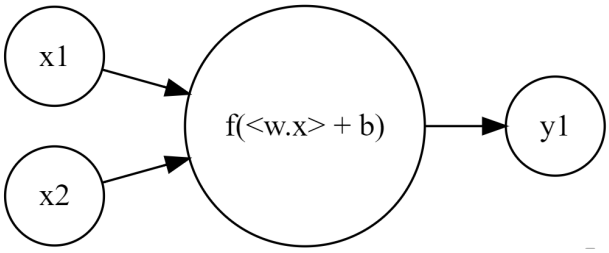

Figura 2.3: *Rede neural mais simples ainda, apenas um neurônio oculto.*

Esta rede possui 2 neurônios na camada de entrada, que são os números reais x_1 e x_2, 1 neurônio na camada oculta, no qual está a sua função de ativação $f(x_1 w_1 + x_2 w_2 + b)$,

e 1 neurônio na camada de saída, que neste caso é um número real y_1. Pode-se notar a semelhança dessa rede neural artificial com a sua inspiração biológica com a ajuda da figura 2.4.

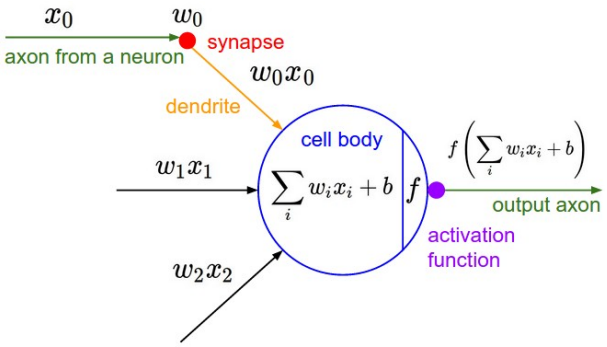

Figura 2.4: *Representação de um neurônio artificial.*[a]

[a]Extraído de https://cs231n.github.io/neural-networks-1/

Temos os sinais de entrada (como o x_1) vindos como viriam os sinais dos axônios de outros neurônios. Eles entram pela camada de entrada da rede, ou dentritos do neurônio. A camada oculta processa as entradas com os pesos, definindo o formato final do sinal através de sua função de ativação, que aqui pode ser uma função real qualquer, mas com funcionalidade similar ao do núcleo do neurônio que ativa/transmite ou não o sinal recebido por ele. Por fim o sinal é enviado à camada de saída, ou aos axônios do neurônio, concluíndo o processamento.

A partir desta analogia podemos compreender o funcionamento básico da rede artificial *perceptron*. Ela recebe uma lista de valores como entrada, que podemos representar por um vetor real x. O neurônio oculto representa uma transformação linear neste vetor, que podemos escrever como o produto escalar por um outro vetor real, o vetor de **pesos** w, ou seja, $< w.x >$, que é o produto escalar usual dos números reais. A seguir, somamos um outro número real b que é chamado de **viés**, que possui o mesmo papel que a constante de interceptação da reta com o eixo vertical de um ajuste linear.

Por fim, é aplicada uma função de ativação não-linear sobre esta transformação, o que configura a saída deste neurônio: $f(< w.x > + b)$, que é transmitida ao neurônio de saída, que pode aplicar uma transformação semelhante ou outra qualquer, dependendo da função de ativação utilizada em cada camada da rede. Por simplicidade mostramos uma rede bem simples, mas na prática podem haver muito mais camadas ocultas, e cada uma delas assim como a camada de saída, podem ter muitos neurônios cada.

Este processo de entrada, processamento e saída da rede é chamado de **feedforward**, e consiste no nível mais fundamental do *perceptron*. A partir daí, a forma como a rede será treinada, é o que define se ela será utilizada para um aprendizado supervisionado ou não-supervisionado.

Uma vez que estamos lidando com o aprendizado supervisionado, dever ser utilizado um algoritmo de treinamento que forneça à rede pares conhecidos de vetores de entradas e saídas esperadas, e um critério de avaliação de quão boa é a performance da rede para aproximar as suas saídas das saídas esperadas.

Este critério é uma função que fornece uma medida da distância entre as saídas obtidas pela rede e as saídas esperadas, que é genericamente chamada de função de custo (*cost function*). Se denotarmos por y uma saída conhecida, e por $a^{(L)}$ uma saída obtida pela última camada, exemplos comumente usados são as normas usuais como a distância euclidiana ($(a^{(L)^2} + y^2)^{1/2}$), a função de erro absoluto ($|a^{(L)} - y|$), e a função de erro quadrático médio ($(a^{(L)} - y)^2$) , (*mean square error*, MSE), que é a usada no algoritmo descrito por Kopec (Kopec, 2019) e que será usado neste trabalho.

2.5 Derivação matemática de um algoritmo de aprendizado

Um dos algoritmos de treinamento que minimizam uma função de custo é o gradiente descendente (*gradient descent*), que segundo Géron (Géron, 2019) é um algoritmo muito geral e que serve para encontrar soluções ótimas para uma grande variedade de problemas de otimização. A ideia geral é ajustar parâmetros iterativamente para otimizar a função de custo gradativamente.

O seu funcionamento utiliza a ideia fundamental do Cálculo em que utilizamos a derivada de uma função de forma a encontrar seus pontos extremos. Dada uma função $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, temos que se um ponto $x = \hat{x} \in \mathbb{R}$ é um ponto extremo de f então é condição necessária[1] que cada derivada parcial de primeira ordem de f exista e seja igual a zero. Denotando $x = (x_0, x_1, \dots, x_n)$ e $\hat{x} = (\hat{x_0}, \hat{x_1}, \dots, \hat{x_n})$, temos:

$$\frac{\partial f(\hat{x_0})}{\partial x_0} = 0, \quad \frac{\partial f(\hat{x_1})}{\partial x_1} = 0, \quad \dots, \quad \frac{\partial f(\hat{x_n})}{\partial x_n} = 0 \tag{2.1}$$

Usando a notação de vetores, podemos simplificar a equação acima, uma vez que o conjunto das derivadas parciais de uma função de várias variáveis é o vetor gradiente desta função, assim, denotando $\mathbf{0} \in \mathbb{R}^n = (0, 0, \dots, 0)$, temos equivalentemente à equação 2.1:

$$\nabla f(\hat{x}) = \mathbf{0} \tag{2.2}$$

onde $\nabla : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ é a função que calcula o gradiente para um dado ponto de uma função.

Geometricamente é nos dada a intuição, por Luis Hamilton Guidorizzi (Guidorizzi, 1986), de que o vetor gradiente de um dado ponto de uma função nos dá a direção de maior aumento da função naquele ponto. Como nosso objetivo é minimizar a função de custo, fica explicado o nome do algoritmo como "gradiente descendente", de forma que devemos utilizar o sentido negativo do vetor gradiente.

Assim, podemos dizer que a direção de minimização da função está na direção do vetor gradiente, o que significa dar um passo ($f(x + dx)$) nessa direção no domínio da função, tal passo com tamanho que seja *proporcional* a cada componente do vetor gradiente. Dessa

[1] Mais detalhes em Guidorizzi (Guidorizzi, 1986). Um curso de cálculo Vol. 2, pág. 894.

forma podemos escrever dx como sendo um passo na direção do mínimo da função de custo dessa forma:

$$dx = -\eta \nabla f(x) \tag{2.3}$$

onde η é a constante de proporcionalidade, que é conhecida como **taxa de aprendizado**, que tem por objetivo tornar a velocidade do treinamento ajustável durante a execução do algoritmo, sendo tarefa do cientista de dados testar e obter os valores que dêem os melhores resultados caso-a-caso.

Podemos observar as diferenças de utilizar uma taxa de aprendizado fixa ou variável (que mude a cada iteração do treinamento, por exemplo), utilizando gráficos. Suponha que estamos querendo minimizar uma função quadrática do tipo $f(x) = x^2$, essa restrição é particularmente útil uma vez que a função de custo que será utilizada, a MSE, é uma função quadrática deste tipo. Outra característica igualmente útil é que a segunda derivada deste tipo de função quadrática é sempre positiva[2], o que implica que o ponto extremo encontrado será necessariamente um ponto de mínimo.

Inicialmente, sorteamos um ponto inicial e calculamos o valor da função e o valor do gradiente neste ponto. Se as componentes do gradiente não forem todas nulas (ou arbitrariamente próximas de zero), então quer dizer que não atingimos o mínimo, e dessa forma obtemos um novo ponto a partir deste somado com dx definido acima na equação 2.3. Repetimos este processo até que o gradiente do ponto atual seja arbitrariamente próximo do vetor nulo. Uma visualização deste processo, utilizando taxa de aprendizado fixada, está na Figura 2.5.

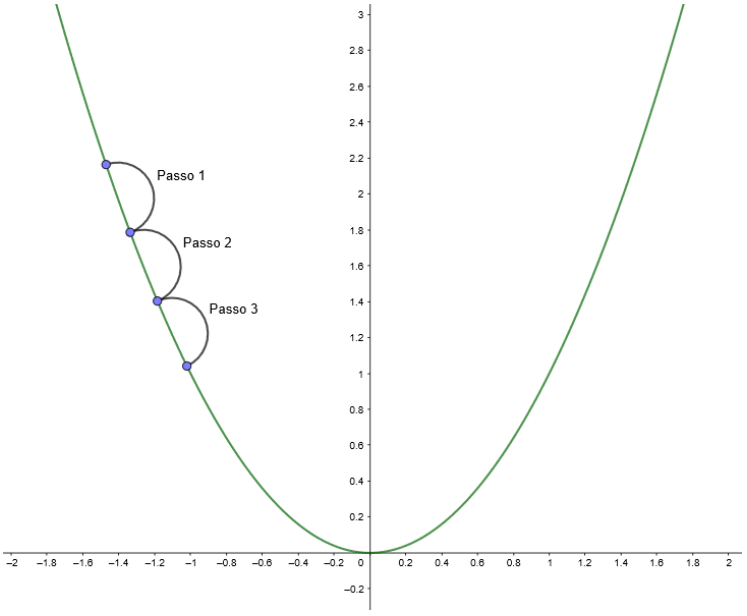

Figura 2.5: *Visualização do método do gradiente descendente com taxa de aprendizado única. Os pontos azuis representam candidatos a ponto mínimo em cada iteração do algoritmo.*

[2] A segunda derivada de uma função de muitas variáveis é a matriz Hessiana, neste caso ela seria uma matriz definida positiva.

Podemos notar que as estimativas aproximam-se a uma velocidade constante do ponto de mínimo, que nesse caso ilustrativo é bem conhecido. Esse é o comportamento gerado por uma taxa de aprendizado fixa, e além disso com uma magnitude mediana. O que poderia acontecer se utilizarmos uma taxa de aprendizado muito grande é que com um passo do algoritmo, o ponto estimado poderia ir para o outro lado do arco da função, e depois retornar, e assim por diante, nunca convergindo para o mínimo, uma ilustração disso está na Figura 2.6.

