Universidade Federal do Espírito Santo

Centro Universitário Norte do Espírito Santo

Curso de Bacharelado em Matemática Industrial

Matheus Becali Rocha

MÉTODOS DE OTIMIZAÇÃO PARA O APRENDIZADO DE MÁQUINA SUPERVISIONADO E APLICAÇÕES

São Mateus

2021

Matheus Becali Rocha

MÉTODOS DE OTIMIZAÇÃO PARA O APRENDIZADO DE MÁQUINA SUPERVISIONADO E APLICAÇÕES

Trabalho submetido ao Colegiado do Curso de Bacharelado em Matemática Industrial da UFES (Campus São Mateus), como requisito parcial para a obtenção do grau de Bacharel em Matemática.

São Mateus

2021

Matheus Becali Rocha

MÉTODOS DE OTIMIZAÇÃO PARA O APRENDIZADO DE MÁQUINA SUPERVISIONADO E APLICAÇÕES

Trabalho submetido ao Colegiado do Curso de Bacharelado em Matemática Industrial da UFES (Campus São Mateus), como requisito parcial para a obtenção do grau de Bacharel em Matemática.

Aprovada em 07 de Outubro de 2021.

Comissão Examinadora

Prof. Leonardo D. Secchin
Universidade Federal do Espírito Santo
Orientador

Prof. Luís Otávio Rigo Júnior
Universidade Federal do Espírito Santo

Prof. Oberlan Christo Romão
Universidade Federal do Espírito Santo

Agradecimentos

A Deus, que me auxiliou para que meus objetivos fossem alcançados e por ter me dado saúde e força durante todos os meus anos de estudos.

Aos meus pais e meu irmão, que me incentivaram nos momentos difíceis e compreenderam a minha ausência enquanto eu me dedicava à realização deste trabalho.

Aos amigos, que estiveram sempre ao meu lado, pela amizade incondicional e pelo apoio demonstrado ao longo de todo o tempo em que me dediquei a este trabalho.

Ao professor Leonardo, por ser meu orientador e ter desempenhado tal função com dedicação e amizade.

A todos que contribuíram, de alguma forma, para a realização deste trabalho.

À FAPES, pelo apoio dado pelo processo 116/2019.

Por fim, a todos o meu muito obrigado.

"It seemed really amazing that you could write a few lines of code and have it learn to do interesting things"

Andrew Ng.

Resumo

Neste trabalho abordamos o que chamamos de aprendizado de máquina, modelos matemáticos e algoritmos capazes de classificar a resposta a uma dada situação (por exemplo, identificar caracteres escritos à mão a partir de padrões preestabelecidos; identificar padrões de rostos, figuras geométricas, etc.). Nesse estudo foram considerados algoritmos tipicamente usados para otimizar os parâmetros de uma rede neural (ou "treinar" o modelo). O treinamento é feito a partir de pares entrada/saída conhecidos (aprendizado supervisionado), minimizando uma função alvo que mede o erro entre as entradas e as respostas da rede neural em treinamento. Após o treinamento, isto é, a otimização dos parâmetros da rede por um algoritmo de otimização, a rede é capaz de fornecer respostas esperadas à dados de entrada desconhecidos. Comparamos o desempenho dos algoritmos de otimização mais conhecidos e utilizados na literatura em problemas de classificação. Dentre eles, o método do gradiente estocástico, e suas variantes, gradiente estocástico com momento, AdaGrad, RMSProp e Adam. Apresentamos outras técnicas, como as redes convolucionais, e aplicações.

Palavras-chave: Aprendizado de máquina supervisionado. Gradiente estocástico. Gradiente estocástico com Momento. Adagrad. Adam. RMSprop. Tensorflow.

Abstract

In this work, we deal with the so-called machine learning, mathematical models, and algorithms capable of classify the response to a given situation (for example, identifying handwritten characters from pre-established patterns; identifying face patterns, geometric figures, etc.). Typical algorithms used to optimize the parameters of a neural network (or "train" the model) were considered. The training is done from known input/output pairs (supervised learning), minimizing a target function that measures the error between the inputs and responses given by the neural network. After training, that is, optimization of the network parameters by an optimization algorithm, the network can provide expected answers to unknown input data. We compare the performance of the best-known optimization algorithms used in the literature on classification problems. Among them, the stochastic gradient method and its variants, stochastic gradient with momentum, AdaGrad, RMSProp, and Adam. We present other techniques, such as convolutional networks, and applications.

Keywords: Supervised machine learning. Stochastic Gradient Descent. Stochastic Gradient Descent with Momentum. Adagrad. Adam. RMSprop. Tensorflow.

Lista de Figuras

Lista de Símbolos

$w_{ij}^{[l]}$	peso do j-ésimo neurônio na $(l-1)$-ésima camada do i-ésimo neurônio da l-ésima camada
$b_j^{[l]}$	viés do j-ésimo neurônio na l-ésima camada
$x^{(i)}$	vetor de coluna para o i-ésimo exemplo de entrada
$y^{(i)}$	valor de saída esperado para o i-ésimo exemplo
$\hat{y}$	vetor de saída calculado pela rede. Também pode ser denotado como $a^{[L]}$, onde L é o número de camadas da rede
$C(w, b)$	função de custo/perda
$\delta_j^{[l]}$	erro do neurônio i na camada l
$\mathbb{E}[X]$	esperança de X
$\mathbb{E}[X\|y]$	esperança condicional de X dado y
$\mathbb{V}[X]$	variância de X
$\mathbb{V}[X\|Y]$	variância condicional de X dado Y

Sumário

1 Introdução

O tema aprendizado de máquina (do inglês *machine learning*) tem se tornado cada vez mais estudado por pesquisadores e setores da indústria. Sua relevância está em ascensão devido aos avanços no poder computacional e no crescente número de dados disponíveis. O aprendizado de máquina é aplicado em diversas áreas, podendo ser utilizada na matemática, medicina (BAXT, 1995), direito (ALARIE; NIBLETT; YOON, 2018), entre inúmeras outras.

O surgimento da primeira ideia de neurônio artificial aconteceu por volta dos anos 40, proposta por McCulloch e Pitts (1943). No início, tratava-se de um modelo simplificado compatível ao que era conhecido na época acerca dos neurônios biológicos. Com o passar dos anos essa ideia entrou em desuso. Em 1949, a publicação do livro *The Organization of Behavior: A Neuropsychological Theory*, uma obra feita por Hebb (1949), acrescentou um fato importante sobre neurônios: quanto mais são utilizados mais fortalecidos eles ficam, um conceito fundamental para lidarmos como os humanos aprendem. Ou seja, se mais de um neurônio dispara ao mesmo tempo, as conexões entre eles são fortalecidas. Em 1958 surgiram os neurônios *perceptrons* (ROSENBLATT, 1958) inspirados no *Modelo McCulloch-Pitts*. Os *perceptrons* são classificadores binários utilizados no âmbito do aprendizado supervisionado e sua ideia consiste em fluir suas entradas $x_1, x_2, \ldots$ para um único valor de saída $f(x)$, seguindo o modelo *feed-forward*. Uma rede compondo vários neurônios *perceptron*, dispostos em várias camadas, é chamada *multilayer perceptron* (MLP).

Entre 1974 a 1980, pesquisas ligadas à inteligência artificial sofreram grandes cortes de financiamento, impedindo avanços na área. Esse período ficou conhecido como Era das Trevas da IA[1] (*Dark Age*) ou Inverno da IA (*AI Winter*). Ao fim dessa era sem mudanças para a IA, houve o surgimento da primeira ideia de *backpropagation*, proposta por Rumelhart, Hinton e Williams (1986). Em 1998, surge o primeiro trabalho envolvendo redes convolucionais (LECUN et al., 1998). A IA teve seu *boom* a partir de 2010, quando surgiram diversos trabalhos na área, tratando do reconhecimento da fala, de objetos, e

[1] Abreviação para Inteligência Artificial

das redes neurais extremamente profundas (*Deep Nets*). Muito desse avanço deveu-se à grande quantidade dados hoje disponíveis, e que cresce a cada ano.

No ramo da inteligência artificial existem inúmeras classes de aprendizagem (RUSSELL; NORVIG, 2002), algumas principais são:

- **Aprendizado de Máquina Supervisionado**, modelos treinados utilizando dados externos para os quais sabemos a resposta para dada entrada. A rede então passa a antever a resposta a dados desconhecidos após a fase de treinamento. Comumente utilizado para dados previamente rotulados (por exemplo, reconhecimento de expressões faciais);
- **Aprendizado de Máquina Não Supervisionado**, modelos que não necessitam que o usuário os supervisione. Buscam descobrir padrões a partir do próprio conjunto de dados sem referência a resultados rotulados ou previsões. O aprendizado, neste caso, ocorre através de agrupamento dos exemplos por similaridade. (por exemplo, detecção de anomalias);
- **Aprendizado por Reforço**, modelos onde todos os agentes têm objetivos explícitos, podendo obter informações do ambiente para escolher ações que influenciam o mesmo. Devem aprender a operar apesar das circunstâncias usando um sistema de recompensas (por exemplo, encontrar a saída de um labirinto).

Esse trabalho visa o estudo introdutório de modelos e técnicas de otimização utilizadas no aprendizado de máquina supervisionado. São abordados algoritmos de otimização tradicionais para treinamento de redes neurais (gradiente estocástico e suas variantes). Esses algoritmos necessitam do cálculo de gradientes, que, no contexto, é realizado por uma técnica baseada na aplicação sucessiva da regra da cadeia ("retro propagação", ou, do inglês, *backpropagation*). Exibiremos em detalhes o processo de *backpropagation* para duas funções de custo específicas. Testes numéricos comparativos são realizados em problemas de classificação de imagem. No primeiro momento, fazemos uma comparação desses algoritmos sobre o problema de classificação de caracteres numéricos. Em um segundo momento, técnicas de *deep learning* são abordadas, tratando problemas um pouco mais complexos.

Foram utilizados conjuntos de dados (*datasets*) pré-processados e de livre acesso MNIST (LECUN; CORTES, 2010), EuroSAT (HELBER et al., 2018) e Beans (LAB, 2020). **MNIST** é um conjunto de imagens de caracteres numéricos de 0 a 9, dispostos em imagens de tamanho (28, 28) *pixels* em escala cinza 8-bits. **EuroSAT** é composta por

imagens de trechos urbanos e rurais retiradas por satélites com foco na União Europeia, cujo tamanho é (64, 64) *pixels* na escala RGB. **Beans** possui imagens RGB (500, 500) de folhas de feijões, saudáveis e não saudáveis, retiradas no campo com auxílio da câmera do celular. Apresentamos o método da **gradiente estocástico** (*Stochastic Gradient Descent* - SGD) que surge como principal ferramenta na área, sendo de extrema valia para os primeiros estudos. Mais detalhes podem ser vistos em (BOTTOU; CURTIS; NOCEDAL, 2018). Apresentamos também variações do método do gradiente, tais como:

- **Gradiente estocástico com momento** (*Stochastic Gradient Descent with Momentum* - SGDM, (SUTSKEVER et al., 2013));
- **Gradiente adaptativo** (*Adaptive Gradient Algorithm* - ADAGrad, (DUCHI; HAZAN; SINGER, 2011));
- **"Propagação da Raiz Quadrada Média"** (*Root Means Square Propagation* - RMSProp, (TIELEMAN; HINTON et al., 2012));
- **"Momento adaptável estimado"** (*Adaptive Moment Estimation* - Adam, (KINGMA; BA, 2015)).

O restante deste trabalho é divido da seguinte forma. No Capítulo 2, mostraremos conceitos básicos sobre redes neurais, apresentando as principais funções de ativações e como uma rede neural é treinada. No Capítulo 3, falaremos sobre o Método do Gradiente Estocástico, introduziremos conceitos de esperança matemática e provaremos sua convergência para o caso de passo constante. Apresentaremos também variantes do SGD e trataremos o problema de reconhecimento de caracteres numéricos, com resultados numéricos. No Capítulo 4, apresentaremos outros problemas de classificação em redes neurais com auxílio da plataforma *TensorFlow*. Por fim, as conclusões vêm no último capítulo.

2 Conceitos básicos

Neste capítulo apresentamos conceitos fundamentais a respeito de redes neurais para representação dos problemas de classificação, bem como o método clássico de gradiente para otimização irrestrita.

2.1 Redes Neurais com topologia de camadas

Uma rede neural *feed-forward* é uma "réplica computacional" dos neurônios do cérebro humano e é um subconjunto de **aprendizagem de máquina** (*machine learning*), formada por neurônios artificiais. O neurônios mais conhecido e importante para o entendimento de redes neurais são os **perceptrons**. Podemos escrever a saída desses neurônios de maneira geral, dada por

$$\hat{y} = g\bigg(\sum_{i=1}^{m} w_i x_i + b\bigg).$$

A notação é simplificada. w e x são vetores cujas componentes são chamadas **peso** e **entrada**, b um valor real, chamado **viés** e g a função de ativação não-linear, aqui podendo ser a **sigmoid**, uma de **degrau** ou outras que observaremos na seção 2.1.1.

Uma rede neural artificial com arquitetura organizada em camadas do tipo *feed-forward* é dada por diversos neurônios artificiais dispostos em diferentes camadas. Cabe destacarmos que existem outras arquiteturas de rede como, por exemplo, Memória associativa bidirecional (BAM), Redes de Hopfield, Redes de Kohonen. **No entanto, por simplicidade, vamos nos referir durante todo o trabalho às redes com arquitetura em camadas tipo *feed-forward* simplesmente como redes neurais.** Existem diversas classes de redes neurais, entre elas, as **redes neurais rasas** (*Shallow Neural Networks*) e as **redes neurais profundas** (*Deep Neural Networks*). As redes neurais rasas se diferem das redes profundas na quantidade de camadas ocultas. Elas possuem apenas uma ou duas camadas ocultas e sua estrutura pode ser visualizada na Figura 1.

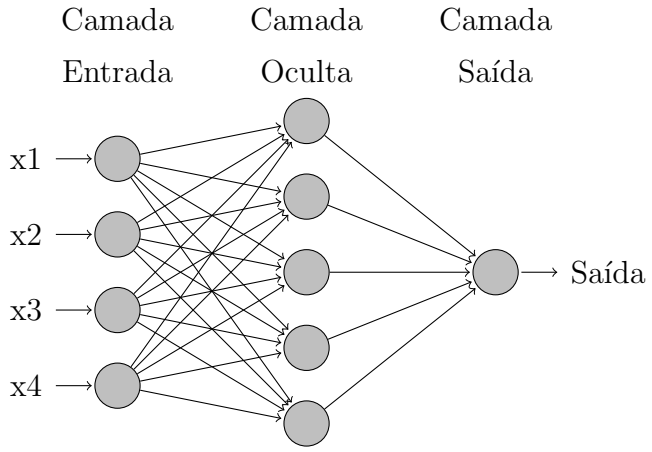


Figura 1: Rede neural artificial rasa.

Já as redes neurais profundas possuem mais camadas ocultas. Essas camadas extras permitem a distribuição de recursos das camadas mais à esquerda para as demais camadas, permitindo o treinamento com menos neurônios intermediários em relação à rede rasa, e obtendo um desempenho igual ou superior. Na Figura 2 um exemplo de uma rede profunda simples, com apenas 3 camadas ocultas.

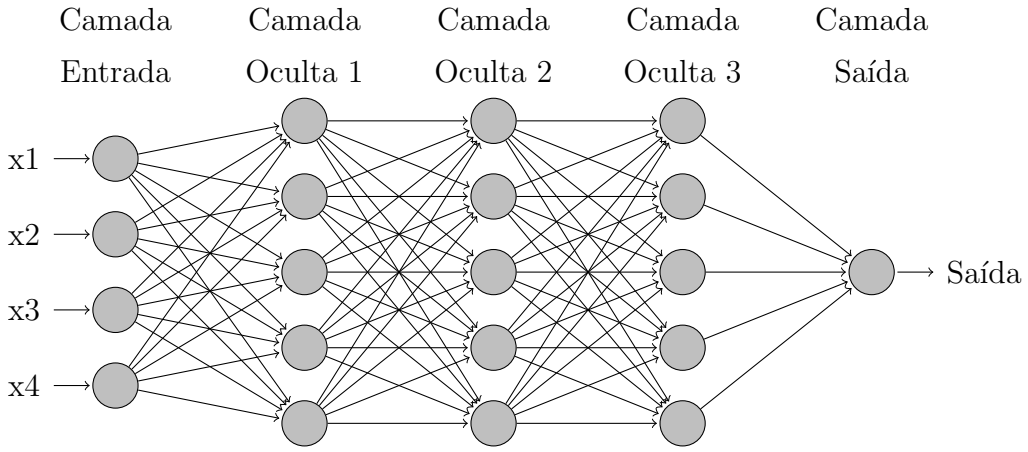


Figura 2: Rede neural artificial profunda.

Nas figuras acima, o processo de propagação pela rede é da esquerda para a direita (redes ***feed-forward***), onde a camada de entrada recebe um vetor x e propaga até a camada de saída realizando cálculos matemáticos. Ao neurônio i da camada l e j da camada $l-1$ são associados um vetor de pesos $w_{ij}^{[l]}$ e o viés $b_i^{[l]}$. Esses objetos são iniciados aleatoriamente, e ajustados por um algoritmo de otimização ao minimizar a função de erro. Os dados fluem, passando por cada camada oculta, e avança até a camada de saída com os parâmetros ajustados, retornando a previsão do modelo.

Existem também **redes neurais convolucionais** (*Convolutional Neural Networks - CNN*). São uma subclasse das redes neurais artificiais, e sua grande diferença está no fato de conseguir tratar matrizes de representação espacial sem a necessidade de pré-processamento para extração de características prévia para redução do número de entradas da rede, permitindo utilizar certas propriedades que antes não eram possíveis, como matrizes de convolução (para maiores detalhes ver Seção 4.2). Isso torna o processo de *feed-forward* muito mais eficiente e reduz a quantidade de parâmetros presentes na rede. Essas redes são amplamente utilizadas para treinamento com imagens em alta definição (ZHAO; DU; EMERY, 2017), obtendo muito mais detalhes. No capítulo 4 veremos um pouco mais sobre redes convolucionais e algumas aplicações com a ajuda do *Tensorflow*.

2.1.1 Funções de ativação

Uma função de ativação ou função de transferência, tem, dentre outras, a finalidade de evitar o acréscimo progressivo dos valores de saída ao longo das camadas da rede, visto que tais funções possuem valores máximos e mínimos contidos em intervalos determinados. De certa forma, elas "modulam" o fluxo para uma correta saída. O neurônio perceptron possui em seu modelo original uma função de ativação chamada **função de degrau** ou *Heaviside step* sendo descrita como:

$$H(z) = \begin{cases} 0 & \text{, se } \sum_i w_i x_i + b \leq 0 \\ 1 & \text{, se } \sum_i w_i x_i + b > 0 \end{cases}$$

e seu gráfico pode ser visualizado na Figura 3.

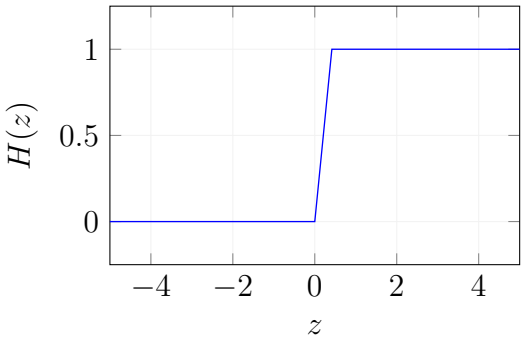
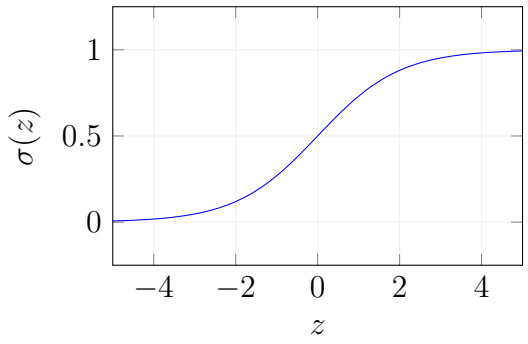


Figura 3: Função de degrau.

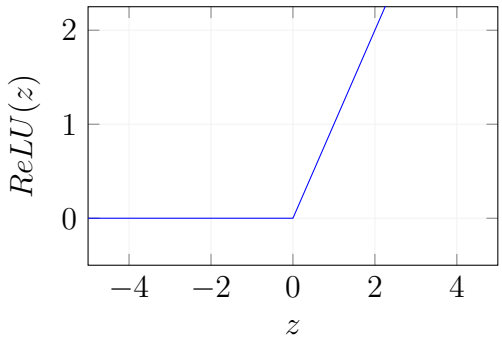
Em geral, não é muito utilizada atualmente devido à variação brusca das saídas. De fato, como a saída é apenas 0 ou 1, ao mudarmos alguns valores nos parâmetros/entrada, ela altera todo o decorrer da rede (olhando a rede neural como uma aplicação $x \to f(x)$,

podemos pensar que a saída da rede não é contínua). É utilizada como um método de pesar evidências para tomada de decisões, estando presentes em portas lógicas da programação (e.g. AND, NAND).

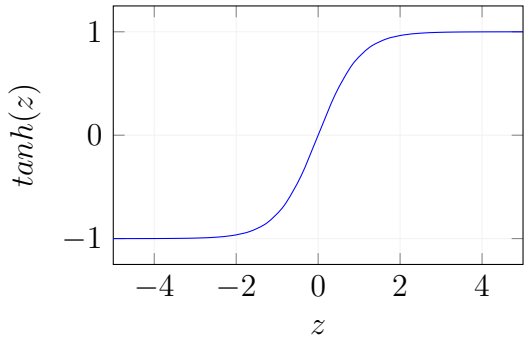
A Figura 4 traz algumas das principais funções de ativação utilizadas em um neurônio *perceptron*. Uma em particular é a função suave **sigmoid**.



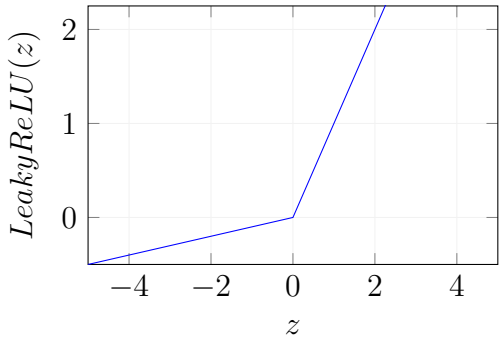
(a) Função ativação Sigmoid.



(b) Função ativação ReLU.



(c) Função ativação tangente hiperbólica.



(d) Função de ativação Leaky ReLU.

Figura 4: Funções de ativação.

Uma breve explicação sobre as principais funções de ativações escritas na Figura 4:

- **Sigmoid** (σ): utilizada para classificação binária por possuir valores máximos e mínimos contidos no intervalo $[0, 1]$, podendo também ser utilizada para multi-classificação. Comumente utilizada na camada de saída da rede. A função e sua derivada são

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \qquad \sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

- **Tangente Hiperbólica** (tanh): semelhante à Sigmoid, tanh é utilizada para classificação binária ou multi-classificação, porém seus valores são contidos no intervalo maior $[-1, 1]$. Comumente utilizada nas camadas intermediárias da rede. A função

e sua derivada são

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \qquad tanh'(z) = 1 - tanh(z)^2$$

- ***Rectified Linear Unit*** $(relu)$: amplamente utilizada em diversas redes, sua principal ideia é transformar valores negativos em zeros. Contudo, caso a rede tenha muitos valores negativos isso se torna um problema. A função e sua derivada são

$$relu(z) = max(0, z), \qquad relu'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{caso contrário} \end{cases}$$

- ***Leaky Rectified Linear Unit*** $(lrelu)$: entra para solucionar o problema dos zeros que ocorre na $ReLU$. Ao invés de transformar os valores negativos em zero, multiplica-os por um valor pequeno, normalmente **0,01**. A função e sua derivada são

$$lrelu(z) = \begin{cases} z, & z > 0 \\ 0.01z, & \text{caso contrário} \end{cases}, \qquad lrelu'(z) = \begin{cases} 1, & z > 0 \\ 0.01, & \text{caso contrário} \end{cases}$$

2.1.2 Treinamento da rede neural

O processo de treinamento de uma rede neural consiste em atualizarmos os pesos e vieses da rede por um algoritmo iterativo. A continuidade proveniente da suavização das aptidões permitem o uso de métodos para minimização sem restrições que utilizam derivadas e a aplicação da regra da cadeia, sendo os esquemas de gradiente os mais usados. O estudo dos algoritmos iterativos para treinamento de redes neurais é o principal foco deste trabalho. Em particular, detalhamos à frente o processo de atualização dos pesos via *backpropagation*. Enunciamos em detalhes o método do gradiente estocástico e suas variantes. Testes numéricos comparativos serão apresentados nos próximos capítulos.

2.1.2.1 Propagação pela rede (*feed-forward*)

O propósito do *feed-forward* é propagar nosso vetor de entradas (os dados de entrada) através da rede, passando pelos neurônios. É nesse processo que os pesos w e vieses b de toda a rede são computados, e consequentemente a função erro. Vale ressaltar que os pesos e vieses não são atualizados nesse processo, esse processo ocorre no *backpropagation*, que será visto com mais detalhes posteriormente. Ao passar pelos neurônios são realizados duas operações fundamentais, de maneira sequencial da camada de entrada à camada de

saída, uma afim (ponderação de x pelos pesos w somado com o viés b), e posteriormente uma operação não linear (aplicação da função de ativação). Esse processo é propagado até chegarmos à camada de saída da rede (ou seja, nossa classificação). A Figura 5 ilustra como é realizado o processo em um neurônio.

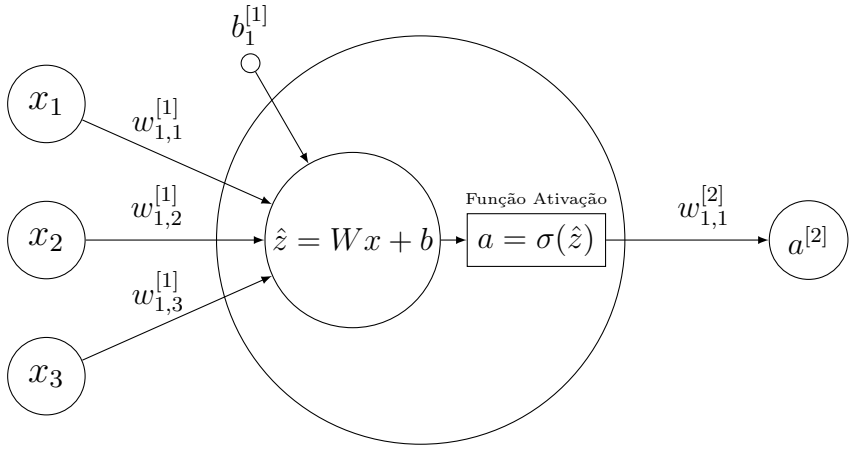


Figura 5: Propagação do fluxo para cálculo de (w, b) (*feed-forward*).

O processo ilustrado é bem simples, apenas para dar uma ideia sobre o que ocorre em um neurônio, primeiramente temos nossos dados de entrada x_1, x_2, x_3 com $x \in \mathbb{R}^{3\times1}$ e os seus respectivos pesos $w_{1,1}, w_{1,2}, w_{1,3}$ com $W \in \mathbb{R}^{1\times3}$ e o viés $b_1 \in \mathbb{R}^{1\times1}$, esses dados dentro do neurônio são calculados por $\hat{z}$ e posteriormente pela ativação e saem com destino ao resultado, encerrando a propagação ou partem para outro neurônio da camada seguinte, no caso do exemplo $a^{[2]}$, realizando os cálculos novamente. Uma observação interessante é que, nas redes neurais, podemos definir funções de ativação diferentes em cada uma das camadas. Por exemplo, uma rede pode ser estruturada contendo camadas com funções ReLU e outras com funções Sigmoid, visualizada de maneira simplificada como:

$$(\text{ENTRADA} \rightarrow \text{ReLU} \rightarrow \text{ReLU} \rightarrow \text{Sigmoid} \rightarrow \text{SAÍDA}).$$

Após o processo de propagação pela rede realizando os cálculos do erro na resposta da rede, ocorre o cálculo de gradientes pelo processo de retro propagação ou *backpropagation*, onde percorremos novamente a rede, mas no sentido contrário. Para aplicarmos esse processo, nossa função de ativação precisa ser diferenciável. O *backpropagation* consiste em aplicarmos recursivamente a regra da cadeia para computar gradientes em w e b. Este processo é central para os algoritmos de otimização do tipo gradiente, pois calcula a direção de anti gradiente usada na iteração desses métodos, movendo a rede cada vez mais perto do erro mínimo. Em outras palavras, realiza o cálculo da influência de cada

peso e viés no erro de classificação ou previsão. A Figura 6 ilustra a ideia de como é este processo, onde tomamos como base a Figura 5. No neurônio mais a direita da imagem $a^{[2]}$ é calculado o gradiente da função custo e passado para os neuronios à esqueda, com isso é realizado a regra da cadeia para computar as derivadas parciais dos pesos e vieses e consequentemente atualizar os pesos e vieses em cada camada da rede. Na Seção seguinte adentraremos nesses calculos.

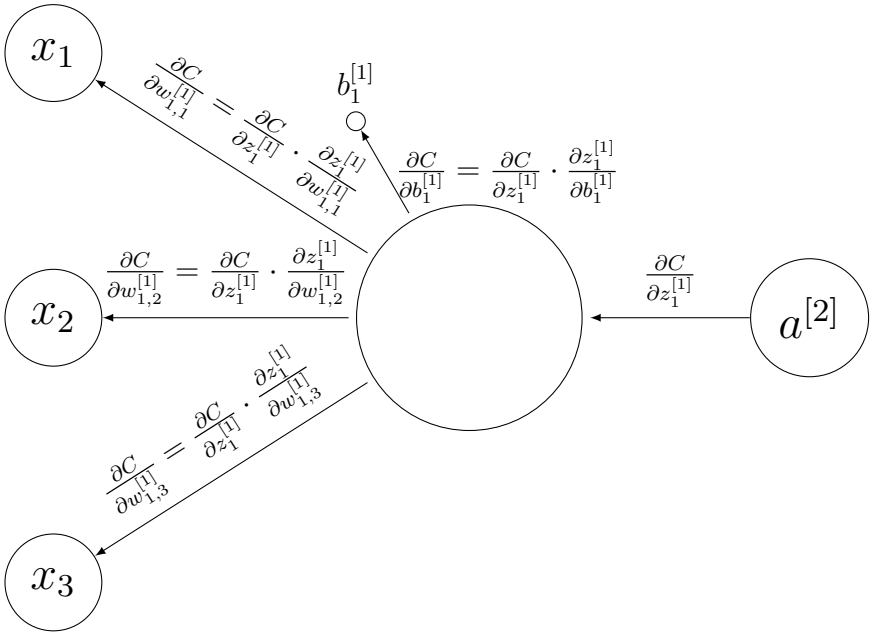


Figura 6: Retro propagação do erro para atualização de (w, b) (*backpropagation*).

Em resumo, as variáveis w e b são definidas na rede neural. Nosso objetivo é minimizar uma função de erro $C(w, b)$, em função de w e b, usando *feed-forward*, *backpropagation* e um algoritmo de otimização. Para isso, inicializamos os pesos w e os vieses b com valores randômicos próximos de zero.

As funções de erro mais conhecidas na literatura são o **erro quadrático médio** e o **erro logarítmico**, e serão consideradas nas próximas subseções. Uma função de erro é definida sobre uma soma em todo o conjunto de treinamento, dada por $C = \frac{1}{m}\big(\sum_{i=1}^{m} \text{erro dado } i\big)$. As funções de erro são de extrema importância, pois medem o "quão bem" a rede neural se saiu em relação ao resultado de sua amostra de treinamento e a saída esperada. Entrando no problema proposto na pesquisa, teremos uma matriz w_{n_e,n_c} onde n_e é o número de entradas e n_c é o número de neurônios da camada. Uma breve introdução à notação a seguir: o sobrescrito (i) denotará o dado de treinamento i enquanto o sobrescrito $[l]$ denotará a camada l da rede.

2.1.2.2 Cálculo eficiente do gradiente (*backpropagation*) – erro quadrático médio

A função Erro Quadrático Médio ou função Custo Quadrático é definida por:

$$C(w,b) = \frac{1}{2m}\sum_{i=0}^{m} ||a(x^{(i)};w,b) - y^{(i)}||^2, \tag{2.1}$$

Aqui, o ponto

$$z = (w_{11}, w_{12}, \ldots, w_{1,n_e}, w_{21}, \ldots, w_{2,n_e}, \ldots, w_{n_c,1}, \ldots, w_{n_c,n_e}, b_1, \ldots, b_{n_c}) \in \mathbb{R}^{(n_e+1)\times n_c}$$

foi computado percorrendo a rede da camada de entrada até à camada de saída (*feed-forward*), conforme a Figura 5. O algoritmo *backpropagation* com a função de ativação Sigmoid σ (dada na Seção 2.1.1), é descrito como segue.