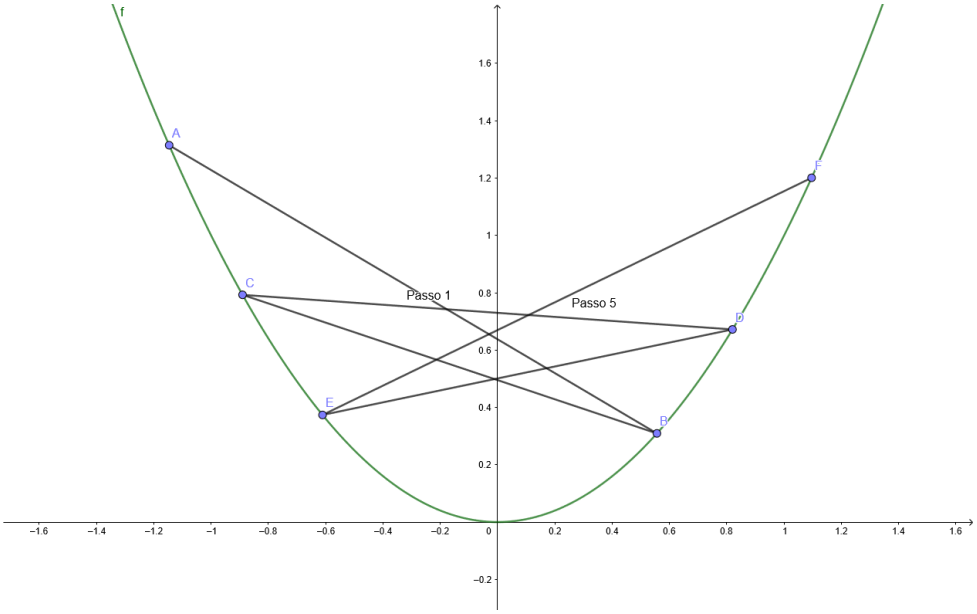

Figura 2.6: *Visualização do método do gradiente descendente com taxa de aprendizado única. Ilustração do uso de um valor de taxa de aprendizado muito grande.*

O caso oposto a este, ou seja, usar uma taxa muito pequena, claramente irá fazer com os passos dados sejam muito pequenos, e dessa forma o algoritmo demore muito a convergir, por isso é importante usar valores medianos que podem ser obtidos de forma heurística, embora na prática, conforme dito por Géron (Géron, 2019), utiliza-se $\eta = 0.1$, sendo este um valor consensualmente utilizado pelo menos como ponto de partida.

A outra abordagem é utilizar valores variáveis, sendo o caso mais comum utilizar uma taxa que começa até mesmo maior do que o valor comum de 0.1 mas que vai diminuindo a cada passo, numa tentativa de obter uma convergência mais rápida. Uma ilustração desse caso está na Figura 2.7.

Qualquer que seja o tipo de taxa de apredizado que venha a ser utilizado, permanece como melhor estratégia testar qual deles irá gerar o melhor resultado, a partir de algum problema com solução previamente conhecida, como a função $f(x) = x^2$, em que sabemos que o ponto de mínimo é atingido em $x = 0$, ou podemos testar em nosso problema-alvo, analisando diretamente os valores candidatos a mínimo obtidos pelo algoritmo como função do número do passo, criando assim outro tipo de gráfico, no qual não precisamos saber o formato da função objetivo, o que é razoável uma vez que não precisariamos de um método numérico para obter seu ponto de mínimo.

Podemos observar este comportamento genérico para os 4 casos acima mencionados,

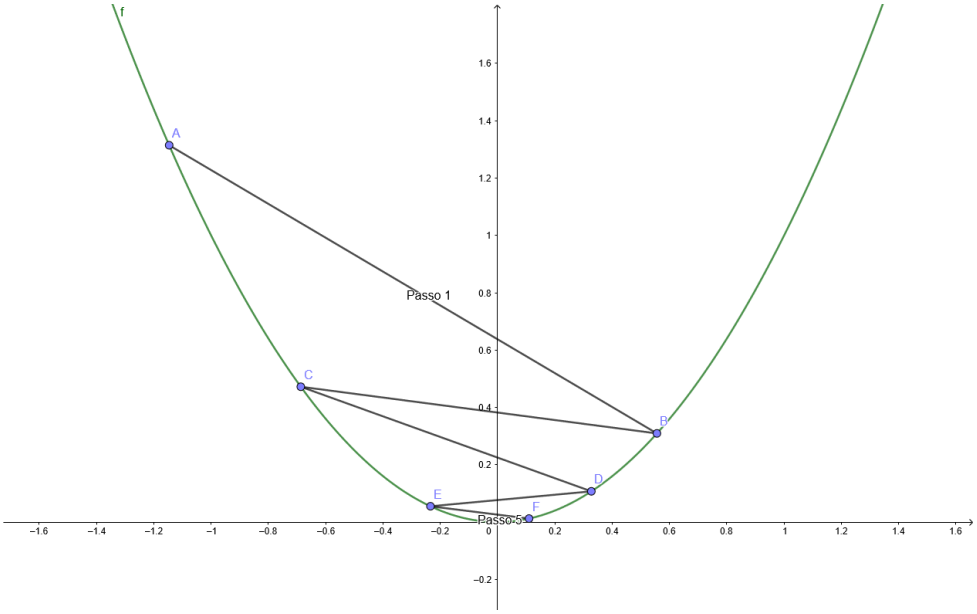

Figura 2.7: *Visualização do método do gradiente descendente com taxa de aprendizado variável que vai diminuindo passo-a-passo do algoritmo.*

na Figura 2.8 temos os gráficos de valores hipotéticos de candidatos a mínimo gerados por (a) taxa de aprendizado fixa e grande, (b) taxa de aprendizado fixa e pequena, (c) taxa de aprendizado fixa e mediana, (d) taxa de aprendizado decrescente.

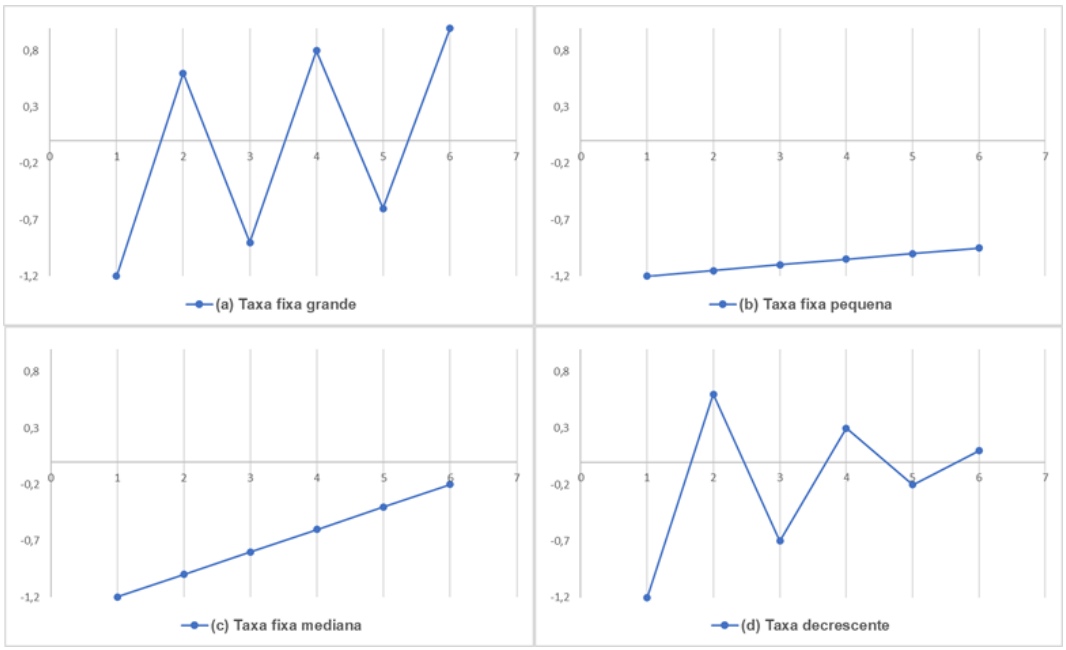

Figura 2.8: *Comportamento de diferentes taxas de aprendizado nos valores candidatos a mínimo.*

A partir deste comportamento geral, podemos testar nosso problema-alvo, verificar a qual comportamento ele mais se parece e assim decidir se devemos aumentar ou diminuir nossa taxa até obtermos um bom comportamento como aqueles vistos em (c) ou (d).

Colocar exemplos de outros algoritmos de otimização de função de custo...

Capítulo 3

Perceptron multi-camadas

Neste capítulo é descrita a implementação e funcionamento de uma versão do algoritmo *perceptron*, feito a partir de um núcleo básico disponibilizado no livro de Kopec (Kopec, 2019), e a partir do qual foram feitas modificações e criação de novos métodos de treinamento, de validação e de avaliação do treinamento.

O perceptron aqui implementado tem o objetivo de ser utilizado muito mais para fins didáticos do que práticos e pode ser usado para tarefas de aprendizagem contanto que sejam problemas que envolvam bases de dados de tamanho pequeno ou mediano, e neste capítulo é apresentadao um caso de uso para um problema de classificação de dados.

Na última parte desse capítulo é exibida uma biblioteca de *machine learning* utilizada nas aplicações reais de *deep learning* de redes neurais, muito mais avançada, com muitos outros recursos que vão além do escopo que esta versão simples do *perceptron* é capaz de lidar. Esta será a biblioteca de redes neurais que será utilizada nas partes práticas deste trabalho.

3.1 Derivação matemática do algoritmo de retropropagação

Para a aprendizagem supervisionada foi utilizado o algoritmo de retropropagação (*retropropagation*), que consiste na minimização de uma função de custos, a partir do gradiente, ou seja, da derivada desta função de custos, neste caso o erro quadrático médio, conforme foi definido no capítulo anterior.

De acordo com Kopec (Kopec, 2019), o perceptron consiste de uma rede cujo sinal, ou seja, os dados, se propagam em uma só direção, da camada de entrada para a camada de saída, passando pelas camadas ocultas uma a uma, e por isso o nome de rede *feedforward* ao perceptron. Por sua vez, o erro que determinamos na camada final propaga-se no caminho inverso, sendo distribuídas correções da saída para a entrada, afetando aqueles neurônios que foram mais responsáveis pelo erro total. Por isso o nome de retropropagação.

Estendendo as definições já usadas no capítulo anterior, segue a derivação matemática do algoritmo de retropropagação. Como ficará claro mais à frente, podemos derivar as

contas para apenas um neurônio por camada sem perda de generalidade. Dessa forma, se temos uma rede com L camadas, o erro quadrático para um neurônio da camada de saída (a camada L) será:

$$C_0 = (a^{(L)} - y)^2$$

onde y é a saída esperada, e $a^{(L)}$ é a saída de um neurônio da camada de saída.

Temos que C_0 é uma função de $a^{(L)}$, uma vez que y é um valor fixo conhecido. Por sua vez, temos que de modo geral a saída de um neurônio é uma função do tipo:

$$a^{(L)} = \sigma(w^{(L)}\ a^{(L-1)} + b^{(L)})$$

onde escrevemos $a^{(L-1)}$ é a saída do neurônio da camada anterior, $w^{(L)}$ é o **peso** atribuido a essa saída, o que seria o parâmetro angular A na Figura 2.3, e $b^{(L)}$ é o chamado **viés** desse neurônio, análogo ao parâmetro linear de uma reta. Por fim temos a **função de ativação** que escrevemos como σ que é aplicada à essa equação linear.

Nota-se que internamente à função de ativação, um neurônio se comporta como uma transformação linear dos neurônios da camada anterior. Caso tivéssemos n neurônios na camada anterior à de saída, teríamos então n pesos, denotados com índice i dessa forma: $\{w_i^{(L)}\}_{i=1}^{n}$. Cabe assim à função de ativação, dar o comportamento não-linear à rede perceptron.