Definimos o erro $\delta_i^{[l]}$ do neurônio i na camada l por

$$\delta_i^{[l]} = \frac{\partial C}{\partial z_i^{[l]}}.$$

Começando da última camada L e indo de encontro a primeira, aplicamos a regra da cadeia no erro $\delta_i^{[L]}$ na camada L, obtemos então

$$\delta_i^{[L]} = \frac{\partial C}{\partial z_i^{[L]}} = \sum_j \frac{\partial C}{\partial a_j^{[L]}}\frac{\partial a_j^{[L]}}{\partial z_i^{[L]}}.$$

o somatório em relação à j corresponde à soma dos neurônios na camada de saída. Como a ativação da camada de saída $a_j^{[L]}$ do j-ésimo neurônio depende somente dos pesos da entrada $z_i^{[L]}$ do i-ésimo neurônio onde $i = j$, por ser a última camada, nossos neurônios i dependem apenas de $z_i^{[L]}$, e o resultado é dado por

$$\delta_i^{[L]} = \sum_i \frac{\partial C}{\partial a_i^{[L]}}\frac{\partial a_i^{[L]}}{\partial z_i^{[L]}}.$$

Como nossa função de ativação é a sigmoid, podemos chamar $a_i^{[L]} = \sigma(z_i^{[L]})$, tendo então

$$\delta_i^{[L]} = \sum_i \frac{\partial C}{\partial a_i^{[L]}}\frac{\partial \sigma(z_i^{[L]})}{\partial z_i^{[L]}} = \sum_i \frac{\partial C}{\partial a_i^{[L]}}\sigma'(z_i^{[L]}).$$

O próximo passo é realizar o cálculo do erro $\delta^{[l]}$ em termos da próxima camada $\delta^{[l+1]}$.

Neste caso, temos

$$\delta_i^{[l]} = \sum_j \frac{\partial C}{\partial z_j^{[l+1]}} \frac{\partial z_j^{[l+1]}}{\partial z_i^{[l]}} = \sum_j \frac{\partial z_j^{[l+1]}}{\partial z_i^{[l]}} \delta_j^{[l+1]}.$$

Sabemos que $z_j^{[l+1]} = \sum_j w_{ij}^{[l+1]} a_j^{[l]} + b_i^{[l+1]} = \sum_j w_{ij}^{[l+1]} \sigma(z_j^{[l]}) + b_i^{[l+1]}$. Derivando $z_j^{[l+1]}$ obtemos

$$\frac{\partial z_j^{[l+1]}}{\partial z_i^{[l]}} = w_{ij}^{[l+1]} \sigma'(z_j^{[l]}).$$

Assim, $\delta_i^{[l]}$ é dado por

$$\delta_i^{[l]} = \sum_i w_{ij}^{[l+1]} \sigma'(z_j^{[l]}) \delta_i^{[l+1]}.$$

Agora com os erros $\delta_i^{[l]}$ de todas as camadas da rede neural, conseguimos calcular as derivadas $\frac{\partial C}{\partial w_{ij}^{[l]}}$ e $\frac{\partial C}{\partial b_i^{[l]}}$:

$$\frac{\partial C}{\partial w_{ij}^{[l]}} = \frac{\partial C}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial w_{ij}^{[l]}} = \delta_i^{[l]} a_j^{[l-1]}, \qquad \frac{\partial C}{\partial b_i^{[l]}} = \frac{\partial C}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial b_i^{[l]}} = \delta_i^{[l]}.$$

Isso será usado à frente na iteração de gradiente. Ou seja, atualizaremos os pesos $w_{ij}^{[l]}$ e os viés $b_i^{[l]}$ da rede, dando um passo na direção oposta ao gradiente, onde η representa a taxa de aprendizagem:

$$w_{ij}^{[l]} \leftarrow w_{ij}^{[l]} - \eta \frac{\partial C}{\partial w_{ij}^{[l]}}, \qquad b_i^{[l]} \leftarrow b_i^{[l]} - \eta \frac{\partial C}{\partial b_i^{[l]}}.$$

2.1.2.3 Cálculo eficiente do gradiente (*backpropagation*) – erro logarítmico

A função de Custo do erro logarítmico é definida por:

$$C(w,b) = -\frac{1}{m} \sum_{i=1}^{m} \Big[y^{(i)} \log a(x^{(i)}; w, b) + (1 - y^{(i)}) \log(1 - a(x^{(i)}; w, b)) \Big]. \tag{2.2}$$

Treinar nossa rede significa atualizar nossos pesos e vieses, w e b, usando o gradiente no ponto corrente. Em cada passo, precisamos calcular $\frac{\partial C}{\partial w_{ij}^{[l]}}$ e $\frac{\partial C}{\partial b_i^{[l]}}$. Para isso, novamente aplicamos a regra da cadeia:

$$\frac{\partial C}{\partial w_{ij}^{[l]}} = \sum_{i=1}^{m} \frac{\partial C}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial w_{ij}^{[l]}}, \qquad \frac{\partial C}{\partial b_i^{[l]}} = \sum_{i=1}^{m} \frac{\partial C}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial b_i^{[l]}}.$$

Aplicando a regra da cadeia novamente calculamos o termo $\frac{\partial C}{\partial z_i^{[l]}}$:

$$\frac{\partial C}{\partial z_i^{[l]}} = \sum_{i=1}^{m} \frac{\partial C}{\partial a_i^{[l-1]}} \frac{\partial a_i^{[l-1]}}{\partial z_i^{[l]}}.$$

Agora,

$$\begin{aligned}
\frac{\partial C}{\partial a_i^{[l-1]}} &= -\sum_{i=1}^{m} \frac{\partial}{\partial a_i^{[l-1]}} \Big(y_i \log(a_i^{[l-1]}) + (1-y_i)\log(1-a_i^{[l-1]}) \Big) \\
&= -\sum_{i=1}^{m} \left(y_i \frac{\partial \log(a_i^{[l-1]})}{\partial a_i^{[l-1]}} + (1-y_i) \frac{\partial \log(1-a_i^{[l-1]})}{\partial a_i^{[l-1]}} \right) \\
&= -\sum_{i=1}^{m} \left(y_i \frac{1}{a_i^{[l-1]}} + (1-y_i) \frac{1}{1-a_i^{[l-1]}} \right).
\end{aligned}$$

Também,

$$\frac{\partial a_i^{[l-1]}}{\partial z_i^{[l]}} = \sigma'(z) = a_i^{[l-1]}(1-a_i^{[l-1]}).$$

Assim,

$$\begin{aligned}
\frac{\partial C}{\partial z_i^{[l]}} &= -\sum_{i=1}^{m} \left(y_i \frac{1}{a_i^{[l-1]}} + (1-y_i) \frac{1}{1-a_i^{[l-1]}} a_i^{[l-1]}(1-a_i^{[l-1]}) \right) \\
&= -\sum_{i=1}^{m} \Big(y_i(1-a_i^{[l-1]}) + (1-y_i)a_i^{[l-1]} \Big) = -\sum_{i=1}^{m} \Big(a_i^{[l-1]} - y_i \Big).
\end{aligned}$$

Podemos agora calcular $\partial z_i^{[l]}$ em relação a cada elemento de w,

$$\frac{\partial z_i^{[l]}}{\partial w_{ij}^{[l]}} = \frac{\partial}{\partial w_{ij}} \Big(\sum_j w_{ij} x_j + b_i \Big) = x_j.$$

Juntando as equações, obtemos

$$\frac{\partial C}{\partial w_{ij}^{[l]}} = \sum_{i=1}^{m} \Big(a_i^{[l]} - y_i \Big) x_i^{[l]}.$$

Semelhante ao cálculo de $\frac{\partial C}{\partial w_{ij}^{[l]}}$, obtemos $\frac{\partial C}{\partial b_i^{[l]}}$ como sendo

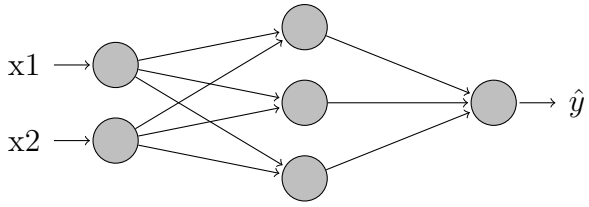
$$\frac{\partial C}{\partial b_i^{[l]}} = \sum_{i=1}^{m} \Big(a_i^{[l]} - y_i \Big).$$

E por fim atualizaremos os pesos w e os vieses b da rede dando um passo na direção oposta ao gradiente:

$$w_{ij}^{[l]} \leftarrow w_{ij}^{[l]} - \eta \frac{\partial C}{\partial w_{ij}^{[l]}}, \qquad b_i^{[l]} \leftarrow b_i^{[l]} - \eta \frac{\partial C}{\partial b_i^{[l]}}.$$

2.1.2.4 Exemplo

Mostraremos um exemplo e como fazê-lo matematicamente para entender como propagamos pela rede através do *feed-forward* e atualizamos os pesos e viés usando o *backpropagation*. Nosso objetivo é obter uma saída $y = 0.9$.



Seja dada a rede neural acima, composta por 2 neurônios de entrada, 3 neurônios na camada escondida e 1 na camada de saída. Os índices das camadas são dados por $l = 0, 1, 2$, onde $l = 0$ se refere à camada de entrada. Consideramos os dados de entrada (camada $l = 0$) $x_1 = 0.5$ e $x_2 = 0.2$, a taxa de aprendizagem $\eta = 0.5$ e os pesos iniciais da rede como sendo:

Na primeira camada $l = 1$:

$$W^{[1]} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}_{3\times 2} = \begin{bmatrix} 0.1 & 0.4 \\ -0.5 & 0.1 \\ 0.9 & 0.5 \end{bmatrix}, \qquad b^{[1]} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}_{3\times 1} = \begin{bmatrix} -0.30 \\ 0.82 \\ 0.03 \end{bmatrix}.$$

Na segunda camada $l = 2$:

$$W^{[2]} = \begin{bmatrix} w_{11} & w_{21} & w_{21} \end{bmatrix}_{1\times 3} = \begin{bmatrix} 0.8 & -0.12 & 0.3 \end{bmatrix}, \qquad b^{[2]} = \begin{bmatrix} b_1 \end{bmatrix}_{1\times 1} = \begin{bmatrix} -0.8 \end{bmatrix},$$

e o vetor de entradas x ou $a^{[0]}$:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{2\times 1}.$$

Agora definidas todos os valores iniciais para as variáveis, começaremos o processo de propagação pela rede realizando o *feed-forward*, onde $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$ pode ser escrito

como

$$z^{[1]} = \begin{bmatrix} w_{11} + w_{21} \\ w_{12} + w_{21} \\ w_{13} + w_{23} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

$$\implies z^{[1]} = \begin{bmatrix} 0.1 & 0.4 \\ -0.5 & 0.1 \\ 0.9 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.2 \end{bmatrix} + \begin{bmatrix} -0.30 \\ 0.82 \\ 0.03 \end{bmatrix},$$

$$\implies z^{[1]} = \begin{bmatrix} -0.17 \\ 0.59 \\ 0.58 \end{bmatrix}.$$

Agora, passaremos pela função de ativação $a^{[1]} = \sigma(z^{[1]})$,

$$a^{[1]} = \begin{bmatrix} \sigma(-0.17) \\ \sigma(0.59) \\ \sigma(0.58) \end{bmatrix} = \begin{bmatrix} 0.4576 \\ 0.6433 \\ 0.6410 \end{bmatrix}.$$

Entramos na próxima camada, $l = 2$. Calculando $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$:

$$z^{[2]} = \begin{bmatrix} 0.8 & -0.12 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.4576 \\ 0.6433 \\ 0.6410 \end{bmatrix} + \begin{bmatrix} -0.8 \end{bmatrix},$$

$$\implies z^{[2]} = \begin{bmatrix} -0.3188 \end{bmatrix}.$$

Agora, passaremos pela função de ativação $a^{[2]} = \sigma(z^{[2]})$:

$$a^{[2]} = \begin{bmatrix} \frac{1}{1+e^{-(-0.3188)}} \end{bmatrix} = \begin{bmatrix} \frac{1}{1+1.3754} \end{bmatrix} = \begin{bmatrix} 0.4209 \end{bmatrix}.$$

Nossa resposta para essa rede simples é $\hat{y} = 0.4209$, como queremos que nossa rede encontre $y = 0.90$, precisamos atualizarmos os pesos, para que na próxima rodada, a resposta chegue mais próximo do objetivo. Aplicaremos agora o *backpropagation*. Para isso, calcularemos o erro da saída da nossa rede. Para simplificar os cálculos do erro, omitiremos os vetores 1×1:

$$\delta^{[2]} = \frac{\partial C}{\partial z^{[2]}} = \frac{\partial C}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial C}{\partial a^{[2]}} \sigma'(z^{[2]}),$$

$$\implies \delta^{[2]} = (\hat{y} - y) \cdot \sigma'(z^{[2]}) = (0.4209 - 0.9) \cdot (\sigma(z^{[2]}) \cdot (1 - \sigma(z^{[2]}))),$$

$$\implies \delta^{[2]} = (-0.4791) \cdot (0.4209 \cdot (1 - 0.4209)) = -0.1167.$$

Calculado nosso erro, calcularemos os gradientes dos pesos e viés em cada camada da rede, de trás para frente.

Calculando $\nabla W^{[2]}$:

$$\frac{\partial C}{\partial W^{[2]}} = \delta^{[2]} \cdot a^{[1]^T} = \begin{bmatrix} -0.1167 \end{bmatrix} \cdot \begin{bmatrix} 0.4576 & 0.6433 & 0.6410 \end{bmatrix} = \begin{bmatrix} -0.0534 & -0.0751 & -0.0748 \end{bmatrix}.$$

Calculando $\nabla b^{[2]}$:

$$\frac{\partial C}{\partial b^{[2]}} = \delta^{[2]} = \begin{bmatrix} -0.1167 \end{bmatrix}.$$

Agora calcularemos o erro na camada intermediária da nossa rede, onde $\odot$ é o produto de Hadamard (isto é, o vetor obtido cujas coordenadas são o produto de coordenadas correspondentes de dois vetores):

$$\begin{aligned} & \delta^{[1]} = \left(W^{[2]^T} \cdot \delta^{[2]}\right) \odot \sigma'(z^{[1]}), \\ \implies & \delta^{[1]} = \begin{bmatrix} 0.8 \\ -0.12 \\ 0.3 \end{bmatrix} \cdot \begin{bmatrix} -0.1167 \end{bmatrix} \odot \begin{bmatrix} 0.2482 \\ 0.2294 \\ 0.2301 \end{bmatrix} = \begin{bmatrix} -0.0231 \\ 0.0032 \\ -0.0080 \end{bmatrix}. \end{aligned}$$

Calculando $\nabla W^{[1]}$:

$$\frac{\partial C}{\partial W^{[1]}} = \delta^{[1]} \cdot a^{[0]^T} = \begin{bmatrix} -0.0231 \\ 0.0032 \\ -0.0080 \end{bmatrix} \cdot \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} = \begin{bmatrix} -0.0115 & -0.0046 \\ 0.0016 & 0.0006 \\ -0.0040 & -0.0016 \end{bmatrix}.$$

Calculando $\nabla b^{[1]}$:

$$\frac{\partial C}{\partial b^{[1]}} = \delta^{[1]} = \begin{bmatrix} -0.0231 \\ 0.0032 \\ -0.0080 \end{bmatrix}.$$

Por fim com os gradientes todos calculados, atualizamos nossos pesos e viés em cada camada $l = 1, 2$:

$$\begin{aligned} & W^{[2]} \leftarrow W^{[2]} - \eta \frac{\partial C}{\partial W^{[2]}}, \\ \implies & W^{[2]} \leftarrow \begin{bmatrix} 0.8 & -0.12 & 0.3 \end{bmatrix} - 0.5 \begin{bmatrix} -0.0534 & -0.0751 & -0.0748 \end{bmatrix}, \\ \implies & W^{[2]} \leftarrow \begin{bmatrix} 0.8267 & -0.0824 & 0.3374 \end{bmatrix}. \end{aligned}$$

Atualização do viés na camada $l = 2$:

$$\begin{aligned} b^{[2]} &\leftarrow b^{[2]} - \eta \frac{\partial C}{\partial b^{[2]}}, \\ \implies b^{[2]} &\leftarrow \begin{bmatrix} -0.8 \end{bmatrix} - 0.5 \begin{bmatrix} -0.1167 \end{bmatrix}, \\ \implies b^{[2]} &\leftarrow \begin{bmatrix} -0.7416 \end{bmatrix}. \end{aligned}$$

Por fim nossos pesos da primeira camada $l = 1$:

$$\begin{aligned} W^{[1]} &\leftarrow W^{[1]} - \eta \frac{\partial C}{\partial W^{[1]}}, \\ \implies W^{[1]} &\leftarrow \begin{bmatrix} 0.1 & 0.4 \\ -0.5 & 0.1 \\ 0.9 & 0.5 \end{bmatrix} - 0.5 \begin{bmatrix} -0.0115 & -0.0046 \\ 0.0016 & 0.0006 \\ -0.0040 & -0.0016 \end{bmatrix}, \\ \implies W^{[1]} &\leftarrow \begin{bmatrix} 0.1057 & 0.4023 \\ -0.5008 & 0.0996 \\ 0.9020 & 0.5008 \end{bmatrix}. \end{aligned}$$

Atualização do viés da primeira camada $l = 1$:

$$\begin{aligned} b^{[1]} &\leftarrow b^{[1]} - \eta \frac{\partial C}{\partial b^{[1]}}, \\ \implies b^{[1]} &\leftarrow \begin{bmatrix} -0.30 \\ 0.82 \\ 0.03 \end{bmatrix} - 0.5 \begin{bmatrix} -0.0231 \\ 0.0032 \\ -0.0080 \end{bmatrix}, \\ \implies b^{[1]} &\leftarrow \begin{bmatrix} -0.2884 \\ 0.8183 \\ 0.0340 \end{bmatrix}. \end{aligned}$$

Agora atualizamos todos os pesos da rede. No código presente no Apêndice A repetimos esse processo por mais 100 vezes, o erro total caiu para -0.0029. Neste ponto, os neurônios de saída geram 0.87383236, ou seja, próximo ao nosso valor alvo que é 0.9.

2.2 Estratégia de gradiente para minimização sem restrições

O método do gradiente clássico para minimizar uma função continuamente diferenciável f consiste na iteração

$$x^{k+1} = x^k - \eta_k \nabla f(x^k),$$

onde $\eta_k > 0$ é o tamanho do passo (RIBEIRO; KARAS, 2013). O método realiza uma busca na direção oposta do gradiente para encontrar o mínimo local da função. No contexto de aprendizado de máquina, η é chamado **taxa de aprendizagem**. Esse método aplicado ao problema de aprendizado de máquina calcula o gradiente da função para todo o conjunto de dados de treinamento de uma só vez, o que gera um enorme custo computacional, pois na soma em (2.1) ou (2.2) há muitos termos. O algoritmo de descida de gradiente

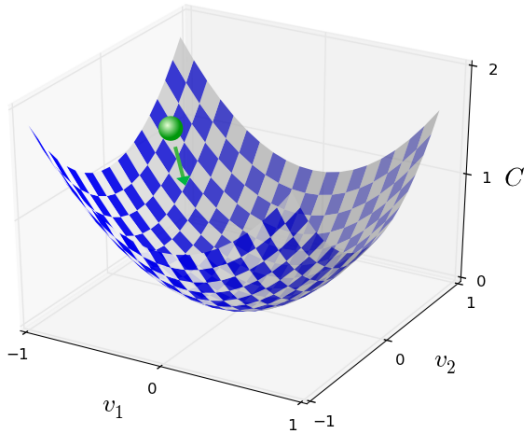


Figura 7: Método do gradiente. Fonte: (NIELSEN, 2015)

computa repetidamente o gradiente ∇f e move-se na direção oposta com objetivo de chegar no vale, como ilustrado pela bola verde no cando superior da Figura 7.

Apesar do método do gradiente clássico não ser aplicado ao nosso problema de interesse devido ao alto custo do cálculo do gradiente da soma na função objetivo, os métodos mais utilizados no contexto consistem em variações dele. Essas variações visam minimizar o custo do cálculo de todos os gradientes da soma dando passos usando alguns poucos gradientes da soma em cada iteração. O capítulo seguinte traz uma revisão dos principais algoritmos.

3 Método do Gradiente Estocástico

O método SGD (*Stochastic Gradient Descent*) (BOTTOU; CURTIS; NOCEDAL, 2018) é uma adaptação do método do gradiente clássico quando só temos à disposição aproximações do gradiente da função a ser minimizada, por exemplo, quando a função depende de uma distribuição de probabilidade desconhecida e a aproximamos por gradientes sobre uma amostra. Esse é o nosso caso: a equação (2.1) é uma aproximação do risco esperado. Por outro lado, essa soma é, em geral, muito longa. O passo do gradiente clássico (que usa os gradientes de todos os termos da soma) é pouco efetivo e acaba sendo custoso. Para acelerar o método, apenas partes do conjunto de treinamento são utilizadas a cada iteração, contrastando com o método do gradiente. A escolha dos dados de treinamento deve ser realizada de forma o mais independente possível de uma iteração para outra, sob pena de perder aleatoriedade, enviesando a busca. Aqui entram escolhas aleatórias de parte dos gradientes da soma em (2.1) a cada iteração. Isto é, ao invés de considerar todos os dados no conjunto de dados (a soma completa), selecionamos apenas um subconjunto distinto por iteração, até que o conjunto todo seja percorrido. Este subconjunto pode ser apenas um único termo ou um conjunto pequeno dos dados chamados mini-lotes (*mini-batches*). Ao utilizarmos apenas uma parte dos gradientes da soma em (2.1), não há garantia de descenso local da função. A Figura 8[1] ilustra a diferença do comportamento entre o SGD em relação ao método do gradiente clássico. Ressaltamos, no entanto, que mesmo sem a garantia de descenso a cada iteração, o SGD costuma ser eficiente para minimizar grandes somas dado que cada iteração é muito barata. Os mini-lotes são utilizados quando há uma abundância de dados, caso contrário sua efetividade diminui. Por exemplo, caso tenhamos no conjunto de treinamento menos de 2000 dados, utilizar o método do gradiente clássico, também conhecido como (*batch gradient descent*), pode ser mais viável. As escolhas típicas do tamanho do mini-lote são na ordem de 2^N, onde $N \in \mathbb{N}$, normalmente para redes com menos de 100mil dados temos mini-lote de tamanho

[1] `https://www.coursera.org/learn/deep-neural-network?specialization=deep-learning`

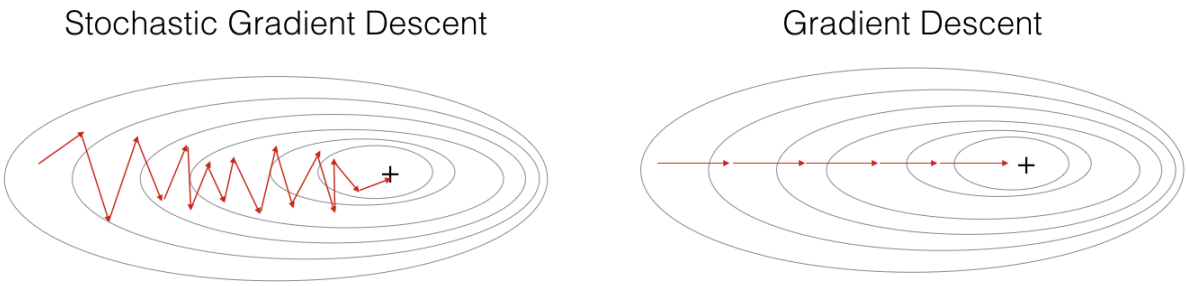


Figura 8: Comparação entre SGD e GD. Fonte: Stanford's Andrew Ng's MOOC Deep Learning Course

$2^5, 2^6$ e 2^7. O SGD é descrito no Algoritmo 1.

Algoritmo 1 Método do gradiente estocástico – SGD – para aprendizado de máquina

1: Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Faça $k = 0$.
2: **para** $k = 0, \ldots, k_{\max} - 1$ **faça**
3: Divida aleatoriamente os m dados de treinamento em mini-lotes $B_0, \ldots, B_{p-1}$
4: $z^{k,0} = z^k$
5: **para** $i = 0, \ldots, p-1$ **faça**
6: Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$, onde C_j é o termo em (2.1) referente à $j \in B_i$
7: Incremente os pesos e vieses: $z^{k,i+1} = z^{k,i} - \eta \nabla C_{B_i}(z^{k,i})$
8: **fim para**
9: Atualize os pesos e vieses: $z^{k+1} = z^{k,p}$
10: **fim para**
11: Retorne os pesos e vieses finais $z^{k_{\max}} = (w^{k_{\max}}, b^{k_{\max}})$.

Cada ciclo que percorre todos os dados (laço interno) é chamado *época*. Caso $|B_k| = 1$ para todo k, chamamos o método simplesmente de gradiente estocástico. Caso $1 < |B_k| < m$ chamamos o método de SGD em mini-lote e por fim, caso $|B_k| = m$ para todo k, recaímos no método do gradiente clássico sobre (2.1). Note que a atualização dos pesos e viéses (linha 9 do algoritmo acima) é feita após o término da época, isto é, após percorrer todos os neurônios da rede.

3.1 Convergência do SGD

O método do gradiente estocástico visa minimizar a função risco. Entretanto, essa função depende de uma distribuição de probabilidade desconhecida. A estratégia é então minimizar uma aproximação sua obtida dos dados de treinamento (amostra). Portanto, a convergência é dada em probabilidade, e precisaremos de elementos mínimos da teoria de probabilidade.

3.1.1 Elementos de probabilidade

Começaremos introduzindo a definição do termo esperança, também chamado valor esperado, de uma variável aleatória. Esse valor para o caso discreto é dado pela soma dos valores que a variável aleatória pode assumir ponderado por suas probabilidades de ocorrência. Isto é, representa o valor médio "esperado" a longo prazo de uma experiência repetida várias vezes. No caso de variáveis aleatórias contínuas, ela é dada pela integral de todos os valores da função de densidade de probabilidade.

Definição 3.1 ((JAMES, 2004))**.** *Seja X uma variável aleatória qualquer e ϕ sua função de distribuição. A esperança de X é definida por*

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x\phi(x)dx,$$

quando a integral está bem definida. Analogamente, a esperança condicional de X dado y é dada por

$$\mathbb{E}[X|y] = \int_{-\infty}^{\infty} x\phi(x|y)dx,$$

onde $\phi(x|y)$ é a função de distribuição condicional.

A função definida por $\phi(y) = \mathbb{E}[X|y]$ leva y a valores reais "determinísticos" pois a esperança de uma variável aleatória X é um número real. Assim, podemos nos referir à variável aleatória $\phi(Y) = \mathbb{E}[X|Y]$, onde Y é uma variável aleatória que assume valores y. Chamamos $\mathbb{E}[X|Y]$ de *esperança condicional de X dado Y*.

Apresentaremos algumas propriedades da esperança condicional que serão utilizadas na próxima seção. Para não fugir do foco principal, omitiremos as provas dessas propriedades. Elas podem ser obtidas da definição utilizando as propriedades de integral. Veja ainda (JAMES, 2004).

Propriedades 3.1. *Propriedades da esperança condicional:*

P1. *Propriedade fundamental:* $\mathbb{E}\big[\mathbb{E}[X|Y]\big] = \mathbb{E}[X]$*;*

P2. *Linearidade: para quaisquer variáveis aleatórias X_1 e X_2, temos*

(i) $\mathbb{E}[aX_1|Y] = a\mathbb{E}[X_1|Y], \qquad \forall a \in \mathbb{R}$ *constante,*

(ii) $\mathbb{E}[aX_1 + bX_2|y] = a\mathbb{E}[X_1|Y] + b\mathbb{E}[X_2|Y], \qquad \forall a, b \in \mathbb{R}$ *constantes,*

desde que o termo à direita esteja bem definido;

P3. *Se $X_1 \leq X_2$, então $\mathbb{E}[X_1|Y] \leq \mathbb{E}[X_2|Y]$, contanto que as esperanças estejam bem definidas;*

P4. *Se $X = c$ para alguma constante $c \in \mathbb{R}$, então $\mathbb{E}[X|Y] = c$.*

Ao aplicarmos o conceito de esperança, é útil verificarmos o quão distantes nossos valores de X estão de sua média. Para isso utilizaremos a medida de dispersão conhecida como variância. Quanto maior o valor da variância, mais dispersos são os dados de sua média e quanto menor for esse valor, menos dispersos da média. A variância é representada pela seguinte definição no caso de variáveis aleatórias contínuas:

Definição 3.2. *Seja X uma variável aleatória com função de distribuição ϕ. A variância de X é definida por*

$$\mathbb{V}[X] = \mathbb{E}\big[(X - \mathbb{E}[X])^2\big] = \int_{-\infty}^{\infty} [x - \mathbb{E}[X]]^2 \phi(x) dx.$$

Como resultado direto da definição, temos a seguinte proposição

Proposição 3.1. *Para cada variável X, a variância pode ser escrita como,*

$$\mathbb{V}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

Demonstração. De fato,

$$\begin{aligned}
\mathbb{V}[X] &= \mathbb{E}\big[(X - \mathbb{E}[X])^2\big], \\
&= \int_{-\infty}^{\infty} \big[x - \mathbb{E}[X]\big]^2 \phi(x) dx, \\
&= \int_{-\infty}^{\infty} \big[x^2 - 2x\mathbb{E}[X] + \mathbb{E}[X]^2\big] \phi(x) dx, \\
&= \int_{-\infty}^{\infty} x^2 \, \phi(x) dx - 2\mathbb{E}[X] \int_{-\infty}^{\infty} x \, \phi(x) dx + \mathbb{E}[X]^2 \int_{-\infty}^{\infty} \phi(x) dx, \\
&= \mathbb{E}[X^2] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2, \\
&= \mathbb{E}[X^2] - \mathbb{E}[X]^2.
\end{aligned}$$

□

A proposição anterior motiva a definição da *variância condicional de X dado Y* como

$$\mathbb{V}[X|Y] = \mathbb{E}[X^2|Y] - \mathbb{E}[X|Y]^2.$$

Aqui, se X for um vetor de variáveis aleatórias então $X^2 = X^T X$.

No contexto de aprendizagem de máquina, queremos

$$\text{minimizar } R(w) = \mathbb{E}[F(g(x,w),y)] = \int F(g(x,w),y)d\phi(x,y),$$

conhecido como **risco esperado** $R(w)$, onde a distribuição conjunta ϕ é desconhecida, e a única informação que temos em mãos são pares entrada/saída (amostras de (x,y)). Portanto, nosso objetivo passa a ser minimizar o risco empírico:

$$\text{minimizar } R_n(w) = \frac{1}{n}\sum_{i=1}^{n} F(g(x_i,w),y_i).$$

O **risco empírico** $R_n(w)$ (ou função de erro) é uma estimativa do risco esperado (de fato, a lei dos grandes números diz que $|R(w) - R_n(w)| \to 0$ quando $n \to \infty$). Desta forma, a iteração do método do gradiente estocástico é dada por

$$x^{k+1} = x^k - \eta_k g(w_k),$$

onde $g(w_k)$ é uma estimativa aleatória de $\nabla R_n(w_k)$ condicionada à escolha w^k. A ideia é, a cada novo iterando w^k gerado pelo método, estimar o que seria a direção do gradiente clássico para gerar o próximo iterando w^{k+1}. Uma maneira de garantir isso é que a escolha de $g(w^k)$ não seja enviesada, isto é,

$$\mathbb{E}[g(w_k)|w_k] = \nabla R_n(w_k).$$

Em outras palavras, a condição acima diz que em média escolhemos ∇R_n, ainda que não o calculemos explicitamente. A análise de convergência que apresentaremos na próxima seção trabalha com hipóteses mais gerais, que pedem apenas que $-g(w_k)$ seja uma direção de descida para R_n a partir de w_k. Neste sentido, ela engloba estratégias além da descida pelo gradiente, por exemplo, esquemas quase-Newton.

Dada essa breve introdução aos elementos de probabilidade, analisaremos a convergência do método SGD em um caso particular simples.

3.1.2 Teorema de convergência

Nossa abordagem é construída sobre a suposição da função objetivo F ser suave e fortemente convexa. A análise segue (BOTTOU; CURTIS; NOCEDAL, 2018), onde F pode representar risco esperado ou empírico.