Como o objetivo é minimizar C_0, temos que calcular a influência dos pesos e dos viéses nesse custo. Já sabemos que isso será obtido com o gradiente, isto é, a derivada dessa função em relação a esses parâmetros que, são os únicos que podemos otimizar. De forma mais clara, temos que no início do treinamento da rede, atribuímos valores aleatórios aos pesos e aos viéses, e então executamos o *feedforward*, de forma que a rede irá calcular sequencialmente os valores de saída em todas as suas camadas, obtidos a partir dos dados de entrada, que serão fixos, e desses parâmetros inicialmente aleatórios. A partir daí, poderemos otimizar esses parâmetros, exatamente da forma que estamos construindo.

O cálculo dessas derivadas é feito segundo a regra da cadeia, e adicionalmente iremos denotar a transformação linear interna à função de ativação por $z^{(L)} = w^{(L)}\ a^{(L-1)} + b^{(L)}$, de forma que $a^{(L)} = \sigma(z^{(L)})$. Assim, ficamos com as derivadas para a camada de saída:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{3.1}$$

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{3.2}$$

Para a camada de saída, podemos calcular diretamente cada termo dessas derivadas:

$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y) \ \propto \ (a^{(L)} - y) \tag{3.3}$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}) \tag{3.4}$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} \tag{3.5}$$

$$\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1 \tag{3.6}$$

O que resulta, fazendo todas as substituições, em:

$$\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)}\, \sigma'(z^{(L)})\, (a^{(L)} - y) \tag{3.7}$$

$$\frac{\partial C_0}{\partial b^{(L)}} = \sigma'(z^{(L)})\, (a^{(L)} - y) \tag{3.8}$$

Na equação 3.3 ocultamos o termo constante 2 sob um símbolo de proporção, que a seguir iremos também ocultar, uma vez que usaremos o algoritmo do gradiente descendente, e assim, em seu lugar, e na verdade, todas as derivadas aqui mostradas serão multiplicadas pelo termo η, a **taxa de aprendizagem**, conforme explicado no capítulo anterior.

Analogamente, podemos pensar numa forma de fazer esses cálculos para as camadas ocultas. A princípio, podemos calcular:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{3.9}$$

Usando o fato de que:

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)} \tag{3.10}$$

Agora, seja a i-ésima camada oculta tal que $1 < i < L$, se observarmos a equação 3.9, e fizermos $i = L - 1$, usando a equação 3.10, ficamos com:

$$\frac{\partial C_0}{\partial a^{(i)}} = w^{(i+1)} \frac{\partial a^{(i+1)}}{\partial z^{(i+1)}} \frac{\partial C_0}{\partial a^{(i+1)}} \tag{3.11}$$

Podemos observar que há um mesmo termo duplo que aparece tanto nas equações 3.1 e 3.2 quanto na equação 3.11 acima, de forma que apenas o índice da camada é diferente. Para simplificar podemos nomear esse termo de *delta da camada i*:

$$\Delta^{(i)} = \frac{\partial a^{(i)}}{\partial z^{(i)}} \frac{\partial C_0}{\partial a^{(i)}} \tag{3.12}$$

Simplificando todas as demais expressões usando essa definição, ficamos com:

$$\frac{\partial C_0}{\partial w^{(i)}} = a^{(i-1)}\Delta^{(i)} \tag{3.13}$$

$$\frac{\partial C_0}{\partial b^{(i)}} = \Delta^{(i)} \tag{3.14}$$

Como vemos, as derivadas que precisamos todas dependem desse termo Δ, que por sua vez depende do cálculo do termo $\frac{\partial C_0}{\partial a^{(i)}}$ que será calculado de 2 formas distintas:

$$\begin{aligned} &\frac{\partial C_0}{\partial a^{(i)}} = w^{(i+1)}\,\Delta^{(i+1)} \implies \\ &\Delta^{(i)} = \sigma'(z^{(i)})\,w^{(i+1)}\,\Delta^{(i+1)} \end{aligned} \tag{3.15}$$

para as camadas ocultas.

$$\begin{aligned} &\frac{\partial C_0}{\partial a^{(L)}} = (y - a^{(L)}) \implies \\ &\Delta^{(L)} = \sigma'(z^{(L)})\,(y - a^{(L)}) \end{aligned} \tag{3.16}$$

para a camada de saída.

Percebe-se a natureza recursiva do algoritmo, onde o caso base é calculado na camada de saída, e que o cálculo vai propagando-se para as camadas ocultas, em direção à camada de entrada. Por essa mesma razão, pudemos derivar as contas para uma camada, e no fim elas estão prontas pra serem implementadas para qualquer número de camadas ocultas.

Outro fato útil é que a expressão interna do neurônio é uma transformação linear, assim as contas podem ser facilmente ajustadas para o caso geral em que há n_i neurônios em dada camada i da rede, conforme já explicado, e que será detalhado diretamente nos trechos de código que serão mostrados a seguir na implementação propriamente dita.

3.2 Implementação do algoritmo de retropropagação

A implementação seguiu uma estrutura orientada a objetos, voltemos então à representação visual da rede perceptron, a partir da imagem 3.1. Cada círculo representa um neurônio, cada coluna vertical de neurônios é uma camada da rede, uma das camadas, a camada oculta nesse caso, está destacada em roxo na imagem. As setas representam as conexões entre as camadas de neurônios, cada neurônio de uma camada está ligado a todos os neurônios da camada anterior, o sentido dessa conexão é da esquerda pra direita, o que indica o processo de *feedforward* da rede.

A implementação do perceptron deste trabalho teve como base a implementação feita por Kopec (Kopec, 2019), a partir da qual foram adicionados outros recursos, como o viés

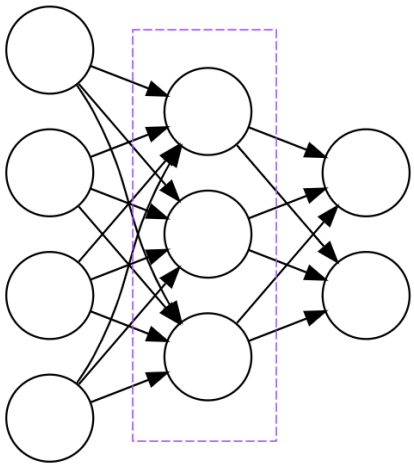

Figura 3.1: *Visão estrutural da rede perceptron. A linha tracejada destaca uma das camadas da rede.*

dos neurônios, não presentes na implementação de Kopec, como o uso da biblioteca *Numpy* para o uso de seus métodos mais eficientes para lidar com listas de números de ponto flutuante. Além dessa base, também estão implementadas várias outras classes, que serão explicadas de modo mais geral.

3.2.1 O neurônio

O primeiro passo é implementar a classe *Neuron* para representar cada neurônio. Esta é uma classe de entidade bem simples, contendo apenas um construtor, e o método *output* que recebe os valores de entrada para esse neurônio e faz o cálculo da transformação linear e a seguir aplica e retorna o valor da função de ativação utilizada, que é passada como parâmetro ao construtor da classe. A listagem 3.1 abaixo mostra este método, em conjunto com o construtor da classe.

```
1 class Neuron:
2     def __init__(self, weights, bias, learning_rate, ativacao, der_ativacao):
3         '''(list[float], float, float, Callable, Callable) -> None'''
4         ...
5
6     def output(self, inputs):
7         '''(list[float]) -> float'''
8         self.output_cache = np.dot(inputs, self.weights) + self.bias
9         return self.ativacao(self.output_cache)
```

Programa 3.1: *Trecho da classe Neuron*

A função *np.dot* da biblioteca *Numpy* é utilizada para calcular o produto escalar entre os valores de entrada e os pesos desse neurônio. O valor da transformação linear é armazenado num atributo de classe antes da aplicação da função de ativação, pois será utilizado mais à frente durante o treinamento da rede.

3.2.2 A função de ativação

A função de ativação possui o papel de ativar ou não a saída de um neurônio, conforme visto no capítulo anterior, e a forma com que essa ativação ocorre é definida pela função utilizada. Aqui o termo *ativar* significa que a função irá retornar um valor mais próximo de 1 enquanto que uma não-ativação retornará um valor mais próximo de 0. Essa é uma restrição para a função de ativação para a camada de saída, que será sempre da forma:

$$f : \mathbb{R} \longrightarrow [0, 1]$$

No caso do neurônio biológico, quando dizemos que ele ativa/transmite ou não o sinal elétrico que chegou até ele, é como se ele *retornasse* apenas 0 ou 1. De fato, poderíamos até usar uma função similar a essa em alguma camada de nossa rede artificial, e este tipo de função escada tem a seguinte definição:

$$f(x) = \begin{cases} 1 & \text{se} \quad x \geq 0 \\ 0 & \text{se} \quad x < 0 \end{cases}$$

A utilização dessa função de ativação, conforme nos diz Grus (Grus, 2016), faria com que um neurônio fizesse simplesmente a distinção entre espaços separados pelo hiperplano de pontos tal que $< w.x > + b = 0$, ou seja, o hiperplano definido pelos pontos de entrada cuja transformação linear resultasse em zero.

Esta função é claramente não contínua e portanto não diferenciável, e precisamos de uma função de ativação que o seja, uma vez que algumas das equações da otimização que calculamos anteriormente, dependem da expressão de sua derivada. É por essa razão, que Grus (Grus, 2016) nos explica que passou-se a considerar uma aproximação suave da função escada, essa aproximação é a função *sigmoid*:

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

Ela retorna valores somente no intervalo $[0, 1]$, igualmente à função escada, sua inspiração. Essa característica, no entanto, não é uma restrição para as camadas ocultas da mesma forma que é para a camada de saída, uma vez apenas a camada de saída será comparada com valores esperados no intervalo $[0, 1]$. A sua derivada pode ser facilmente calculada, e sua expressão simplifica-se como:

$$\sigma^{'}(x) = \sigma(x)(1 - \sigma(x))$$

Podemos comparar o comportamento dessas funções de ativação no gráfico presente na imagem 3.2 abaixo. A seguir, na listagem 3.2, um trecho do script `util.py` com a implementação da função *sigmoid* e de sua derivada.

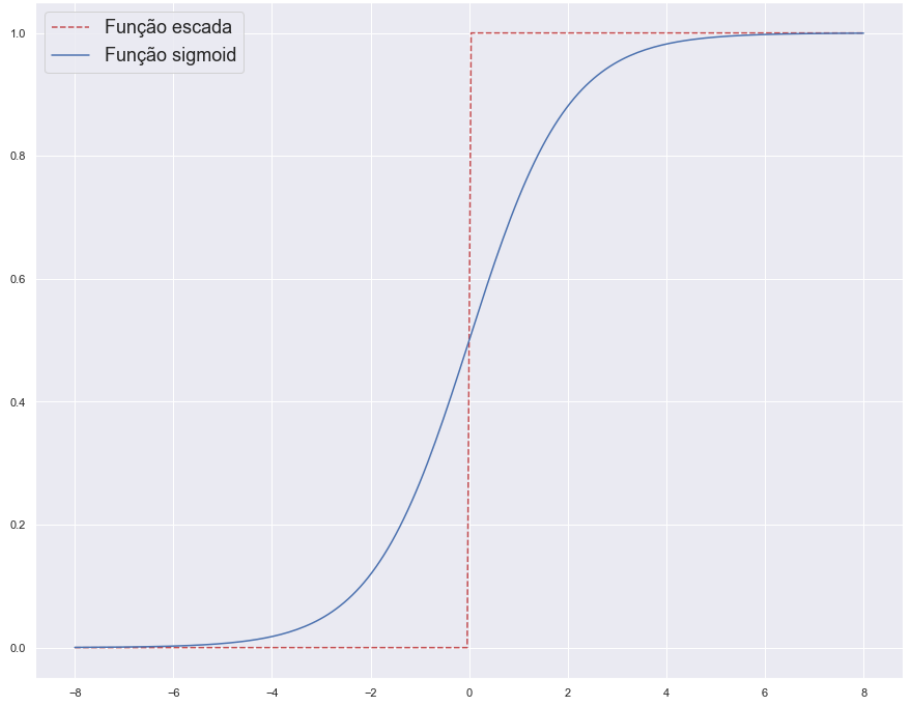

Figura 3.2: *Comparação entre as funções de ativação do tipo escada e a* sigmoid.

```
1 def sigmoid(x):
2     '''(float) -> float'''
3     return 1.0 / (1.0 + np.exp(-x))
4
5 def der_sigmoid(x):
6     '''(float) -> float'''
7     sig = sigmoid(x)
8     return sig * (1 - sig)
```

Programa 3.2: *Trecho do script util.py*

Com a popularização das redes neurais, várias outras funções de ativação foram criadas para ativarem as camadas ocultas do treinamento, devido aos problemas que podem acontecer ao se utilizar função *sigmoid*. Podemos identificar um desses problemas analisando seu gráfico. Vemos que ela se aproxima de 1, que é a ativação máxima, rapidamente a partir de $x > 4$, e aproxima-se simetricamente de zero com valores a partir de $x < -4$.