Hipótese 3.1. *(Gradiente Lipschitz contínuo) F é continuamente diferenciável e seu*

gradiente é Lipschitz contínuo com constante de Lipschitz $L > 0$, i.e.,

$$||\nabla F(v) - \nabla F(w)|| \leq L||v - w||$$

A Hipótese 3.1 garante que o ∇F não mude rapidamente em relação a v, fazendo com que o gradiente forneça um bom indicador sobre o tamanho do passo que deve realizar para diminuir F. Essa hipótese é essencial para análise de convergência da maioria dos métodos que se baseiam em gradiente. Uma consequência importante da Hipótese 3.1 é que

$$F(v) \leq F(w) + \nabla F(w)^T(v - w) + \frac{L}{2}||v - w||^2. \tag{3.1}$$

De fato, definindo $h(t)$ como sendo $h(t) = F(w + t(v - w))$, calculando em $t = 1$ e $t = 0$, temos $h(1) = F(v)$ e $h(0) = F(w)$. Pelo teorema fundamental do cálculo,

$$\int_a^b f(t)dt = F(b) - F(a),$$

onde f é uma antiderivada de F. Sabemos que $f(t) = h'(t) = \nabla F(w + t(v - w))^T(v - w)$, logo

$$\int_0^1 h'(t)dt = F(1) - F(0) = F(v) - F(w).$$

Isolando $F(w)$, obtemos

$$F(w) = F(v) + \int_0^1 h'(t)dt = F(v) + \int_0^1 \nabla F(w + t(v - w))^T(v - w)dt.$$

Somando na integral $\pm\nabla F(w)^T(v - w)$,

$$F(w) = F(v) + \int_0^1 \nabla F(w + t(v - w))^T(v - w) - \nabla F(w)^T(v - w) + \nabla F(w)^T(v - w)dt.$$

Usando a propriedade que a integral da soma é igual à soma das integrais,

$$\begin{aligned} F(w) &= F(v) + \int_0^1 \nabla F(w + t(v - w))^T(v - w) - \nabla F(w)^T(v - w)dt + \int_0^1 \nabla F(w)^T(v - w)dt, \\ &= F(v) + \nabla F(w)^T(v - w) + \int_0^1 \nabla F(w + t(v - w))^T(v - w) - \nabla F(w)^T(v - w)dt. \end{aligned}$$

Colocando $(v - w)$ em evidência.

$$F(w) = F(v) + \nabla F(w)^T(v - w) + \int_0^1 [\nabla F(w + t(v - w)) - \nabla F(w)]^T(v - w)dt.$$

Agora, aplicando norma em toda a integral, obtemos

$$\begin{aligned} F(w) &\leq F(v) + \nabla F(w)^T(v-w) + \int_0^1 ||[\nabla F(w+t(v-w)) - \nabla F(w)]^T(v-w)||dt \\ &\leq F(v) + \nabla F(w)^T(v-w) + \int_0^1 ||[\nabla F(w+t(v-w)) - \nabla F(w)]^T|| \cdot ||(v-w)||dt. \end{aligned}$$

Pela Hipótese 3.1, temos

$$F(w) \leq F(v) + \nabla F(w)^T(v-w) + \int_0^1 L||t(v-w)|| \cdot ||(v-w)||dt.$$

Colocando para fora da integral valores que não dependem de t,

$$\begin{aligned} F(w) &= F(v) + \nabla F(w)^T(v-w) + L||(v-w)|| \int_0^1 ||t(v-w)||dt \\ &= F(v) + \nabla F(w)^T(v-w) + L||(v-w)|| \int_0^1 |t| \cdot ||(v-w)||dt, \\ &= F(v) + \nabla F(w)^T(v-w) + L||(v-w)||^2 \int_0^1 |t|dt, \\ &= F(v) + \nabla F(w)^T(v-w) + L||(v-w)||^2 \cdot \frac{1}{2}, \\ &= F(v) + \nabla F(w)^T(v-w) + \frac{L}{2}||(v-w)||^2. \end{aligned}$$

Por fim, chegamos a desigualdade esperada,

$$F(w) \leq F(v) + \nabla F(w)^T(v-w) + \frac{L}{2}||(v-w)||^2.$$

A partir de (3.1), obtemos o primeiro lema fundamental.

Lema 3.1 (Lema 4.2 de (BOTTOU; CURTIS; NOCEDAL, 2018))**.** *Sob a Hipótese 3.1, as iterações do SGD satisfazem a seguinte desigualdade para todo $k \in \mathbb{N}$:*

$$\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\alpha_k \nabla F(w_k)^T \mathbb{E}[g(w_k)|w_k] + \frac{1}{2}\alpha_k^2 L \mathbb{E}[||g(w_k)||^2|w_k] \tag{3.2}$$

Demonstração. Pela Hipótese 3.1, temos

$$F(w_{k+1}) - F(w_k) \leq \nabla F(w_k)^T(w_{k+1} - w_k) + \frac{L}{2}||w_{k+1} - w_k||^2. \tag{i}$$

No método,

$$w_{k+1} - w_k = -\alpha_k g(w_k). \tag{ii}$$

Usando (ii) em (i), obtemos

$$\begin{aligned} F(w_{k+1}) - F(w_k) &\leq \nabla F(w_k)^T(-\alpha_k g(w_k)) + \frac{L}{2}|| - \alpha_k g(w_k)||^2 \\ &\leq -\alpha_k \nabla F(w_k)^T g(w_k) + \frac{L\alpha_k^2}{2}||g(w_k)||^2. \end{aligned}$$

Tomando as esperanças condicionais na inequação acima, obtemos

$$\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\alpha_k \nabla F(w_k)^T \mathbb{E}[g(w_k)|w_k] + \frac{1}{2}\alpha_k^2 L \mathbb{E}[||g(w_k)||^2|w_k],$$

como queríamos. □

Tendo em vista o Lema 3.1, a fim de garantir a descida de F precisamos de requisitos adicionais sobre os termos do lado direito de (3.2). Note que se a escolha de $g(w_k)$ for sem viés, isto é, se $\mathbb{E}[g(w_k)|w_k] = \nabla F(w_k)$, então o termo com α_k torna-se $-\alpha_k||\nabla F(w_k)||^2 < 0$ e obtemos a seguinte inequação:

$$\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\alpha_k||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 L \mathbb{E}[||g(w_k)||^2|w_k]. \tag{3.3}$$

Neste caso, o último termo com α_k deve ser limitado por algo relacionado com a norma do gradiente de F em w_k. Para tal, restringimos a variância de $g(w_k)$. Lembre-se da seção anterior que a variância condicional de $g(w_k)$ dado w_k é

$$\mathbb{V}[g(w_k)|w_k] \coloneqq \mathbb{E}[||g(w_k)||^2|w_k] - ||\mathbb{E}[g(w_k)|w_k]||^2. \tag{3.4}$$

Geralmente, a escolha de $g(w_k)$ deve ser de tal forma que $\mathbb{E}[g(w_k)|w_k]$ tenha produto suficientemente positivo com $\nabla F(w_k)$. Assim, impomos a seguinte hipótese:

Hipótese 3.2. *A função objetivo F e SGD satisfazem as seguintes condições:*

(a) F é limitada inferiormente por F_{inf};

(b) Existem escalares $\mu_G \geq \mu > 0$ tais que, $\forall k \in \mathbb{N}$,

$$\nabla F(w_k)^T \mathbb{E}[g(w_k)|w_k] \geq \mu||\nabla F(w_k)||^2, \tag{3.5}$$

$$||\mathbb{E}[g(w_k)|w_k]|| \leq \mu_G||\nabla F(w_k)||. \tag{3.6}$$

(c) Existem escalares $M \geq 0$ e $M_v \geq 0$ tais que, $\forall k \in \mathbb{N}$,

$$\mathbb{V}[g(w_k)|w_k] \leq M + M_v||\nabla F(w_k)||^2 \tag{3.7}$$

A primeira condição, a Hipótese 3.2 (a), requer apenas que F seja limitada inferiormente. A segunda condição, a Hipótese 3.2 (b), afirma que a esperança condicional

$\mathbb{E}[g(w_k)|w_k]$ é uma direção de descida suficiente para F a partir de w_k com uma norma comparável à norma do gradiente, não sendo necessário obtermos um gradiente em média. A terceira condição, a Hipótese 3.2 (c), afirma que a variância condicional $\mathbb{V}[g(w_k)|w_k]$ é restrita por escalares positivos M e M_v de tal forma que, caso F seja uma função quadrática convexa, a variância pode diferir de zero em qualquer ponto estacionário para F e pode crescer quadraticamente em qualquer direção.

Pela Hipótese 3.2 e por (3.4),

$$\mathbb{E}[||g(w_k)||^2|w_k] \leq M + M_G||\nabla F(w_k)||^2, \text{ onde } M_G \coloneqq M_v + \mu_G^2 \geq \mu^2 > 0. \tag{3.8}$$

O seguinte lema fundamental baseia-se no **Lema 3.1** sob as condições adicionais estabelecidas na **Hipótese 3.2**.

Lema 3.2 (Lema 4.4 de (BOTTOU; CURTIS; NOCEDAL, 2018))**.** *Sobre as hipóteses 3.1 e 3.2, as iterações do SGD satisfazem as seguintes inequações para todo* $k \in \mathbb{N}$*:*

$$\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\mu\alpha_k||\nabla F(w_k)||^2 + \frac{L\alpha_k^2}{2}\mathbb{E}[||g(w_k)||^2|w_k] \tag{3.9}$$

$$\leq -\Big(\mu - \frac{1}{2}\alpha_k L M_G\Big)\alpha_k||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 LM. \tag{3.10}$$

Demonstração. Provamos (3.9) pelo Lema 3.1 e por (3.5) multiplicada por -1:

$$\begin{aligned}\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) &\leq -\alpha_k \nabla F(w_k)^T \mathbb{E}[g(w_k)|w_k] + \frac{1}{2}\alpha_k^2 L\mathbb{E}[||g(w_k)||^2|w_k] \\ &\leq -\alpha_k\mu||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 L\mathbb{E}[||g(w_k)||^2|w_k]. \end{aligned} \tag{i}$$

Agora, provaremos (3.10) pela inequação (3.7), pela hipótese 3.2 e por (3.4). De (3.4), temos

$$\mathbb{E}[||g(w_k)||^2|w_k] = \mathbb{V}[g(w_k)|w_k] + ||\mathbb{E}[g(w_k)|w_k]||^2. \tag{ii}$$

Substituindo (ii) em (i) obtemos

$$\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\alpha_k\mu||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 L(\mathbb{V}[g(w_k)|w_k] + ||\mathbb{E}[g(w_k)|w_k]||^2).$$

De (3.7), temos que

$$\begin{aligned}&\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\alpha_k\mu||\nabla F(w_k)||^2 + \\ &\frac{1}{2}\alpha_k^2 L(M + M_v||\nabla F(w_k)||^2 + ||\mathbb{E}[g(w_k)|w_k]||^2).\end{aligned}$$

Pela inequação (3.6) ao quadrado, temos

$$\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) \leq -\alpha_k \mu ||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 L(M + M_v||\nabla F(w_k)||^2 + \mu_G^2||\nabla F(w_k)||^2)$$

Finalmente, chamando $M_G = M_v + \mu_G^2$, obtemos

$$\begin{aligned}\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) &\leq -\alpha_k \mu ||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 L(M + M_G||\nabla F(w_k)||^2) \\ &\leq -(\mu - \frac{1}{2}\alpha_k L M_G)\alpha_k ||\nabla F(w_k)||^2 + \frac{1}{2}\alpha_k^2 LM.\end{aligned}$$

□

Agora analisaremos o método SGD no contexto de minimizar uma função objetivo fortemente convexa.

Hipótese 3.3. *(Fortemente Convexa): A função $F : \mathbb{R}^d \to \mathbb{R}$ é fortemente convexa, isto é, existe uma constante $c > 0$ tal que*

$$F(w) \geq F(v) + \nabla F(v)^T(w - v) + \frac{1}{2}c||w - v||^2$$

Neste caso, F tem um único minimizador, que denotaremos por $w^* \in \mathbb{R}$ com $F^* := F(w^*)$.

Um fato da análise convexa é que, sob a hipótese 3.3, pode-se limitar a lacuna de otimalidade em um determinado ponto em termos da norma ao quadrado do gradiente naquele ponto, isto é:

$$2c(F(w) - F^*) \leq ||\nabla F(w)||^2. \tag{3.11}$$

Dadas as hipóteses acima, podemos enunciar e provar a convergência do SGD com passo fixo em funções fortemente convexas.

Teorema 3.1 (Teorema 4.6 de (BOTTOU; CURTIS; NOCEDAL, 2018))**.** *Sob as hipóteses 3.1, 3.2 e 3.3 (com $F_{inf} = F^*$), suponha que o método SGD com passo fixo ($\alpha_k = \alpha$ para todo $k \in \mathbb{N}$) satisfaça*

$$0 < \alpha \leq \frac{\mu}{LM_G}. \tag{3.12}$$

A lacuna de otimalidade esperada satisfaz a seguinte desigualdade:

$$\mathbb{E}[F(w_{k+1}) - F^*] \leq \frac{\alpha LM}{2c\mu} + (1 - \alpha c\mu)^{k-1}\left[F(w_1) - F^* - \frac{\alpha LM}{2c\mu}\right] \xrightarrow{k\to\infty} \frac{\alpha LM}{2c\mu}.$$

Demonstração. Pelo Lema 3.2 com (3.12) e (3.11), temos

$$\begin{aligned}\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) &\leq -(\mu - \frac{1}{2}\alpha LM_G)\alpha||\nabla F(w_k)||^2 + \frac{1}{2}\alpha^2 LM \\ &\leq -(\mu - \frac{1}{2}\frac{\mu}{LM_G}LM_G)\alpha||\nabla F(w_k)||^2 + \frac{1}{2}\alpha^2 LM \\ &\leq -\frac{1}{2}\mu\alpha||\nabla F(w_k)||^2 + \frac{1}{2}\alpha^2 LM. \qquad \text{(i)}\end{aligned}$$

Multiplicando (3.11) por -1 em ambos os lados, obtemos

$$-||\nabla F(w_k)||^2 \leq -2c(F(w_k) - F^*). \qquad \text{(ii)}$$

Agora, de (ii) em (i) vem

$$\begin{aligned}\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) &\leq \frac{1}{2}\mu\alpha(-2c(F(w_k) - F^*)) + \frac{1}{2}\alpha^2 LM \\ &\leq -\mu\alpha c(F(w_k) - F^*) + \frac{1}{2}\alpha^2 LM.\end{aligned}$$

Subtraindo F^* em ambos os lados,

$$\begin{aligned}&\mathbb{E}[F(w_{k+1})|w_k] - F(w_k) - F^* \leq -\mu\alpha c(F(w_k) - F^*) + \frac{1}{2}\alpha^2 LM - F^* \\ \implies &\mathbb{E}[F(w_{k+1})|w_k] - F^* \leq -\mu\alpha c(F(w_k) - F^*) + \frac{1}{2}\alpha^2 LM - F^* + F(w_k) \\ &\leq (1 - \mu\alpha c)(F(w_k) - F^*) + \frac{1}{2}\alpha^2 LM.\end{aligned}$$

Tomando as esperanças totais,

$$\mathbb{E}[F(w_{k+1}) - F^*] \leq (1 - \mu\alpha c)\mathbb{E}(F(w_k) - F^*) + \frac{1}{2}\alpha^2 LM.$$

Subtraindo a constante $\alpha LM/2c\mu$ em ambos os lados,

$$\begin{aligned}\mathbb{E}[F(w_{k+1}) - F^*] - \frac{\alpha LM}{2c\mu} &\leq (1 - \mu\alpha c)\mathbb{E}(F(w_k) - F^*) + \frac{1}{2}\alpha^2 LM - \frac{\alpha LM}{2c\mu} \\ &= (1 - \mu\alpha c)\left(\mathbb{E}[F(w_k) - F^*] - \frac{\alpha LM}{2c\mu}\right). \qquad (3.13)\end{aligned}$$

Por (3.12)

$$0 < \alpha c\mu \leq \frac{c\mu^2}{LM_G}$$

e por (3.8)

$$0 < \alpha c\mu \leq \frac{c\mu^2}{LM_G} \leq \frac{c\mu^2}{L\mu^2} = \frac{c}{L} \leq 1$$

O resultado segue aplicando (3.13) repetidas vezes:

$$\begin{aligned}
\mathbb{E}[F(w_{k+1}) - F^*] - \frac{\alpha LM}{2c\mu} &\leq (1-\mu\alpha c)\left(\mathbb{E}[F(w_k) - F^*] - \frac{\alpha LM}{2c\mu}\right) \\
&\leq (1-\mu\alpha c)\left[(1-\mu\alpha c)\left(\mathbb{E}[F(w_{k-1}) - F^*] - \frac{\alpha LM}{2c\mu}\right)\right] \\
&= (1-\mu\alpha c)^2\left(\mathbb{E}[F(w_{k-1}) - F^*] - \frac{\alpha LM}{2c\mu}\right) \\
&\leq (1-\mu\alpha c)^2\left[(1-\mu\alpha c)\left(\mathbb{E}[F(w_{k-2}) - F^*] - \frac{\alpha LM}{2c\mu}\right)\right] \\
&= (1-\mu\alpha c)^3\left(\mathbb{E}[F(w_{k-2}) - F^*] - \frac{\alpha LM}{2c\mu}\right) \\
&\dots \\
&\leq (1-\mu\alpha c)^k\left(\mathbb{E}[F(w_{k-(k-1)}) - F^*] - \frac{\alpha LM}{2c\mu}\right) \\
&\leq (1-\mu\alpha c)^{k-1}\left(F(w_1) - F^* - \frac{\alpha LM}{2c\mu}\right).
\end{aligned}$$

□

O teorema anterior mostra que se o tamanho do passo não for muito grande, então, em média, a sequência de valores da função $F(w_k)$ fica próxima do valor ótimo. Note que devido ao tamanho do passo ser fixo, ao se aproximar do valor ótimo o método pode oscilar. Ou seja, não há garantia que alcançemos F^*. Para contornar esse problema foram propostos métodos com passo variável. Na seção seguinte veremos alguns desses métodos.

3.2 Variantes do SGD

Há vários métodos na literatura usados no treinamento de redes neurais. Eles se diferenciam basicamente por dois fatores: (i) cálculo da taxa de aprendizado que representa o tamanho do passo, que pode se adaptar a cada iteração; (ii) uso do chamado "momento" na direção de busca, que consiste na combinação de $-\nabla f$ com a direção da iteração anterior.

3.2.1 SGD com momento (SGDM)

O SGDM, proposto por Sutskever et al. (2013), agrega à direção de busca do método a ideia do momento linear presente na física, visando assim simular a tendência de um corpo em trajetória retilínea permanecer em movimento. É proposto com o intuito de

contornar um dos problemas presentes no SGD: convergência a mínimos locais ruins e pontos de sela. O SGDM para o nosso caso de interesse é descrito no Algoritmo 2.

Algoritmo 2 SGD com Momento – SGDM

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $V^{-1} = 0$.
para $k = 0, \ldots, k_{\max} - 1$ **faça**
    Divida aleatoriamente os m dados de treinamento em mini-lotes $B_0, \ldots, B_{p-1}$
    $z^{k,0} = z^k, \quad V^{k,-1} = V^k$
    para $i = 0, \ldots, p-1$ **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$
        Calcule o momento: $V^{k,i} = \beta V^{k,i-1} + (1-\beta)\nabla C_{B_k}(z^{k,i})$
        Incremente os pesos e vieses: $z^{k,i+1} = z^{k,i} - \eta V^{k,i}$
    fim para
    Atualize os pesos, vieses e o momento: $z^{k+1} = z^{k,p}$, $V^k = V^{k,p}$
fim para
Retorne os pesos e vieses finais $z^{k_{\max}} = (w^{k_{\max}}, b^{k_{\max}})$.

O escalar β é um parâmero em $[0, 1)$. V^k pode ser interpretado como a velocidade do objeto no ponto z^k. Note que $\beta = 0$ recai no SGD sem momento (Algoritmo 1), enquanto que se $\beta > 0$, V^k leva em consideração a velocidade na iteração anterior.

3.2.2 ADAGrad

O ADAGrad (do inglês *Adaptative Gradient*), proposto por Duchi, Hazan e Singer (2011), é um método com taxa de aprendizado adaptativa, isto é, a taxa muda durante o processo de minimização. Tem o intuito de contornar um dos problemas dos métodos anteriores, a taxa de aprendizagem constante que pode levar o método a oscilar próximo à solução. Esse método ajusta a taxa de aprendizagem com base nos gradientes armazenados no decorrer das iterações, realizando atualizações maiores ou menores conforme a frequência dos parâmetros. Além de ser adequado para lidar com dados esparsos. Assim, ele evita o ajuste manual da taxa de aprendizagem e acelera a convergência. O seu algoritmo é descrito no Algoritmo 3:

Algoritmo 3 ADAGrad

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $G^{-1} = 0$.
para $k = 0, \ldots, k_{\max} - 1$ **faça**
    Divida aleatoriamente os m dados de treinamento em mini-lotes $B_0, \ldots, B_{p-1}$
    $z^{k,0} = z^k, \quad G^{k,-1} = G^k$
    para $i = 0, \ldots, p-1$ **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$
        Atualize a soma das normas dos gradientes: $G^{k,i} = G^{k,i-1} + \left\| \nabla C_{B_i}(z^{k,i}) \right\|^2$
        Incremente os pesos e vieses: $z^{k,i+1} = z^{k,i} - \frac{\eta}{\sqrt{G+\epsilon}} \nabla C_{B_i}(z^{k,i})$
    fim para
    Atualize os pesos, vieses e gradiente acumulado: $z^{k+1} = z^{k,p}$, $G^k = G^{k,p}$
fim para
Retorne os pesos e vieses finais $z^{k_{\max}} = (w^{k_{\max}}, b^{k_{\max}})$.

G é o quadrado da norma dos gradientes acumulados até a iteração atual. O escalar η pode ser visto como a taxa de aprendizado "referência", e é escalada de acordo com os tamanhos dos gradientes calculados. Note que à medida que o método avança, G aumenta e logo o passo diminui. O escalar ϵ serve como estabilizador quando $G \approx 0$.

3.2.3 RMSProp

O RMSProp (do inglês *Root Means Square Propagation*) é um algoritmo não publicado proposto por Tieleman, Hinton et al. (2012), e tem o intuito de melhorar o ADAGrad. Ele não acumula todos os gradientes; pelo contrário, considera somente os gradientes em uma certa janela de iterações recentes. Contorna o problema de aprendizagem presente no final do ADAGrad, que conforme o passar das iterações, o gradiente se torna muito grande levando a taxa de aprendizagem à zero rapidamente. O RMSProp é descrito no Algoritmo 4.

Algoritmo 4 RMSProp

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $G^{-1} = 0$.
para $k = 0, \ldots, k_{\max} - 1$ **faça**
    Divida aleatoriamente os m dados de treinamento em mini-lotes $B_0, \ldots, B_{p-1}$
    $z^{k,0} = z^k, \quad G^{k,-1} = G^k$
    para $i = 0, \ldots, p-1$ **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$
        Atualize a soma das normas dos gradientes: $G^{k,i} = \beta G^{k,i-1} + (1 - \beta) \left\| \nabla C_{B_i}(z^{k,i}) \right\|^2$
        Incremente os pesos e vieses: $z^{k,i+1} = z^{k,i} - \frac{\eta}{\sqrt{G+\epsilon}} \nabla C_{B_i}(z^{k,i})$
    fim para
    Atualize os pesos, vieses e gradiente acumulado: $z^{k+1} = z^{k,p}$, $G^k = G^{k,p}$
fim para
Retorne os pesos e vieses finais $z^{k_{\max}} = (w^{k_{\max}}, b^{k_{\max}})$.

Como podemos observar, a construção do algoritmo é semelhante ao ADAGrad com a entrada de β na soma das normas dos gradientes G.

3.2.4 Adam

O Adam (do inglês *Adaptive Moment Estimation*), proposto por Kingma e Ba (2015), é um método que utiliza taxa de aprendizagem adaptativa para cada parâmetro. Ele mescla as ideias principais dos algoritmos RMSProp, SGD com Momento e ADAGrad, isto é, emprega taxa adaptativa escalada pelas normas dos gradientes calculados e momento. Funciona bem com gradientes esparsos. As médias móveis do Adam são estimadas pelo primeiro momento (a média) e pelo segundo momento bruto (a variância não centrada) do gradiente, no nosso caso eles são $V^{k,i}$ e $G^{k,i}$. Seu processo é relativamente estável e amplamente utilizado na literatura atual, adequado para a maioria dos problemas de otimização não convexa com grande conjuntos de dados. Algumas das vantagens do Adam são que seus tamanhos de passos são limitados pelo hiperparâmetro η, trabalha com gradientes esparsos e naturalmente executa uma forma de recozimento do tamanho do passo (*step size annealing*).

Algoritmo 5 Adam

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $G^{-1} = 0$ e $V^{-1} = 0$.
para $k = 0, \ldots, k_{\max} - 1$ **faça**
    Divida aleatoriamente os m dados de treinamento em mini-lotes $B_0, \ldots, B_{p-1}$
    $z^{k,0} = z^k, \quad G^{k,-1} = G^k, \quad V^{k,-1} = V^k$
    para $i = 0, \ldots, p-1$ (número de mini-lotes) **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$
        Calcule o momento: $V^{k,i} = \beta_1 V^{k,i-1} + (1 - \beta_1) \nabla C_{B_k}(z^{k,i})$
        Atualize a soma dos gradientes: $G^{k,i} = \beta_2 G^{k,i-1} + (1 - \beta_2) \left\| \nabla C_{B_i}(z^{k,i}) \right\|^2$
        Calcule $\hat{V}^{k,i} = \frac{V^{k,i}}{1-\beta_1^k}$ e $\hat{G^{k,i}} = \frac{G^{k,i}}{1-\beta_2^k}$
        Incremente os pesos e vieses: $z^{k,i+1} = z^{k,i} - \frac{\eta}{\sqrt{\hat{G}^{k,i}} + \epsilon} \hat{V}^{k,i}$
    fim para
    Atualize os pesos, vieses, gradiente acumulado e o momento:
    $z^{k+1} = z^{k,p}, \quad G^k = G^{k,p}, \quad V^k = V^{k,p}$
fim para
Retorne os pesos e vieses finais $z^{k_{\max}} = (w^{k_{\max}}, b^{k_{\max}})$.

3.2.5 Outros métodos

Existem diversos outros métodos interessantes não abordados no trabalho, tais como: *Nesterov Accelerated Gradient Descent* (NAGD), que utiliza o mesmo conceito que o SGD com Momento, com a diferença de que ao invés de atualizar o gradiente atual, realiza um passo para "adiantado" utilizando o momento anterior para atualizar o próximo gradiente; *Stochastic Average Gradient* (SAG), proposto como melhoria para a velocidade de convergência; *Stochastic Variance Reduction Gradient* (SVRG), proposto para melhorar o desempenho da otimização de modelos complexos. Esses e mais métodos podem ser vistos com mais detalhes em (SUN et al., 2019).

3.3 Um problema simples: reconhecimento de caracteres numéricos

Nesta seção apresentamos o problema de classificação de caracteres numéricos, que consiste em treinar o computador para reconhecer dígitos escritos à mão. É um problema interessante que para um ser humano, sua resolução é automática. Ao nos depararmos com um número escrito, milhares de neurônios são ativados em nosso cérebro e em

milissegundos conseguimos interpretar qual seu valor. Para um computador essa tarefa não é fácil assim, precisamos treiná-lo para realizar tal reconhecimento.

No contexto de aprendizado de máquina supervisionado, é necessário que tenhamos um banco de dados para realizar o treinamento da rede, e no caso de classificação de caracteres numéricos, precisamos de um contendo caracteres de "0" a "9" escritos à mão com suas respectivas legendas. Nosso modelo baseia-se em um grafo, a rede neural, cujo fluxo (dados da imagem) é propagado da camada de entrada até a camada de saída (a resposta da rede – 0, 1, ..., 9), conforme detalhado na Seção 2.1.

Os algoritmos de otimização que treinam a rede dependem do cálculo do gradiente (ou parte dele) da função que mede o erro entre entradas/saídas. Quando o algoritmo de otimização declara parada, a função de erro é dita minimizada e dizemos que a rede está treinada. A rede treinada é utilizada então para classificar imagens não utilizadas no treinamento como 0, 1, ... ou 9, presentes no conjunto de testes. Precisamos que a rede treinada esteja dentro de uma margem de acerto considerável, que neste caso definimos 90% (o ideal, claro, é 100% de acerto). Entretanto é necessário se preocupar com *overfitting*, um problema de "alta variância", que se caracteriza por uma taxa de acertos muito alta nos dados de treinamento, mas nos dados de teste uma taxa relativamente baixa, por exemplo 99% e 89%, respectivamente. Há também o problema de *underfitting*, um problema de "alto viés", onde as taxas de acertos nos dados de treinamento e nos dados de teste são próximas porém baixas, por exemplo, 85% e 84% respectivamente. O cenário ideal é uma alta taxa de acerto no treinamento próxima a dos dados de testes, por exemplo, $99,5\%$ e 99%, respectivamente. Caso nossa rede apresente *overfitting* e/ou *underfitting*, algumas técnicas podem ser utilizadas para contorná-los realizando um *"tradeoff"* entre a variância e o viés. São elas:

- **Viés alto (*underfitting*):**
  - Treinar com uma rede neural maior;
  - Realizar um treinamento com mais épocas (iterações do método);
  - Utilizar outra arquitetura de redes neurais;
- **Variância alta (*overfitting*):**
  - Obter mais dados;
  - Realizar uma regularização;
  - Utilizar outra arquitetura de redes neurais.

Um banco de imagem bastante conhecido é a MNIST[2] (LECUN; CORTES, 2010). Este é considerado como ponto de partida para a visão computacional, semelhante ao *"Hello World"* da programação. Ele possui ao todo 70 mil imagens de tamanho 28×28 *pixels* divididas em um conjunto de treinamento com 60 mil imagens e um conjunto de teste com 10 mil imagens, todas na escala *8-bits* de cinza. É um subconjunto do banco de dados de caracteres e formulários manuscritos do Instituto Nacional de Padrões e Tecnologia dos EUA (NIST)[3]. Na Figura 9, podemos ver alguns dígitos extraídos desse conjunto de dados.

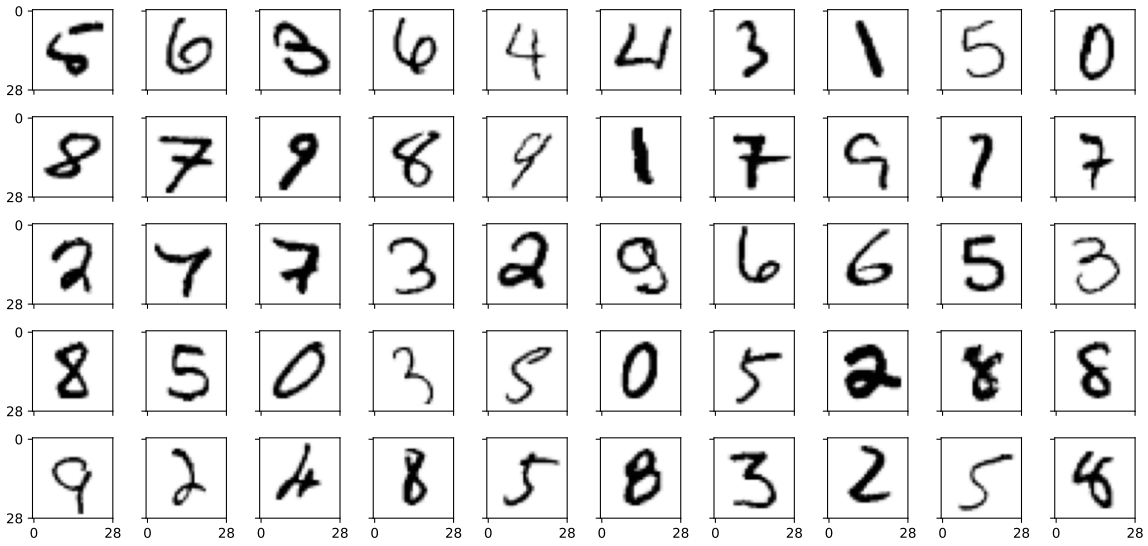


Figura 9: Dígitos presentes na MNIST.

A entrada da rede neural é dada por um vetor x de tamanho 784, que corresponde aos $28 \times 28 = 784$ *pixels* da imagem e flui até a camada de saída, o vetor $\hat{y}$ de tamanho 10. Para entender como uma camada intermediária funciona, de maneira simples, pense que a rede busca padrões para realizar o reconhecimento, onde cada neurônio presente na camada intermediária irá reconhecer partes da imagem para uma melhor resposta. Por exemplo, um neurônio será responsável apenas pelo reconhecimento do traço que difere um 8 para um 9, já outro, reconhece o traço que difere o 7 para o 1.

Os dados precisam ser normalizados antes de entrar na rede, de modo que definir que a magnitude dos valores que um recurso assume são aproximadamente os mesmos, tornando o aprendizado mais fácil. Uma possível normalização é em relação aos valores dos *pixels* presentes na imagem. Precisamos convertê-los para valores entre $[0, 1]$. Isto é necessário pois os valores dos *pixels* são inteiros entre $[0, 255]$ e podem interromper ou

[2] *"Modified National Institute of Standards and Technology"*
[3] `https://www.nist.gov/srd/nist-special-database-19`

retardar o processo de aprendizagem. Para realizar essa conversão dividimos cada *pixel* por 255. Alguns bancos de dados apresentam ruídos, geralmente dados faltantes ou mal classificados. Ao utilizarmos esses bancos, é importante tratarmos esses erros antes de treinar a rede, sob pena de classificar os dados incorretamente.

Na MNIST, precisamos converter os valores dos rótulos para podermos comparar o nosso y com o $\hat{y}$ calculado pela rede. Para isso, aplicamos a estratégia denominada *One-Hot-Encoding*, utilizada para gerar vetores binários para cada valor inteiro. O vetor é composto de zeros e tem apenas um no valor desejado. Essa abordagem é útil, pois elimina problemas de hierarquias ou ordem dos valores, tendo um neurônio na saída da rede para cada uma das classificações. No nosso problema ela transforma o rótulo da imagem para um vetor de tamanho $(10, 1)$, por exemplo, $y = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]^T$, onde 1 é inserido na posição referente ao número presente no rótulo, neste caso 2. Na Tabela 1, podemos ver um exemplo de como é observada a saída do *One-Hot-Enconding*.