Como o método do gradiente tenta ajustar os valores dos pesos a partir dos valores de saída e esses ajustes dependem da derivada da função de ativação, temos que levar em conta o comportamento da derivada da função *sigmoid*, o qual podemos observar a partir de seu gráfico na figura 3.3.

Como podemos ver, a derivada retorna valores sempre menores do que 1, e além disso aproxima-se de 0 tão rapidamente quanto a *sigmoid* aproxima-se de 1. Isso faz com que atualizações para valores de saída que já estão muito altos não sejam efetivos para diminuí-los, pois justamente nessa região a derivada está muito próxima de 0. Essa é a desvantagem da função *sigmoid*.

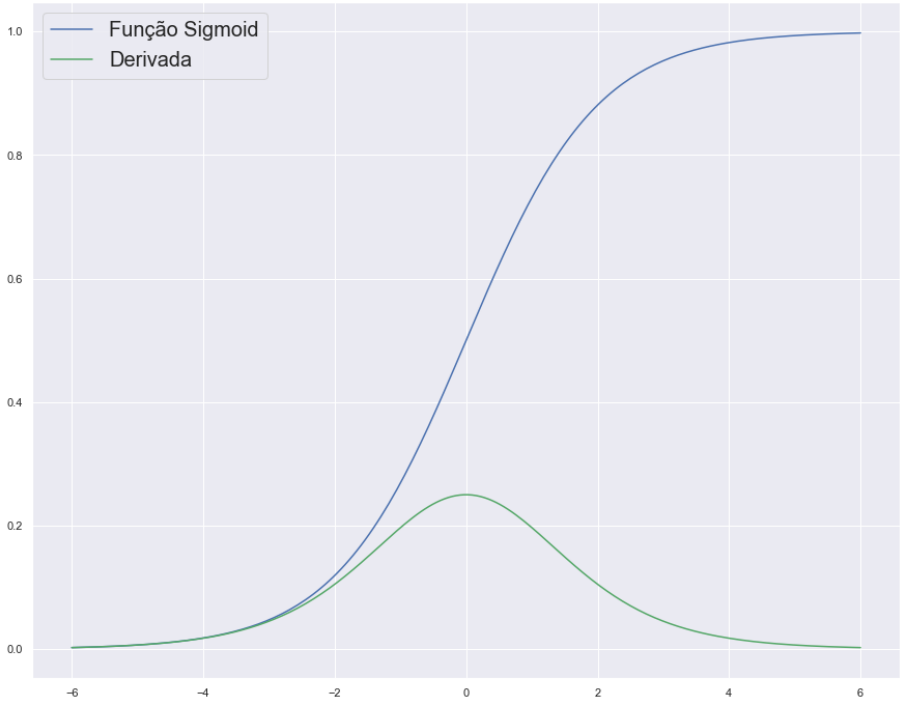

Figura 3.3: *Gráficos da função* sigmoid *e sua derivada.*

Um problema relacionado a este é que se a regra da cadeia em 3.1 e 3.2, com as derivadas da função de ativação dadas em 3.4 multiplicadas através das várias camadas, pode resultar num número muito grande, se todas as derivadas resultarem em valores maiores do que 0, ou resultar num número muito próximo de 0 se todas as derivadas forem menores do que 0. Isto faz com que atualizações dadas pelo gradiente sejam instáveis. Este é o problema descrito por Matheus Facure (Facure, 2017a) e nomeado como problema do gradiente explodindo/desvanecendo. (*exploding/vanishing gradient problem*).

Assim, conforme nos diz Facure (Facure, 2017b), a utilização da função *sigmoid* não é mais recomendada em problemas que envolvam de redes neurais maiores, sendo bem comum o problema do gradiente explodindo, já que a derivada é sempre maior do que 0. Porém, ele também diz que alguns modelos probabilísticos de variáveis binárias, modelagem de problemas biológicos onde ela é uma aproximação mais plausível da ativação elétrica-biológica, e também alguns modelos não supervisionados de redes tem restrições que fazem com que seja não só desejável como também necessário o uso da função *sigmoid*.

A próxima função de ativação é a o tangente hiperbólico $\tanh(x)$, ela é similar à *sigmoid* e pode ser escrita em função dela. Ela retorna valores no intervalo $[-1, 1]$ mas sua derivada retorna valores mais próximos de 1, chegando ao valor máximo de 1 quando $x = 0$. A expressão em função da função *sigmoid* e a derivada da função tangente hiperbólica são dadas por:

$$tanh(x) = 2\sigma(2x) - 1 \qquad tanh'(x) = 1 - tanh^2(x)$$

Na figura 3.4 podemos ver o gráfico da função e de sua derivada, a partir do que

podemos notar como a derivada da *tanh* retorna valores maiores do que a derivada da função *sigmoid*.

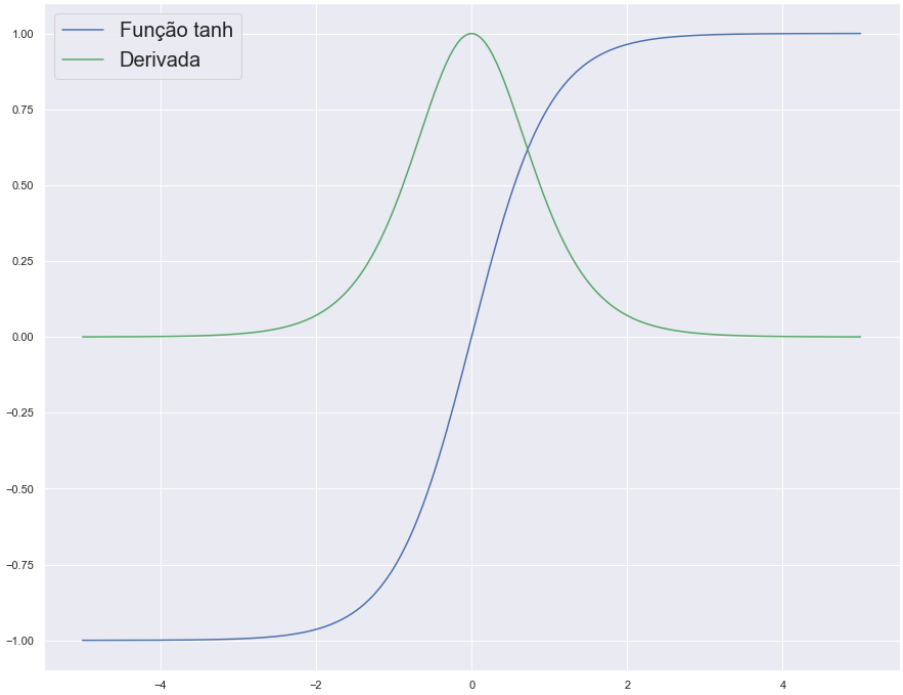

Figura 3.4: *Gráficos da função tangente hiperbólico e sua derivada.*

O próximo avanço é conseguido com a função de ativação linear retificada (**RELU**). Essa função é quase a função identidade, exceto que na região negativa do domínio ela vale identicamente 0. Ela não é derivável no ponto $x = 0$, mas podemos estender a definição fixando seu valor em 1 nesse ponto. Sua definição e de sua derivada estendida é dada por:

$$ReLU(x) = max\{0, x\} \qquad ReLU'(x) = \begin{cases} 1, & \text{se } x \geq 0 \\ 0, & \text{c.c.} \end{cases}$$

Podemos ver seus gráficos, na figura 3.5, a seguir. Usar essa função de ativação torna até mesmo a execução do código mais rápida, uma vez que não há cálculos matemáticos a serem feitos, apenas uma função de máximo que é trivial. Além disso, podemos notar que a derivada se mantém com o valor 1 constante enquanto o neurônio é ativado, sendo uma forma de tentar resolver o problema do gradiente explodindo/desvanecendo, além de agilizar o processo de treinamento.

Essa é a razão, conforme explica Facure (Facure, 2017b), dessa função ter contribuido para o recente aumento de popularidade das redes neurais. Adicionalmente, Bing Xu (Xu *et al.*, 2015) ressalta que outra vantagem das funções do tipo *RELU*, além de resolver o problema do gradiente explodindo/desvanescendo, é a de aumentar a velocidade da convergência do algoritmo de treinamento rumo a um mínimo da função de custos.

Uma desvantagem da função *RELU* é a chance de neurônios serem desativados

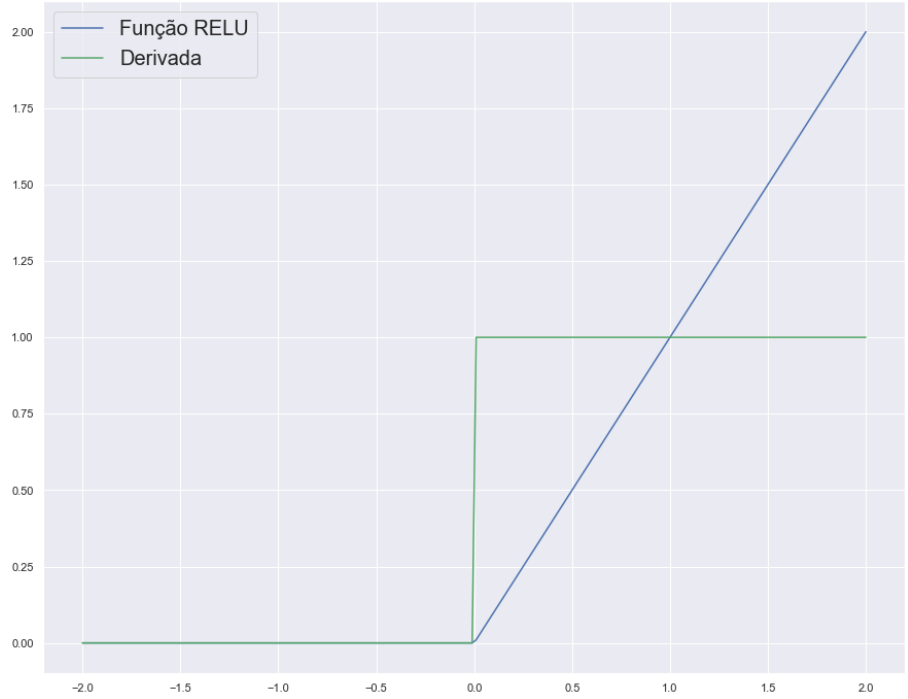

Figura 3.5: *Gráficos da função* RELU *e sua derivada.*

permanentemente, já que uma vez que ele zera, a função de ativação e sua derivada são ambos 0, de forma que ele nunca mais irá aumentar durante o treinamento, tornando-se neurônios *mortos*.

O próximo avanço foi dado pela função conhecida como *Leaky RELU*. Quase identica à *RELU*, exceto que na parte negativa do domínio ao invés de 0 a função retorna x/α, onde $\alpha \in (0, \infty)$. Isso já imediatamente corrige o problema dos neurônios desativados. A definição da função e de sua derivada, dada por Xu (Xu *et al.*, 2015), é:

$$LeakyReLU(x, \alpha) = \begin{cases} x, & \text{se } x \geq 0 \\ x/\alpha, & \text{c.c.} \end{cases} \qquad LeakyReLU'(x, \alpha) = \begin{cases} 1, & \text{se } x \geq 0 \\ \alpha, & \text{c.c.} \end{cases}$$

A partir dos resultados dos estudos feitos por Xu (Xu *et al.*, 2015), a função *Leaky RELU*, e suas variações, se saíram consistentemente melhores do que a *RELU* para as bases de dados de pequeno e médio portes. Além disso, ele testou a performance para diferentes valores de α, obtendo os melhores resultados com $\alpha = 5.5$. Este parâmetro é conhecido como **vazamento**, que dá o nome à função. Podemos ver o seu comportamento no gráfico da imagem 3.6.

Por fim, temos a função de unidade linear exponencial *ELU*, proposta por Djork-Arné Clevert (Clevert *et al.*, 2015), que é definida, com $\alpha > 0$, por:

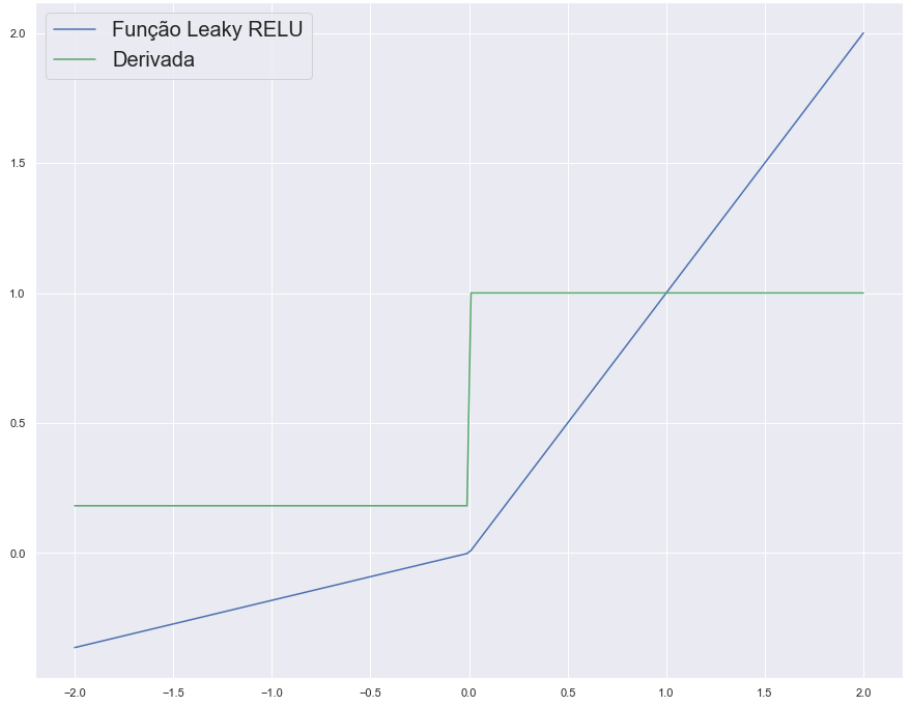

Figura 3.6: *Gráficos da função* Leaky RELU *e sua derivada.*

$$ELU(x,\alpha) = \begin{cases} x, & \text{se } x \geq 0 \\ \alpha(e^x - 1), & \text{c.c.} \end{cases} \qquad ELU'(x,\alpha) = \begin{cases} 1, & \text{se } x \geq 0 \\ ELU(x,\alpha) + \alpha, & \text{c.c.} \end{cases}$$

Em seu artigo, Clevert (Clevert *et al.*, 2015) utiliza o valor $\alpha = 1$, e com a função *ELU* conseguiu perfomances melhores, tanto de resultados mais corretos, quanto de velocidade de treinamento, em relação às funções *RELU* e *Leaky RELU* para as mesmas bases de dados avaliadas por XU (Xu *et al.*, 2015), mesmo com o uso da função exponencial em sua definição o que em teoria deveria diminuir a performance do treinamento. Podemos observar o comportamento dessa função e de sua derivada, com $\alpha = 1$ no gráfico mostrado na figura 3.7.

Testes feitos em condições similares por Facure (Facure, 2017b), mostram que essa diferença não é tão significativa em relação à *Leaky RELU*, mas que ambas, *Leaky RELU* e a *ELU* são sim melhores do que a original *RELU*, o que é consistente com o fato delas resolverem teoricamente as desvantagens dela. E são todas obviamente melhores escolhas do que a função *sigmoid*, em todos os estudos acima citados.

Na prática, podemos testar qual função de ativação irá performar melhor para o problema que queremos resolver. A abordagem mais comum, conforme descrita por Facure (Facure, 2017b) é utilizar a função *Leaky RELU* nas camadas ocultas, sendo o modo mais simples de obtermos bons resultados graças ao seu comportamento. Podemos avaliar a utilização das outras de acordo com o problema em questão, dado que algumas funções se saem melhor em alguns contextos específicos como é o caso da função *sigmoid*.

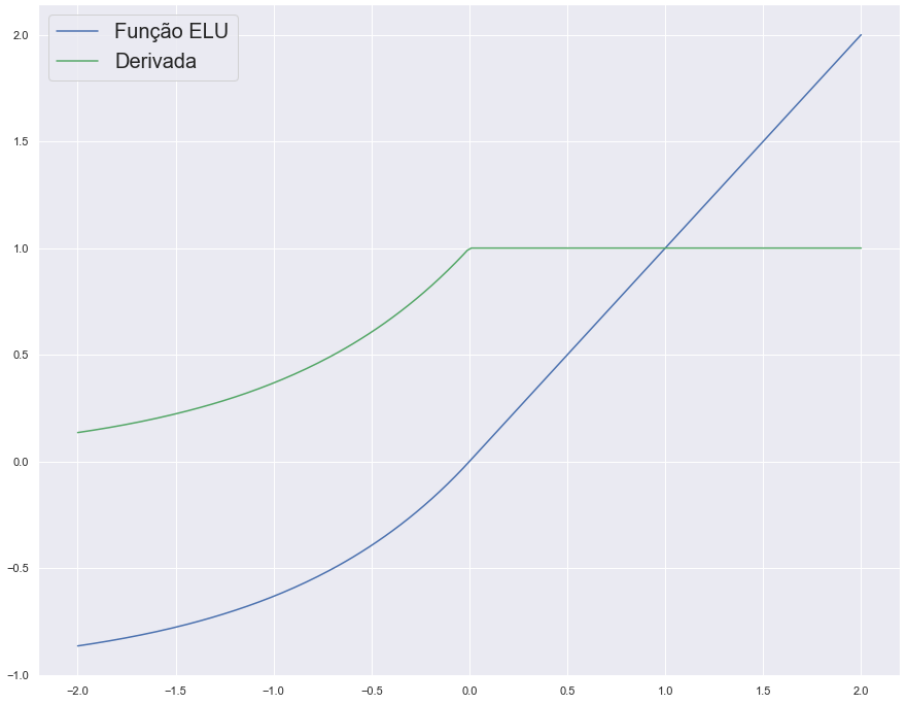

Figura 3.7: *Gráficos da função* ELU *e sua derivada.*

3.2.3 As camadas

A classe *Layer* representa uma camada de neurônios. Cada camada conecta-se com a sua camada anterior, com exceção da camada de entrada. Por essa razão, a rede perceptron possui um sentido único de conexão, que vai da entrada para a saída, passando por cada camada oculta. A classe é constituída de uma lista de objetos da classe *Neuron*, uma referência à camada anterior e uma lista para armazenar as saídas dos seus neurônios.

O construtor de *Layer* é responsável por inicializar seus neurônios. Nessa implementação todos os neurônios de uma camada irão usar a mesma função de ativação e a mesma taxa de aprendizagem. Além de receber esses parâmetros, o número de neurônios dessa camada, e a referência da camada anterior, o construtor inicializa os pesos de cada neurônio, lembrando que cada neurônio de uma camada possui a mesma quantidade de pesos do que a quantidade de neurônios da camada anterior, e também inicializa o viés de cada neurônio.

Na listagem 3.3 está o trecho do construtor que inicializa os pesos dos neurônios. Se a camada que estamos inicializando é a camada de entrada, então não criamos pesos e o viés, pois os valores de entrada serão utilizados diretamente como a saída dessa camada, este é o teste presente na linha 11 da listagem, pois a camada de entrada não possui referência a uma camada anterior já que ela é a primeira camada da rede.

A linha 12 de 3.3 inicializa os pesos dos neurônios da camada. Para fazer isso utiliza uma função que está definida no script `util.py`, que sortea números aleatórios para esses pesos seguindo uma distribuição normal de média 0 e desvio-padrão 0.3 e além disso *truncada* no intervalo $[-1, 1]$, para que nenhum peso esteja fora desse domínio e para que em média esse valor seja 0. O viés é inicializado com um valor constante, próximo de zero,

nesse caso com 0.01.

```
class Layer:
    def __init__(self, previous_layer, num_neurons, learning_rate,
                 ativacao=None, der_ativacao=None):
        '''(Layer, int, float, Callable, Callable) -> None
        Construtor da Camada de Neurônios
        '''
        ...
        for i in range(num_neurons):
            pesos = None
            bias = None
            if previous_layer is not None:
                pesos = normal_t.rvs(len(previous_layer.neurons))
                bias = 0.01

            neuron = Neuron(pesos, bias, learning_rate, ativacao, der_ativacao)
            self.neurons = np.append(self.neurons, neuron)
```

Programa 3.3: *Trecho da classe* Layer

Este procedimento é usado para tentar mitigar dois problemas que podem acontecer, conforme explicado por James Dellinger (Dellinger, 2019). Se inicializarmos os pesos com muitos números não tão próximos de 0, numa rede como muitos neurônios e muitas camadas, esses pesos podem somar-se rapidamente através das camadas, resultando em números com valores absolutos muito grandes na camada de saída o que pode prejudicar o treinamento e aprendizado da rede.

Se pelo contrário, inicializarmos todos os pesos com números muito próximos de 0, ocorre o problema oposto, os neurônios tem seus valores zerados, tornando-se *neurônios desativados*, que se tornam inúteis para o aprendizado já que serão ignorados durante o restante do treinamento.

Dellinger (Dellinger, 2019) distute esses problemas no contexto de redes bem grandes, com mais de 100 camadas, e exibe sua solução heurística que é utilizar uma distribuição normal (não-truncada) com média 0 e com desvio-padrão $\sqrt{2/n}$, sendo n o número de neurônios da camada anterior.

Para nossos fins didáticos, testei alguns desvios-padrão como 1, 0.3 e 0.1, em um dos exemplos que serão mostrados ainda nesse capítulo, e dentre eles, o valor 0.3 se saiu melhor sendo o suficiente para não explodir e nem desativar os neurônios da única camada oculta que foi usada na aplicação-exemplo em questão.

O desvio-padrão de 0.3 auxilia na tarefa de restringir os valores no intervalo $[-1, 1]$, sem que precisemos truncar muitos valores o que poderia aumentar a massa de probabilidade dos extremos -1 e 1, já que valores mais distantes da média do que 3 vezes o desvio-padrão são raramentes obtidos de uma distribuição normal.