(a) Dados Brutos

Número	Rótulo
1	1
3	3
2	2
1	1
4	4

(b) One-Hot-Encoder

Número	‘1’	‘2’	‘3’	‘4’
1	1	0	0	0
3	0	0	1	0
2	0	1	0	0
1	1	0	0	0
4	0	0	0	1

Tabela 1: Comparação entre os dados.

O código foi construído com a ideia de testarmos como diferentes algoritmos de otimização se comportam na classificação de caracteres. Nossa rede é dada por 784 neurônios de entrada, uma camada oculta com variação entre 100, 200 e 300 neurônios e uma camada de saída com 10 neurônios. Apenas a função de ativação sigmoid foi utilizada, e para comparação dos resultados da rede com o esperado y, tomamos a posição do maior valor presente no vetor $\hat{y}$. Não buscamos aqui os melhores resultados encontrados para o conjunto de dados, ou seja, não otimizamos nosso código para a melhor performance possível.

Além do conjunto de dados da MNIST, utilizamos 32 imagens de autoria própria para testar como a rede se comporta com dados de outra origem. Esses dados foram utilizados somente após a rede ter sido treinada com a MNIST. Convertemos cada dígito para uma imagem de tamanho 28×28 na escala de cinza, semelhante aos presentes na MNIST. A Figura 10 traz exemplos desses dígitos.

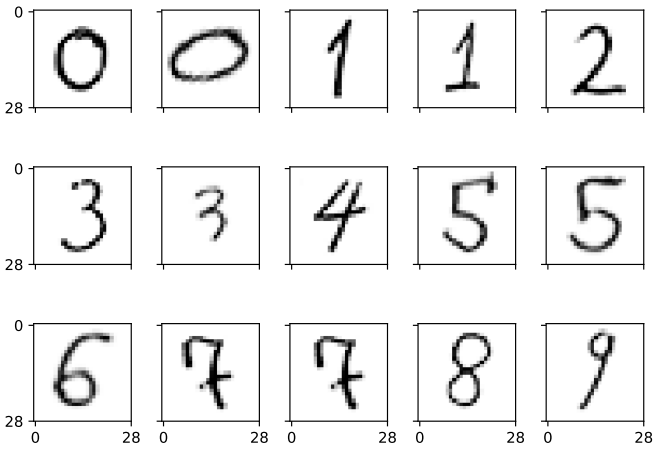


Figura 10: Dígitos próprios.

3.3.1 Resultados Numéricos

A implementação foi feita na linguagem Python versão 3.8 e para a construção do código próprio, foi tomado como referência os códigos de Michael A. Nielsen[4] e de Kyohei Sahara[5], onde a partir deles, realizamos nossas implementações próprias, passo a passo, para cada um dos métodos comentados neste capítulo. Os testes foram executados em um computador equipado com Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz 10 núcleos (20 threads), 160 GB RAM. Os pesos e vieses foram iniciados aleatoriamente no intervalo $[-1, 1]$ e de maneira uniforme com uma semente fixa para comparação justa entre os métodos.

A Tabela 2 contém os tempos de treinamento e a precisão de cada método ao ser executado com os dados de teste da MNIST, ao todo 10.000 exemplos ou 14,28% do total de imagens nunca observadas pela rede. Todos os testes foram realizados com taxa de aprendizagem $\eta = 0,001$, para SGD com Momento e RMSProp temos $\beta = 0,9$ e para Adam temos $\beta_1 = 0,9$ e $\beta_2 = 0,999$. Os neurônios da camada oculta variaram entre 300, 200 e 100, representada na coluna "NHL" na tabela.

Note que nos métodos SGD e SGD com Momento, ao aumentarmos os neurônios da camada oculta, a porcentagem de acertos diminui. Isso possivelmente se deve ao fato da taxa de aprendizagem ser constante, pois ao se aproximar do valor ótimo o passo pode ser grande e acabar se afastando. Para contornarmos esse problema, uma solução é diminuir a taxa de aprendizagem, porém acarretaria uma convergência lenta. Ao avaliarmos os

[4] `https://github.com/mnielsen/neural-networks-and-deep-learning`
[5] `https://github.com/schwalbe10/ThinkageDeepLearning`

Método	NHL	Tempo (s)	Acerto (%)
SGD	300	7.608,94	89,73
	200	5.382,59	89,84
	100	3.304,11	89,89
SGDM	300	8.827,15	94,14
	200	6.240,75	94,52
	100	3.718,60	94,65
ADAGrad	300	8.682,42	94,84
	200	6.271,35	94,72
	100	3.742,94	93,92
RMSProp	300	9.739,17	97,63
	200	6.938,55	97,38
	100	4.107,94	97,17
Adam	300	12.334,39	98,07
	200	8.548,00	97,65
	100	4.975,84	97,27

Tabela 2: Resultados dos testes na MNIST.

métodos com taxa de aprendizagem variável, esse problema não ocorre.

Na Figura 11 podemos visualizar graficamente os erros em cada estágio do processo de treinamento da rede, sendo realizado uma predição com os pesos e os dados de testes em cada época (*epochs*, as iterações externas dos métodos). O eixo vertical está em escala logarítmica para evidenciar erros próximos de zero. Os métodos foram executados com 300 neurônios na camada oculta e 100 épocas. Ao analisarmos o gráfico, o erro relativo atingido pelos métodos foi inferior a 10%, com exceção do SGD. Observa-se que SGD foi o método que pior performou, com erro consideravelmente alto nas primeiras *epochs*; ele não conseguiu reduzir de forma contínua o erro percentual e obteve uma baixa diminuição a partir da *epoch* 40. Ao contrário, Adam obteve um percentual considerável de melhora na diminuição do erro, juntamente com o método RMSProp.

Podemos concluir que o método que mais se destacou em relação à eficácia foi o Adam com 300 NHL, tendo $\approx$ 98,1% de acerto no caso de teste; em contrapartida, seu tempo de execução foi o maior (cerca de 3,43 horas). Para o melhor custo-benefício entre tempo e acertos, o método Adam com 100 NHL se destaca, com $\approx$ 97,3% de acerto no caso de teste e $\approx$ 1,38 horas de execução.

Realizamos também, testes com os 32 dígitos próprios escritos a mão, onde foram escaneados e ajustados. Apesar de os dados serem de outra fonte, completamente diferente da original, a rede conseguiu um desempenho aceitável na nossa abordagem. O treina-

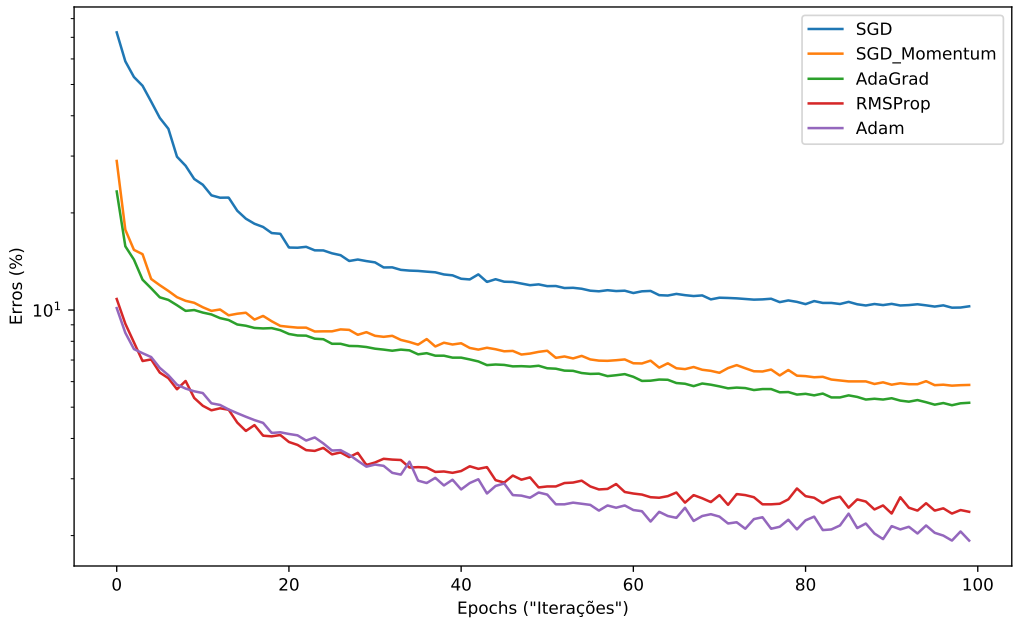


Figura 11: Gráfico comparativo entre os métodos com 300 NHL.

mento com o método Adam e 300 NHL, a rede foi capaz de acertar 23 dos 32 dígitos (71.88%). Já com SGD e 300 NHL obtivemos 20 acertos (62.50%). Nossa rede não conseguiu identificar os dígitos 9.

Na Figura 12, temos matriz de confusão que nos dá uma visualização dos acertos referentes aos números presentes no conjunto de testes da MNIST, dados retirados da rede treinada com o método SGD com 300 NHL. Nota-se que a rede se confunde mais com o número 4 e 7 ao tentar reconhecer o dígito 9 (veja a linha referente ao 9). Outra observação é que o dígito que mais ocorre erros na classificação é o dígito 5, com destaque na quantidade de previsões erradas no número 3. Isso provavelmente ocorre pois esses números são parecidos (sobretudo 4 e 9) e talvez a rede não tenha muitos dados para treinar nessa categoria de casos. Para contornar esse problema seria aconselhado entrar com mais dados semelhantes ao erro da rede para se ter um melhor reconhecimento desses números, ou ainda, utilizar técnicas mais complexas não tratadas nesse trabalho. De qualquer forma, as taxas de erro alcançadas são compatíveis com as reportadas na literatura utilizando as mesmas técnicas.

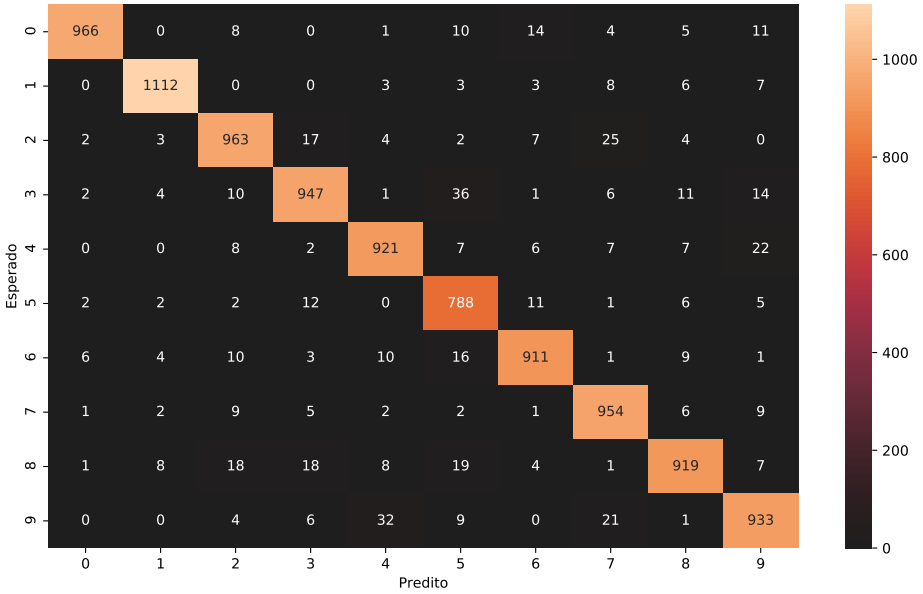


Figura 12: Matriz de confusão sobre os acertos nos dados de teste.

Feito os testes com a nossa rede implementada, abordaremos no próximo capítulo o uso de plataformas que já possuem processo de propagação e treinamento pré-implementados e otimizados. Algumas dessas plataformas são de código-fonte aberto e possibilitam agilidade na construção da rede e utilização de recursos como a utilização de GPUs. As plataformas mais conhecidas são o *TensorFlow* e *Pytorch*. No nosso trabalho, daremos ênfase no *TensorFlow*, desenvolvida pelo Google.

4 Outros problemas de classificação

Neste capítulo consideramos outras aplicações interessantes presentes na literatura para o problema de classificação. Ao contrário da Seção 3.3, usaremos aqui a plataforma *TensorFlow* (ABADI et al., 2015) em nossos testes. Ou seja, aqui utilizamos as implementações do gradiente estocástico e suas variantes já implementadas no *Tensorflow*. *TensorFlow* é uma ferramenta desenvolvida pelo Google que reúne as principais bases de dados, técnicas e métodos para aprendizado de máquina, além de oferecer um ambiente completo para modelagem e resolução de problemas de alta produtividade. Códigos para os testes realizados são reproduzidos no Apêndice B.

4.1 A plataforma *TensorFlow*

O *TensorFlow*[1] (TF) foi escrito com uma **Python API**[2] (*Application Programming Interface*) sobre um mecanismo C/C++ que o torna rápido e otimizado. O TF é bastante intuitivo e fácil de ser utilizado, sendo possível aprender o básico com os guias presentes em seu site. Não se limita apenas à redes neurais profundas, mas também suporta o aprendizado por reforço e outras técnicas. Grande parte de suas operações utilizam o *Numpy*[3], uma biblioteca bastante utilizada que suporta o processamento de vetores e matrizes multi-dimensionais junto a uma linguagem de alto nível para realização das operações com essas matrizes. Por ser de código aberto, a comunidade ajuda a melhorar cada vez mais a plataforma, e dentre todas as plataformas de aprendizagem profunda o *TensorFlow* se destaca pela ampla comunidade presente na internet. A plataforma *TensorFlow* armazena dezenas de APIs muito úteis no aprendizado de máquina. Entre

[1] https://www.tensorflow.org/

[2] Uma boa API facilita desenvolver um programa, fornecendo todos os blocos de construção. Um programador então reúne os blocos.

[3] https://numpy.org/

elas está o **Keras**[4], uma API de aprendizado profundo escrita em Python e bastante utilizada, executada por cima do *TensorFlow*.

Uma das vantagens do TF está na vasta quantidade de conjuntos de dados armazenados no módulo **Conjuntos de dados do *TensorFlow***[5] (TFDS). Além disso, a comunidade pode adicionar mais *datasets* à plataforma. A utilização desses dados são feitas através de um *pipeline*[6], criada pelo `tf.data`, uma **API** do *TensorFlow* para criar *pipelines* de dados de forma eficiente. Podemos importar os dados e utilizá-los como arquivos *numpy*, entretanto, é necessário convertê-los antes.

No Apêndice B há um código simples de um exemplo de utilização do *TensorFlow* que comentaremos a seguir. Suponhamos que queremos prever o preço de uma casa conforme a quantidade de quartos. O valor da casa é dado por \$50.000 e a cada quarto, há um acréscimo de \$50.000. Treinamos uma rede simples para conseguir prever o preço de uma casa com 7 quartos, ou seja, a rede deve prever que o preço é \$400.000. Por simplicidade, damos seus valores na escala $y = (\$50.000 \cdot x)/100.000$, onde y é o valor da casa e x a quantidade de quartos.

De maneira simples, o *TensorFlow* funciona como um auxiliar na hora da implementação. Acima, vimos um exemplo de como treinar uma rede para classificar o valor de uma casa com 7 quartos. A criação do modelo é dado pelo **Keras**. Utilizamos o `tf.keras.Sequential`[7], onde cada camada tem exatamente uma entrada e uma saída, como uma pilha linear de camadas, fluindo da esquerda para a direita. Utilizamos uma única camada densa para o nosso exemplo e nossa saída é apenas um valor. Note que não precisamos de implementar o método de otimização **SGD** (Algoritmo 1 da Seção 3), e nem a função de perda (2.1). É suficiente apenas chamarmos a função `compile`[8], passando como argumento o optimizador desejado e a função de custo. Treinamos nossa rede com a função `fit`[8], passando as entradas xs, ys e o número de épocas. Retornamos a predição em `y_new`. Por fim, chamamos nossa função com o valor a ser predito como argumento. Obtemos como resultado o valor $\approx 4{,}05$ que é igual a \$405.000.

[4]https://keras.io/guides/

[5]https://www.tensorflow.org/datasets

[6]Um *pipeline* é uma série de algoritmos encadeados e heterogêneos para realizar o processamento de um fluxo de dados.