Após inicializar cada neurônio, salvamos ele na lista de neurônios dessa camada, que é um dos atributos de classe discutidos no primeiro parágrafo. A próxima tarefa de uma camada é processar as entradas recebidas e retornar as saídas. Podemos observar esse comportamento na listagem 3.4.

```
def outputs(self, inputs):
        '''(list[float]) -> list[float]
        Armazena em cache as saidas dos neuronios e a retornam
        Se for uma camada de entrada, usa elas diretamente
        '''
        if self.previous_layer is None:
            self.output_cache = inputs
        else:
            self.output_cache = np.array([n.output(inputs) for n in self.neurons])
        return self.output_cache
```

Programa 3.4: *Trecho da classe* Layer

A camada de entrada não processa os dados, usando-os diretamente. As demais camadas devem processar cada neurônio, usando seu próprio método de processamento, aplicando a transformação linear e em seguida a função de ativação. O resultado é armazenado numa lista *numpy* que é o atributo de classe `output_cache`, que armazena as saídas dessa camada para uso posterior; por fim, a lista das saídas da camada é retornada.

A última tarefa da classe *Layer* é calcular os termos Δ definidos pelas equações 3.15 e 3.16, que definem respectivamente o cálculo que é feito se estamos calculando as derivadas para as camadas ocultas e o cálculo feito para a camada de saída. As duas versões são exibidas na listagem 3.5 abaixo.

```
def calcular_delta_camada_de_saida(self, expected):
        '''(list[float]) -> None'''
        for i, neuron in np.ndenumerate(self.neurons):
            der_cost = expected[i[0]] - self.output_cache[i]
            neuron.delta = neuron.der_ativacao(neuron.output_cache) * der_cost

    def calcular_delta_camada_oculta(self, next_layer):
        '''(Layer) -> None'''
        for i, neuron in np.ndenumerate(self.neurons):
            next_weights = np.array([n.weights[i[0]] for n in next_layer.neurons])
            next_deltas = np.array([n.delta for n in next_layer.neurons])
            der_cost = np.dot(next_weights, next_deltas)
            neuron.delta = neuron.der_ativacao(neuron.output_cache) * der_cost
```

Programa 3.5: *Trecho da classe* Layer

Os algoritmos são as traduções quase literais das equações 3.15 e 3.16. Podemos ver a natureza recursiva da regra da cadeia nas 2 linhas finais, onde usamos os deltas calculados da próxima camada para calcular os deltas da camada atual. O caso base é a função que calcula o delta da camada de saída. A lógica que orquestra essa recursão está implementada na próxima classe.

3.2.4 A rede

A classe *Network* representa a rede neural como um todo. Ela armazena uma lista de camadas, ou seja, objetos do tipo *Layer*, a partir dos parâmetros que recebe em seu construtor, que são a estrutura da rede que será criada, que é um vetor de inteiros que

representam as quantidades de neurônios para cada camada. Além disso, recebe a taxa de aprendizado que será utilizada em toda a rede, nessa versão, e quais as funções de ativação que serão utilizadas nas camadas ocultas e na camada de saída.

A listagem 3.6 exibe o trecho do construtor que cria cada camada e insere na lista de camadas do objeto da classe atual. A camada de entrada não possui camada anterior, nem função de ativação. Além disso, a camada de saída pode utilizar uma função de ativação diferente daquela utilizada pelas camadas ocultas, que usarão a mesma.

```
1  class Network:
2      def __init__(self, layer_structure, taxa, ativacoes):
3          '''(list[int], float, Tuple[Callable]) -> None'''
4          ...
5          self.layers = np.array([], dtype=np.float64)
6          self.estrutura = layer_structure
7
8          # camada de entrada
9          input_layer = Layer(None, self.estrutura[0], taxa)
10         self.layers = np.append(self.layers, input_layer)
11
12         # camadas oculta(s)
13         for previous, qtd_neurons in np.ndenumerate(self.estrutura[1::l]):
14             next_layer = Layer(self.layers[previous[0]], qtd_neurons, taxa,
15                                ativacoes[0], ativacoes[1])
16             self.layers = np.append(self.layers, next_layer)
17
18         # camada de saída
19         output_layer = Layer(self.layers[-1], self.estrutura[-1], taxa,
20                              ativacoes[2], ativacoes[3])
21         self.layers = np.append(self.layers, output_layer)
```

Programa 3.6: *Trecho da classe* Network

A primeira tarefa da classe Network é o de processar entradas, fazendo elas atravessarem a rede, camada a camada, até a camada de saída, e retornar as saídas obtidas. É o processo de *feedforward* explicado no início do capítulo. Sua implementação mesmo para o caso geral de multi-camadas é bem simples, conforme exibido na listagem 3.7 abaixo.

```
1  def feedforward(self, entrada):
2      '''(list[float]) -> list[float]'''
3      ...
4      saida = self.layers[0].outputs(entrada)
5      for i in range(1, len(self.layers)):
6          saida = self.layers[i].outputs(saida)
7      return saida
```

Programa 3.7: *Trecho da classe* Network

A próxima tarefa é treinar a rede, passando uma lista de entradas e saídas esperadas, realizando o procedimento de *backpropagate* para atualizar os pesos e viéses dos neurônios de cada camada, tudo isso em sequência, para cada entrada fornecida. É o que está literalmente implementado na listagem 3.8 abaixo.

```
def train(self, entradas, saidas_reais):
    '''(list[list[floats]], list[list[floats]]) -> None'''
    ...
    for i, xs in enumerate(entradas):
        ys = saidas_reais[i]
        _ = self.feedforward(xs)
        self.backpropagate(ys)
        self.update_weights()
        self.update_bias()
  return saida
```

Programa 3.8: *Trecho da classe* Network

Cada chamada à função *train* significa o procedimento de treinamento sendo executado uma única vez. Cada vez que a rede é treinada dizemos que ela avançou em uma **época** de treinamento. O treinamento consiste em primeiramente executar o *feedforward* para uma entrada, para que as camadas possam armazenar as saídas correspondentes aos valores atuais de seus parâmetros, os pesos e viéses, assim como as saídas em seus atributos *output_cache*, que serão usados pelo método *backpropagate* a seguir, de acordo com as equações que derivamos para o processo de treinamento.

O funcionamento do método *backpropagate* pode ser visto na listagem 3.9 a seguir. Tudo o que ele faz aqui é calcular os deltas das camadas, na ordem correta, começando pela camada de saída, e depois percorrendo as demais camadas do final para o início da rede, fazendo a chamada para cada objeto da classe *Layer* que constitui a rede.

```
def backpropagate(self, saidas_reais):
    '''(list[float]) -> None
    Calcula as mudanças em cada neurônio com base nos erros da saída
    em comparação com a saída esperada
    '''
    # calcula delta para os neurônios da camada de saída
    last_layer = len(self.layers) - 1
    self.layers[last_layer].calcular_delta_camada_de_saida(saidas_reais)

    # calcula delta para as camadas ocultas, da saída para o início da rede
    for l in range(last_layer - 1, 0, -1):
        self.layers[l].calcular_delta_camada_oculta(self.layers[l + 1])
```

Programa 3.9: *Trecho da classe* Network

A seguir, atualiza-se os pesos e os viéses com os métodos correspondentes, que podem ser visualizados na listagem 3.10. Como os valores são todos armazenados nos atributos de estado dos neurônios e das camadas, implementamos diretamente as contas das equações que obtemos para o método do gradiente e da regra da cadeia da retroprogação. Dessa forma, o que fazemos na classe *Network* é basicamente traduzir a matemática para a sintaxe da linguagem Python.

```
def update_weights(self):
    '''(None) -> None'''
    ...
    for layer in self.layers[1:]: # pula a camada de entrada
        for neuron in layer.neurons:
            for w in range(len(neuron.weights)):
                neuron.weights[w,] = neuron.weights[w,] + (neuron.learning_rate
                    * (layer.previous_layer.output_cache[w]) * neuron.delta)

def update_bias(self):
    '''(None) -> None'''
    ...
    for layer in self.layers[1:]: # pula a camada de entrada
        for neuron in layer.neurons:
            neuron.bias = neuron.bias + neuron.learning_rate * neuron.delta
```

Programa 3.10: *Trecho da classe* Network

A próxima responsabilidade da classe *Network*, uma vez que já foi treinada, é fazer a previsão de classes de novos dados de entrada. Isto é feito pelo método mostrado na listagem 3.11. Isto significa simplesmente processar as entradas fornecidas pelo método *feedforward*, que usará os parâmetros que foram ajustados anteriormente para fazer os cálculos.

```
def predict(self, entradas, interpretar):
    '''(list[list[floats]], Callable) -> list[list[floats]]
    '''
    self.previsoes = np.array([], dtype=np.float64)
    for entrada in entradas:
        self.previsoes = np.append(self.previsoes, interpretar(self.feedforward(entrada)))
    return self.previsoes.reshape(-1, 1)
```

Programa 3.11: *Trecho da classe* Network

Ao final, os dados da última camada são uma lista, isto é, um vetor de valores reais, que são interpretados por uma função que é passada por parâmetro que identifica a qual classe, previamente definida, pertence esse vetor de saída. A lógica dessa interpretação é externa à classe *Network* e será vista mais adiante. A lista de classes preditas em formato *numpy* é retornada.

A última responsabilidade dessa classe é calcular uma métrica de avaliação para esta rede, que irá servir de avaliação do quão boa a rede é para classificar os dados utilizados. A métrica mais simples a ser utilizada é a **acurácia** da previsão. Ela basicamente mede a proporção de classificações corretas dentre todas as classificações realizadas para um conjunto de dados. Essa lógica bem simples é implementada na listagem 3.12 abaixo.

```
def validate(self, esperados):
    '''(list[list[floats]], list[list[floats]], Callable) -> float
    '''
    ...
    corretos = 0
    for y_pred, esperado in zip(self.previsoes, esperados):
        if y_pred == esperado:
```

```
8                 corretos += 1
9         acuracia = corretos / len(self.previsoes)
10        return acuracia
```

Programa 3.12: *Trecho da classe* Network

Nota-se que ela utiliza as previsões salvas no atributo de classe que é atualizado toda vez que executamos o método *predict*, acima. As classes esperadas são passadas como parâmetro, uma vez que estamos lidando no *Perceptron* com uma aprendizagem supervisionada. Dessa forma, ao treinarmos a rede utilizamos um conjunto de dados para os quais já sabemos as classes, e ainda dividimos esse conjunto em duas partes, as quais chamamos de **conjunto de treino** e de **conjunto de teste ou validação**.

Treinamos a rede com o conjunto de treino, que deve sempre ser a maior parte de nossa partição, uma vez que é a partir dele que iremos usar o *backpropagate* para aproximar as saídas da rede às saídas esperadas contidas no conjunto. Géron (Géron, 2019) cita que tipicamente escolhemos aleatoriamente 20% dos dados como nosso conjunto de teste, ficando o restante como o conjunto de treino. A seguir, podemos calcular a acurácia da classificação do conjunto de treino, e a seguir prever e medir a acurácia do conjunto de teste para comparar os resultados.

Naturalmente a acurácia para o conjunto de teste tende a ser menor, mas não pode ser muito menor, senão dizemos que nossa rede sofre de um problema de classificação conhecido como **overfitting**, ou sobreajuste, o que significa que ela está boa para lidar com o conjunto com o qual foi treinado, o que era esperado dado que foi construída para isso, mas sofre para classificar dados novos, com os quais não foi treinada, e não é isso o que queremos.

O que queremos é justamente o contrário, que nossa rede, ou seja, nosso algoritmo de aprendizagem seja bom em **generalizar** os dados de entrada que fornecemos a ele. Anas Al-Masri (Al-Masri, 2019) define o termo generalização como a habilidade do modelo para fornecer saídas sensíveis para conjuntos de entradas que ele nunca viu antes.

Dentre as várias técnicas existentes para melhorar a generalização/prevenir o *overfitting* estão a inicialização dos pesos dos neurônios segundo uma distribuição normal de média zero, procedimento explicado anteriormente. Outra técnica é denominada de **dropout**, a qual Amar Budhiraja (Budhiraja, 2016) define como ignorar alguns neurônios, escolhidos aleatoriamente, durante o treinamento da rede, isto é, não calculamos deltas durante uma execução do *backpropagate* e nem usamos seu valor em consideração quando processamos uma entrada na rede como o *feedforward*.

É como se deliberadamente "desativássemos" alguns neurônios durante a fase de treinamento, de forma a diminuir o sobreajuste aos dados de treino, uma vez que, segundo Budhiraja (Budhiraja, 2016) esse procedimento previne que todos os neurônios se tornem dependentes um dos outros, isto é, que a derivada (da função de custo) de cada um se torne dependente das derivadas de todos os outros, criando uma mútua dependência geral para a diminuição do erro, diminuindo assim o poder individual que cada um teria se fossem ajustados de forma mais independente em função de sua contribuição para o custo total da rede.

Em nosso *Perceptron* didático optei por não implementar o procedimento de *dropout*, já que o objetivo dessa versão aqui demonstrada não é fornecer um modelo que seja utilizado em problemas reais, mas apenas didáticos. Em contrapartida essa técnica está presente e pode ser utilizada nas bibliotecas de redes neurais que usaremos na parte prática do trabalho.

3.2.5 A classe *Perceptron*

A última classe implementada, não foi baseada em um exemplo, mas criada a partir da necessidade de encapsular o comportamento da rede, para facilitar os testes do seu funcionamento que iremos realizar. Já discutimos sobre as diferentes taxas de aprendizagem para o gradiente descendente, discutimos sobre as diferentes funções de ativação que podem ser utilizadas nas redes neurais, assim como a sua topologia no que se refere apenas à quantidade de neurônios e a quantidade de camadas de neurônios.

Esses são basicamente os atributos que teremos que ajustar de acordo com a necessidade que os dados utilizados em nosso aprendizado irão criar. Dessa forma criamos a classe *Perceptron* que possui um construtor que irá lidar com a seleção dessas opções. Na listagem 3.13 está apenas a definição de seu construtor e a documentação explicativa.

```
1  class Perceptron():
2      def __init__(self, N=[1], M=50, ativacao="l_relu", taxa=0.001, debug=0):
3          '''(None, str, list[int], int, float, float, str, float) -> None
4          Construtor da minha classe Perceptron
5          Parâmetros da classe:
6              *N: quantidade de neurônios da camada oculta, podendo ser
7                  especificada um vetor de várias camadas ocultas ou apenas
8                  uma.
9              *M: quantidade de treinamentos desejada, denominado de número
10                 de "épocas" da rede, o valor padrão é50;
11             *ativacao: escolha de uma das funções de ativação disponíveis
12                 para a(s) camada(s) oculta(s).
13             *taxa: taxa de aprendizagem, padrão de 0.001
14             *debug: flag para exibição de parâmetros durante o treinamento
15         '''
16         ...
```

Programa 3.13: *Trecho da classe* Perceptron

No construtor é feita uma seleção dentre as funções de ativação existentes no script *util.py*, de forma que só precisamos passar um texto com o nome da função, que o construtor irá selecionar a função e sua derivada para posteriormente informá-las à classe *Network*. Deixamos por padrão a escolha da função de ativação *Leaky RELU*, de acordo com a orientação geral dada por Facure (Facure, 2017b).

O parâmetro *M* define a quantidade inicial padrão de **épocas** que serão treinadas. Uma época corresponde a uma passagem do conjunto de treino pelo *feedforward* e a seguir pelo *backpropagation*, ou seja, configura um único ajuste dos parâmetros através de nosso algoritmo de treinamento.

A arquitetura de camadas padrão definida pelo parâmetro *N* não tem de fato esse papel, serve apenas para uma validação existente no construtor para que não permita a passagem

de uma quantidade ≤ 0 de camadas ou de neurônios. Se quiséssemos uma arquitetura de 3 camadas com 4, 5, e 6 neurônios cada, por exemplo, então o parâmetro passado durante a criação de um objeto *Perceptron* deveria ser $N = [4, 5, 6]$.

Outra tarefa do construtor é criar um objeto da classe *OneHotEncoder*[1] da biblioteca *scikit-learn*[2]. Essa é uma das mais famosas e mais utilizadas bibliotecas da linguagem Python para tarefas de aprendizado de máquina. É a biblioteca utilizada pela maioria dos autores dos livros-textos da área de ciência de dados, como por exemplo Géron (Géron, 2019) e Grus (Grus, 2016).

Esse objeto codificador (*encoder*) é salvo como um atributo de classe: `self._enc`, e será utilizado no método de treinamento para criar automaticamente as classes numéricas a partir das classes fornecidas pelos conjuntos de treinamento e validação. Essas classes numéricas representam cada classe no formato de um vetor com todos os componentes zerados exceto um, o que identifica unicamente as classes.

Suponha, por exemplo, que estamos treinando um conjunto de fotos que possuem as classes *cachorro*, *gato* e *rato*. A função codificadora poderá transformar a palavra *cachorro* no vetor $[1, 0, 0]$, *gato* no vetor $[0, 1, 0]$ e *rato* no vetor $[0, 0, 1]$. De forma que estes serão os valores esperados para os 3 neurônios de saída que nossa rede obrigatoriamente deverá ter (número de classes = número de neurônios de saída). A ordem dessa codificação é irrelevante, sendo gerenciada internamente pela classe *OneHotEncoder*.

A seguir, no método utilizado para treinar a rede, que recebe os dados de entrada e as classes esperadas correspondentes a cada entrada, o primeiro passo é fazer essa codificação. O primeiro trecho do método `treinar` está na listagem 3.14.

```
1  def treinar(self, x_train, y_train, M=0):
2      '''(np.array, np.array, int) -> None
3      Processo de treinamento da rede neural
4      1- Tratar os dados, obtendo as classes das respostas
5      2- Treinar um número M de épocas
6      3- Armazenar no objeto o estado final da rede,
7      com os pesos e vieses ajustados pelo treinamento
8      '''
9      # onehotencoder extrai as classes únicas já ordenadas alfabeticamente
10     y_encoded = self._enc.fit_transform(y_train)
11     classes = self._enc.categories_[0]
```

Programa 3.14: *Trecho da classe* Perceptron

Nessa classe e nos exemplos subsequentes neste trabalho, sempre nomeio o conjunto dos dados de treino de `x_train` e `y_train`, e do conjunto de teste de `x_test` e `y_test`. A letra x indica que é o conjunto de dados de entrada e y indica as classes esperadas de classificação. No trecho acima usamos a classe *OneHotEncoder* para transformar quaisquer formatos que as classes forem informadas nos vetores numéricos que serão os valores esperados da saída da rede.

O próximo trecho de código irá se encarregar de criar a estrutura geral da rede, ao final

[1] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html
[2] https://scikit-learn.org/stable/index.html

criando um objeto da classe *Network* e salvando-o como um atributo de classe. É o que está presente na listagem 3.15. Os dados de entrada são esperados no formato de lista do tipo *numpy*, dessa forma a primeira linha do trecho obtém a quantidade de características (*features*) dos dados de entrada, ou seja, das variáveis explicativas do modelo, através do método `shape(1)` do tipo *numpy*, essa será a quantidade de neurônios da camada de entrada da rede, um para cada característica.

Usando o exemplo das fotos de animais, os pixels da foto seriam as variáveis explicativas para um modelo de classificação, assumindo que todas as fotos possuem a mesma quantidade de pixels e que cada pixel possui apenas um valor de intensidade de cinza, ou outra cor única qualquer. Se utilizássemos fotos coloridas, com por exemplo, as intensidades de 3 cores diferentes por pixel, então o número de variáveis explicativas de nosso modelo seria $3n$, sendo n o número de pixels da foto.

```
1  neurons_in = x_train.shape[1]
2
3  if self.network is None:
4      neurons_out = len(self.classes)
5      for i in range(len(self.N)):
6          if len(self.N) == 1 and self.N[0] < neurons_out:
7              self.N[0] = min(int(np.ceil(neurons_in*2/3 + neurons_out)), neurons_in)
8
9      rede = []
10     rede.append(neurons_in)
11     for hidden in self.N:
12         rede.append(hidden)
13     rede.append(neurons_out)
14
15     ativacoes = (self.ativacao, self.der_ativacao,
16                  self.ativacao_saida, self.der_ativacao_saida)
17
18     self.network = Network(np.array(rede), self.taxa, ativacoes)
```

Programa 3.15: *Trecho da classe* Perceptron

A seguir, o método irá criar a estrutura da rede, se essa é a primeira vez que o objeto estiver sendo utilizado para o treinamento, do contrário ele irá realizar outras M épocas de treinamento, a partir dos dados existentes na rede.

Obtém-se a quantidade de neurônios para a camada de saída a partir da quantidade de classes identificadas. A seguir ele utiliza as quantidades de neurônios para as camadas ocultas se estas foram previamente informadas manualmente durante a criação da classe, ou então, é feito um cálculo, para que seja utilizada uma quantidade apropriada de neurônios para a primeira camada oculta, que pode ser a única camada oculta por padrão.

Essa quantidade apropriada é definida por Jeff Heaton (Heaton, 2017) como sendo 2/3 da quantidade de neurônios de entrada mais a quantidade de neurônios de saída. Esta é uma das 'regras de ouro' que ele descobriu empiricamente pois se mostraram as regras mais gerais para garantir o bom funcionamento da rede em vários casos. Outra regrinha que ele encontrou, que é mais geral mas que incluí esta é que a quantidade de neurônios de uma camada oculta única deve ser tal que seja menor que a quantidade de neurônios de entrada mas maior que a quantidade de neurônios de saída.

Naturalmente o treinamento aceita quaisquer número de camadas ocultas e de neurônios em cada camada, apenas não há garantias de que o treinamento irá suceder sem ocorrer o problema do gradiente explodindo/desaparecendo. Mesmo a utilização das regrinhas de Heaton (Heaton, 2017) não asseguram esse sucesso do treinamento, apenas testes com outras quantidades de neurônios e camadas, e com outras funções de ativação e taxas de aprendizado que poderão resultar eventualmente num treinamento bem sucedido, caso essas opções-padrão não sejam suficientes.

Essa é uma dificuldade inerente das redes neurais, ainda mais quando tenta lidar com conjuntos de dados muito grandes e com um grande número de variáveis explicativas. Essa é uma das razões principais para ser preferível a utilização de uma biblioteca já consolidade e com muitos anos de desenvolvimento e ajustes por muitos desenvolvedores e cientistas de dados ao redor do mundo.

O próximo trecho, na listagem 3.16 é o trecho principal deste método, é o treinamento *backpropagate* feito um número M de épocas. São passados como parâmetros os dados de entrada, e as classes esperadas já codificadas. Ao final desse treinamento, a rede está salva no atributo de classe `self.network` com os parâmetros já ajustados e prontos para serem utilizados para validação e previsão de novos dados.

```
1 for _ in self.tqdm(range(self.M)):
2     self.network.train(x_train, y_encoded)
```

Programa 3.16: *Trecho da classe* Perceptron

O próximo método realiza a previsão das classes a partir dos dados informados, simplesmente utilizando a função da classe *Network* criada para isso. É o que está na listagem 3.17, abaixo.

```
1 def prever(self, X, interpretar=None):
2     '''(np.array, Callable) -> np.array'''
3     ...
4     if interpretar is None:
5         return self.network.predict(X, self.reinterpretar_saidas)
6     return self.network.predict(X, interpretar)
```

Programa 3.17: *Trecho da classe* Perceptron

Por padrão a função que irá interpretar os neurônios de saída, convertendo-os em uma classe, foi criada da forma como será mostrada na listagem 3.18, a seguir.

```
1 def reinterpretar_saidas(self, saidas):
2     '''(array) -> np.array
3     '''
4     maximo = max(saidas)
5     saida = np.array([int(x == maximo) for x in saidas])
6     return self._enc.inverse_transform(saida.reshape(1, -1))
```

Programa 3.18: *Trecho da classe* Perceptron

Essa função realiza a interpretação padrão dos neurônios de saída, primeiramente eles são convertidos em vetores com identificação única, ou seja, no formato $[0, \ldots, 0, 1, 0, \ldots, 0]$,

sendo que a posição que irá receber 1 é aquela que tiver originalmente o valor máximo, ou seja, mais distante de 0, e o restante será convertido em zeros. Dessa forma, basta utilizarmos a decodificação inversa do objeto *OneHotEncoder*, que está salvo no atributo de classe, e daí obtemos qual a classe mais provável à qual pertence o dado de entrada que foi processado pela rede.

Isto nos mostra que uma possível interpretação dos neurônios de saída é que cada um possui uma probabilidade de que o dado pertença àquela classe indexada na mesma posição a qual esse neurônio está na camada de saída. Se usarmos como exemplo nosso modelo fictício dos animais, ao processar uma foto, a rede iria devolver os valores dos neurônios de saída, por exemplo, como o seguinte vetor: $[0.002, 0.976, 0.013]$. Pode-se dizer que há uma probabilidade maior de que essa foto pertença à segunda classe, que digamos ser a classe *Gatos*, por exemplo.

O que a função `reinterpretar_saidas` faz é converter esse vetor de saídas da rede no vetor $[0, 1, 0]$, que agora está no formato das classes numéricas geradas por nosso objeto codificador. Dessa forma, executar uma decoficação com esse mesmo objeto, que havia originalmente codificado a palavra *Gato* no vetor $[0, 1, 0]$, irá fazer a operação inversa, retornando a palavra *Gato* como sendo a classe mais provável para a foto processada.

Alternativamente podemos utilizar outra função de interpretação dos dados de saída, devendo ser informada diretamente por referência para o método `prever`.

Por razões didáticas, ou mesmo em casos em que não queremos obter classes diretamente a partir dos neurônios de saída, mas queremos observar diretamente os valores calculados pela rede sem interpretação, criei um método que realiza apenas o *feedforward* para uma lista de entradas fornecidas como parâmetro. É o que vemos na listagem 3.19, a seguir.

```
1 def processar(self, X):
2     '''(np.array) -> np.array'''
3     ...
4     saidas = []
5     for x in X:
6         saidas.append(self.network.feedforward(x))
7     return np.array(saidas)
```

Programa 3.19: *Trecho da classe* Perceptron

Por fim, podemos querer observar qual o erro, ou custo, que nossa rede está produzindo para um dado par de conjuntos de entrada e saídas esperadas. Para isso criei o método `funcao_erro` exibido na listagem 3.20, a seguir.

```
1 def funcao_erro(self, X, Y):
2     '''(np.array, np.array) -> float'''
3     ...
4     y_encoded = self._enc.fit_transform(Y)
5     return self.network.mse(X, y_encoded)
```

Programa 3.20: *Trecho da classe* Perceptron

Ele utiliza a função de erro que está implementada na classe *Network*, que é a função de custo que utilizamos como base para a criação de nosso algoritmo de otimização, o erro quadrático médio (MSE), também conhecido como a norma euclidiana do vetor distância entre os vetores das saídas esperadas e o das saídas obtidas pela rede.

Em todos os trechos da classe *Perceptron* acima, várias linhas estão ocultas sob o símbolo de reticências, são linhas que fazem verificações dos dados utilizados e do estado atual do objeto, se ele pode ser usado para previsão por exemplo, ou seja, se já foi treinado previamente, entre outras verificações gerais para um bom funcionamento.

3.3 Caso de uso - Base de dados MNIST

...Aqui vou descrever meu teste com o mnist 8x8... (ótima hora para mostrar a classe da matriz de confusão)

3.4 Keras - Uma API de redes neurais para *deep learning*

De acordo com seu site oficial[3], *Keras* é uma API (Interface de Programação de Aplicativos) de *deep learning* escrita em Python, e que roda sobre a plataforma de *machine learning* chamada de *TensorFlow*[4] que é de fato a biblioteca que devemos instalar em nosso ambiente Python, para podermos utilizar as redes neurais ali implementadas e quaisquer outros recursos da API *Keras* em nosso projeto de aprendizado.

[3] https://keras.io/about/
[4] https://github.com/tensorflow/tensorflow

Capítulo 4

Séries temporais

Neste capítulo são apresentados os conceitos básicos de séries temporais, com destaque para séries históricas de conversões de moedas estrangeiras. Tais séries estão disponíveis para *download* em diversos sites, incluindo o site do Banco Central do Brasil[1]. Também são discutidos brevemente os métodos tradicionais de análise das séries como a *transformada de Fourier* e de previsões de valores futuros da série, como o *SARIMA*.

A análise e predição de fatores de câmbios de moedas estrangeiras, preços de *commodities*, ou outros dados econômicos, constituem uma área essencial da economia e que exige a avaliação de um número enorme de fatores, muitos dos quais de características humanas e portanto imprevisíveis em sua exatidão, até mesmo por se tratar de um fenômeno estatístico e probabilístico, sujeito às suas incertezas.

Apesar disso pode-se estudar as variações de preços, tanto de curto prazo (dias, semanas) quanto de longo prazo (anuais, semestrais) separando seus harmônicos de oscilação presentes nas séries temporais de preços, utilizando para isso a Transformada de Fourier.

A transformada discreta de Fourier é uma função 2π-periódica e definida em:

$$F : (x_j)_{j\in\mathbb{N}} \longrightarrow (z_j)_{j\in\mathbb{N}}$$

onde $x_j \in [0, 2\pi]$ e $z_j \in C$, com $C \subset \mathbb{R}$ ou $C \subset \mathbb{C}$. Ou seja, é tabelada em $2N$ pontos igualmente espaçados (no caso das oscilações usamos os preços de cada dia, portanto levando a valores reais, e o intervalo de dias é o domínio), no intervalo $[0, 2\pi]$ (basta normalizar o intervalo de dias nesse intervalo, ou seja, denotando $x_j = j\pi/N$, com $j = 0, 1, \ldots, 2N-1$). A função pode ser definida para um conjunto de pontos que podem ser complexos ou mesmo reais. Faz isso associando aos valores $F(x_j)$ os coeficientes de Fourier (c_k), dessa forma:

$$F(x_j) = \sum_{k=0}^{2N-1} c_k e^{ikx_j}\ ,\ j = 0, 1, \ldots, 2N-1 \tag{4.1}$$

[1] https://www.bcb.gov.br/

$$c_k = \frac{1}{2N} \sum_{j=0}^{2N-1} F(x_j) e^{-ikx_j} \ , \ k = 0, 1, \dots, 2N-1 \tag{4.2}$$

Capítulo 5

Comparação dos modelos

Neste capítulo serão feitas comparações entre os modelos preditivos de séries temporais, através de medidas de desempenho comuns como percentual de acerto das previsões e observações das funções de erros dos algoritmos utilizados.

Referências

[Alsmadi *et al.* 2009] M. k. Alsmadi, K. B. Omar, S. A. Noah e I. Almarashdah. "Performance comparison of multi-layer perceptron (back propagation, delta rule and perceptron) algorithms in neural networks". Em: *2009 IEEE International Advance Computing Conference* (mar. de 2009), pgs. 296–299. doi: 10.1109/IADCC.2009.4809024 (citado na pg. 3).

[Ballini 2000] Rosangela Ballini. "Análise e Previsão de Vazões Utilizando Modelos de Séries Temporais, Redes Neurais e Redes Neurais Nebulosas". Doutorado em Engenharia Elétrica. Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, 2000 (citado na pg. 7).

[Blei e Smyth 2017] David M. Blei e Padhraic Smyth. "Science and data science". Em: *PNAS* 114.33 (ago. de 2017), pgs. 8689–8692 (citado na pg. 1).

[Budhiraja 2016] Amar Budhiraja. *Dropout in (Deep) Machine learning*. https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5. Dez. de 2016 (citado na pg. 32).

[*Ciência de Dados* 2020] *Ciência de Dados*. https://pt.wikipedia.org/wiki/Ci%C3%AAncia_de_dados. Mar. de 2020 (citado na pg. 1).

[Clevert *et al.* 2015] Djork-Arné Clevert, Thomas Unterthiner e Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". Em: *arXiv e-prints*, arXiv:1511.07289 (nov. de 2015), arXiv:1511.07289. arXiv: 1511.07289 [cs.LG] (citado nas pgs. 24, 25).

[Dellinger 2019] James Dellinger. *Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming*. https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79. Abr. de 2019 (citado na pg. 27).

[Facure 2017a] Matheus Facure. *Dificuldades no Treinamento de Redes Neurais - Examinando o problema de gradientes explodindo ou desvanecendo*. https://matheusfacure.github.io/2017/07/10/problemas-treinamento/. Jul. de 2017 (citado na pg. 22).

[Facure 2017b] Matheus Facure. *Funções de Ativação - Entendendo a importância da ativação correta nas redes neurais*. https://matheusfacure.github.io/2017/07/12/activ-func/. Jul. de 2017 (citado nas pgs. 22, 23, 25, 33).

[Géron 2019] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2º. O'Reilly, 2019 (citado nas pgs. 2, 5, 6, 10, 12, 32, 34).

[Grus 2016] Joel Grus. *Data Science do Zero*. 1º. O'Reilly, 2016 (citado nas pgs. 2, 20, 34).

[Guidorizzi 1986] Hamilton Luiz Guidorizzi. *Um curso de cálculo*. v. 2. LTC, 1986. isbn: 8521604254 (citado na pg. 10).

[Heaton 2017] Jeff Heaton. *The Number of Hidden Layers*. https://www.heatonresearch.com/2017/06/01/hidden-layers.html. Jun. de 2017 (citado nas pgs. 35, 36).

[Kopec 2019] David Kopec. *Problemas Clássicos de Ciência da Computação com Python*. 1º. Novatec, 2019 (citado nas pgs. 2, 5, 7, 10, 15, 18).

[Al-Masri 2019] Anas Al-Masri. *What Are Overfitting and Underfitting in Machine Learning?* https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690. Jun. de 2019 (citado na pg. 32).

[Morettin e Singer 2020] Pedro A. Morettin e Julio M. Singer. *Introdução à Ciência de Dados - Fundamentos e Aplicações*. Departamento de Estatística. Universidade de São Paulo, 2020 (citado na pg. 1).

[Xu *et al.* 2015] Bing Xu, Naiyan Wang, Tianqi Chen e Mu Li. "Empirical Evaluation of Rectified Activations in Convolutional Network". Em: *arXiv e-prints*, arXiv:1505.00853 (mai. de 2015), arXiv:1505.00853. arXiv: 1505.00853 [cs.LG] (citado nas pgs. 23–25).