[7]https://keras.io/api/models/sequential/

[8]https://keras.io/api/models/model_training_apis/

4.2 Técnicas avançadas

Com a introdução da plataforma *TensorFlow*, podemos entrar em outra arquitetura de redes neurais e mostrar que o aprendizado de máquina junto à aprendizagem profunda podem ir muito além de apenas reconhecer dígitos escritos à mão. Redes neurais artificiais, como a presente na Figura 2, não funcionam muito bem com imagens grandes. Na MNIST, onde temos imagens de tamanho $28 \times 28 \times 1$ (28 altura, 28 largura, 1 canal de cor[9]), a rede que recebe esses neurônios é uma rede de neurônios totalmente conectados (*Fully connected neural network*), onde a camada de entrada contém $28 * 28 * 1 = 784$ neurônios. O tamanho é grande, mas é aceitável. Suponha agora que nossa imagem de entrada possua um tamanho $500 \times 500 \times 3$. Nossa rede precisaria de $500 * 500 * 3 = 750.000$ neurônios na camada de entrada. Para minimizar essa grande quantidade de parâmetros, utilizamos a arquitetura de redes neurais convolucionais.

4.2.1 Redes Neurais Convolucionais

Como foi introduzida na Seção 2.1, a **rede neural convolucional** (CNN) é um subconjunto de redes neurais artificiais, com a diferença que as entradas são vistas como imagens/matrizes e não como um vetor. Isso nos permite utilizar certas propriedades que antes não eram possíveis, como matrizes de convolução. A CNN auxilia na redução da quantidade de parâmetros necessários para o treinamento da rede e podemos visualizar a disposição dos seus neurônios em 3 dimensões (altura, largura e profundidade). A Figura 13 ilustra uma estrutura de uma CNN simples[10], onde l representa o número da camada. Note que de CONV para POOL há um salto de $l = 1$ para $l = 3$, devido à camada CONV e a de ativação que estão juntas na figura.

Na Figura 13 é observado 3 principais camadas da CNN: **Camadas de convolução** (*Convolutional Layer*); **Camadas de Pooling** (*Pooling Layer*); e as **Camadas de neurônios totalmente conectados** (*Fully-Connected Layer*). Cada camada tem uma função particular na rede. As **camadas de convolução** (CONV) possuem **filtros de aprendizagem**, também chamados de *kernels*. Sua função é aprender padrões únicos presentes na imagem, por exemplo, destacar apenas linhas verticais ou horizontais. Seus tamanhos são relativamente menores que a camada de entrada e o fluxo é guiado por matrizes de convolução. Uma convolução é uma operação de somatório de produtos entre

[9]Refere-se à coloração da imagem, por exemplo, 3 equivale a uma imagem no espectro RGB, e 1 a uma imagem com somente uma cor predominante.

[10]https://davidstutz.de/illustrating-convolutional-neural-networks-in-latex-with-tikz/

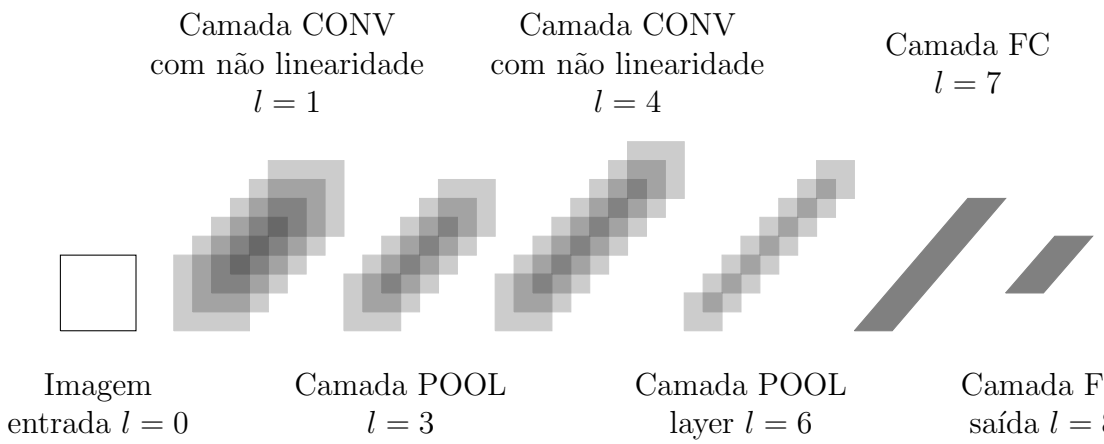


Figura 13: Rede neural convolucional.

as matrizes, ao longo da região coberta pelo filtro. Podemos visualizar alguns exemplos de filtros de tamanho 3×3 na Figura 14. Os filtros são tratados como parâmetros, ou

1	0	-1
1	0	-1
1	0	-1

1	1	1
0	0	0
-1	-1	-1

Figura 14: Filtros de tamanho 3×3 para detecção de linhas. À esquerda para detecção de linhas verticais e à direita linhas horizontais.

seja, podemos utilizá-los para treinar a rede, onde a representação é dada por w_i com i variando conforme o tamanho da imagem. O cálculo da multiplicação do filtro com a imagem é simples: analisamos cada *pixel* (*pixel* preto pontilhado na Figura 15) da imagem e selecionamos os seus valores vizinhos para compor uma matriz de mesmo tamanho do filtro (*pixels* vermelhos na Figura 15). Para cada conjunto de *pixels* selecionados na imagem, seus valores são multiplicados pelos correspondentes no filtro. Esses novos valores são somados e se tornam o valor final daquele *pixel* (*pixel* preto pontilhado) sendo posto na saída (*pixel* verde na Figura 15). Esta operação é feita em cada um dos canais de cor e se repete até computar todos os *pixels* para a camada de saída. Na realização das contas, ao encontrarmos valores maiores que 255, definimos o *pixel* de saída como 255 e caso o valor for menor que 0, definimos o *pixel* como 0. Ao utilizar um filtro nossa imagem é reduzida de tamanho, pois ignoramos as bordas da imagem. O processo descrito acima pode ser observado na Figura 15. Onde $*$ representa o operador de convolução.

Para convoluções em 3D é necessário repetirmos esse processo para cada filtro presente

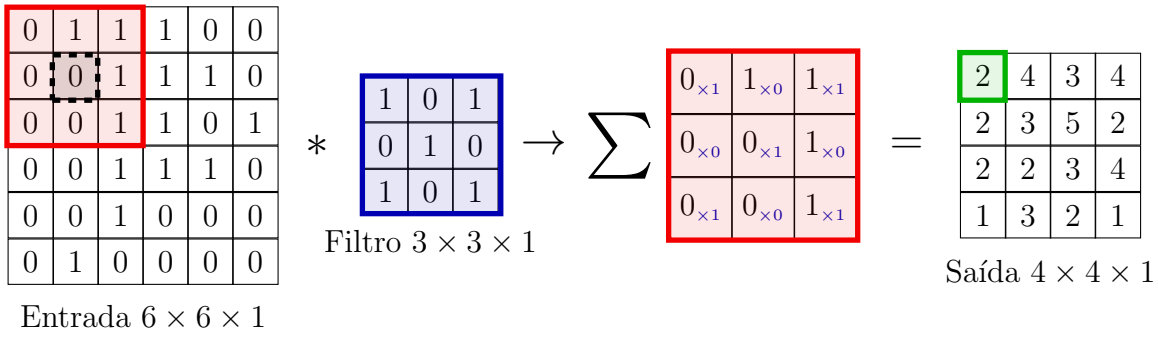


Figura 15: Aplicação de um filtro à uma imagem de entrada.

na camada. Temos a seguinte representação para a forma (*shape*) da matriz de saída

$$(n \times n \times n_c) * (f \times f \times n_c) = \left[(n - f + 1) \times (n - f + 1) \times n'_c\right],$$

onde n é o tamanho da imagem, f o tamanho do filtro, n_c o número de canais de cor e n'_c o número de filtros utilizados. Para contornarmos o problema da saída encolher e consequentemente jogar fora as informações presentes nas bordas, utilizamos da estratégia chamada *padding*. Essa estratégia normalmente consiste em adicionarmos zeros nas bordas da imagem (*zero-padding*), podendo ser um *same-padding*, como visualizado na Figura 16, que introduz zeros de tal forma a saída mantém o tamanho da entrada ou um *full-padding*, que introduz zeros de tal forma que ao aplicarmos o filtro, todos os *pixels* sejam visitados a mesma quantidade de vezes, aumentando a nossa saída. Mais técnicas podem ser encontradas em (DUMOULIN; VISIN, 2018). O *Padding* aumenta o tamanho da entrada, mas evita o desperdício de informações. Se adicionarmos uma borda de largura p, a largura em *pixels* da imagem passa a ser

$$n + 2p - f + 1 = n \implies p = \frac{f - 1}{2},$$

onde p é o tamanho do *padding* e f geralmente é ímpar.

0	0	0	0	0	0	0
0	1	4	3	4	1	0
0	1	2	4	3	3	0
0	1	2	3	4	1	0
0	1	3	3	1	1	0
0	3	3	1	1	0	0
0	0	0	0	0	0	0

Figura 16: *Padding* de tamanho $p = 1$. A região escura é a matriz original e está cercada de zeros devido à estratégia utilizada.

Além dos filtros e do *padding*, temos os *strides*. Eles definem o tamanho dos deslocamentos para aplicação do filtro, ou seja, a quantidade de posições que serão puladas de uma aplicação do filtro para outra. Por exemplo, na Figura 17, podemos observar em uma imagem $5 \times 5 \times 1$ a diferença entre os *strides* de tamanho 1 e 2. É possível observar que em

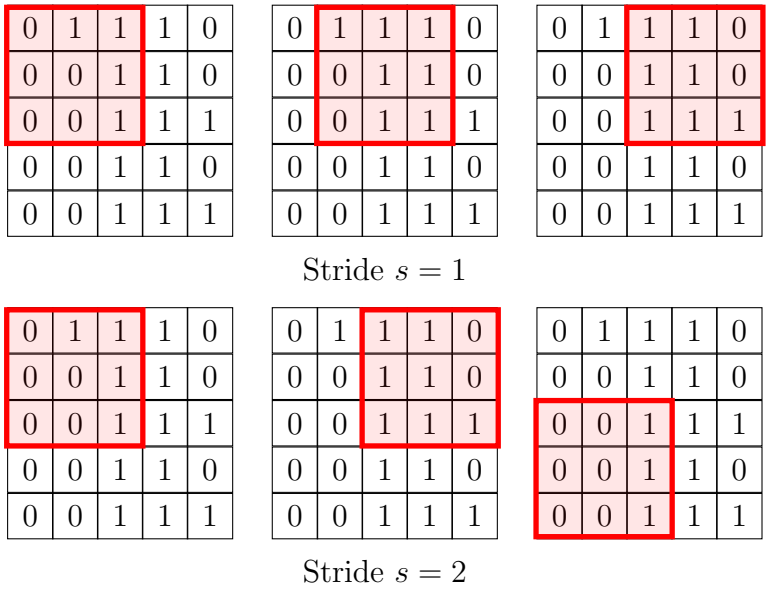


Figura 17: Diferentes tipos de *Strides*.

$s = 1$ deslocamos o bloco do filtro 3×3 apenas uma posição e para o $s = 2$ deslocamos o bloco do filtro 3×3 duas posições. A representação completa para o tamanho da imagem de saída com o *stride* e o *padding* é dado por:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor,$$

onde $\lfloor z \rfloor$ é o maior inteiro menor ou igual a z. Essas técnicas andam juntas na camada de convolução, sendo de extrema importância para o treinamento da rede.

A **camada de pooling** (POOL) também conhecida como subamostragem (*subsampling*) é geralmente adicionada logo após a camada de convolução e serve para reduzir progressivamente o tamanho espacial do modelo, reduzindo assim a quantidade de parâmetros na rede. Com essa redução, a camada de *pooling* entra para controlar também o *overfitting* dos dados. Existem duas principais categorias de *pooling*: *Max Pooling*, que seleciona apenas o maior número da unidade para a camada de saída; e *Average Pooling*, que seleciona a média dos números da unidade para a camada de saída. Geralmente são utilizados como filtros de tamanho 2×2 aplicados com um *stride* igual a 2. Isso faz com que nossa camada reduza para 1/4 do tamanho original, ou seja, dada uma camada de tamanho $28 \times 28 \times 6$, ao aplicarmos o *pooling* com $f = 2$ e $s = 2$, obtemos a saída de tamanho

$14 \times 14 \times 6$. Na Figura 18 podemos observar visualmente o processo descrito acima, onde cada região demarcada é uma passada do filtro de *pooling*. Na região vermelha, o valor máximo $\max\{1, 3, 2, 9\} = 9$ é colocado na saída. A ideia é análoga para os demais valores.

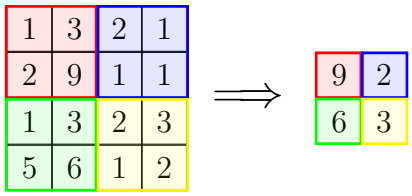


Figura 18: *Max Pooling* em uma matriz 4×4.

Por fim, temos a **camada de neurônios totalmente conectados** (FC). A estrutura desta camada é a usada na seção 2.1. Sua entrada é a saída da última camada de *pooling*, achatando a matriz e transformando em vetor para realizar o processo de propagação, passando pelos neurônios até a saída, nos retornando a resposta esperada. Aqui, normalmente, por serem muito utilizadas para reconhecimento de diversas categorias, nossas ativações nas camadas intermediarias são $ReLU$ e na saída, *softmax* (utilizada para redes neurais de múltiplas classificações).

Diferentemente das redes observadas no Capítulo 2, as camadas da rede convolucional compartilham recursos entre si, isso significa que todos os neurônios de uma determinada camada detectam o mesmo recurso em locais diferentes na imagem de entrada, isto é, um detector de bordas verticais é útil em uma parte da imagem, provavelmente será útil em outras partes da imagem. Esse compartilhamento de recursos reduz o número de parâmetros presentes na rede, ou seja, dado uma camada com 16 filtros de tamanho $5 \times 5 \times 3$, podemos definir o número de parâmetros pela seguinte equação

$$(f^{[l]} \times f^{[l]} \times n_c^{[l-1]} + 1) \times n_c^{[l]},$$

onde f é o tamanho do filtro, $n_c^{[l-1]}$ é o número de canais na camada anterior, $n_c^{[l]}$ é o número de canais na camada atual e 1 é o viés. Dessa forma temos um total de $(5 * 5 * 3 + 1) * 16 = 1216$ parâmetros na camada convolucional. Já em uma camada de neurônios totalmente conectados, com 784 neurônios na entrada e 15 na camada oculta, temos um total de $784 * 15 + 15 = 11775$ parâmetros. Em outras palavras, a camada de neurônios totalmente conectada teria 9 vezes mais parâmetros que a camada convolucional.

4.3 Algumas aplicações

Nesta seção são discutidas algumas aplicações com redes neurais convolucionais utilizando a plataforma *TensorFlow*. Apresentaremos dois conjuntos de dados disponíveis no TFDS e mostraremos como treinar a rede para o reconhecimento desses tipos de dados. Por fim, compararemos os modelos apresentados no Capítulo 3, visando encontrar o método com melhor desempenho para determinado conjunto de dados.

4.3.1 EuroSAT

- **Descrição do problema:** O conjunto de dados EuroSAT/RGB é baseado em imagens de livre acesso capturadas pelo satélite *Sentinel-2*, fornecidas no programa Copernicus de observação da Terra com foco na União Europeia. Pode ser aplicado em detecção de desmatamento, mudanças no território, entre outros.

- **Características da base de dados:** As imagens de tamanho $64 \times 64 \times 3$ contêm apenas as bandas de frequência ópticas RGB, codificadas como imagem no formato `.JPEG`. São constituídas em 10 classes com 2.000 a 3.000 imagens em cada, totalizando 27.000 amostras de treinamento marcadas e georreferenciadas em seus rótulos. Uma amostra pode ser observada na Figura 19. Há outro *dataset* contendo imagens cobrindo 13 bandas espectrais, chamado EuroSAT/all.

Figura 19: Exemplos de imagens presentes no *dataset* EuroSAT.

- **URL da base de dados:** https://github.com/phelber/eurosat.
- **Base de dados no *TensorFlow*:** https://www.tensorflow.org/datasets/catalog/eurosat.

Dado as características da EuroSAT (HELBER et al., 2018), realizamos alguns testes sobre os dados. O código está presente no Apêndice B. Primeiramente capturamos os dados da EuroSAT através do TFDS, onde separamos os dados na seguinte proporção: 80% treinamento, 10% testes e 10% validação. Em seguida normalizamos os dados para valores entre [0,1] e separamos em mini-lotes.

Criamos a estrutura da rede neural convolucional, onde montamos uma rede com 2 camadas de convolução e *pooling*, uma de neurônios totalmente conectados e a saída contendo 10 neurônios com função de ativação *softmax*. Cada camada de convolução e de *pooling* são construídas com *same-padding*[11]. Para criação do modelo foi utilizada uma API Funcional[12] para manipular o modelo com mais flexibilidade. A rede pode ser vista da seguinte maneira:

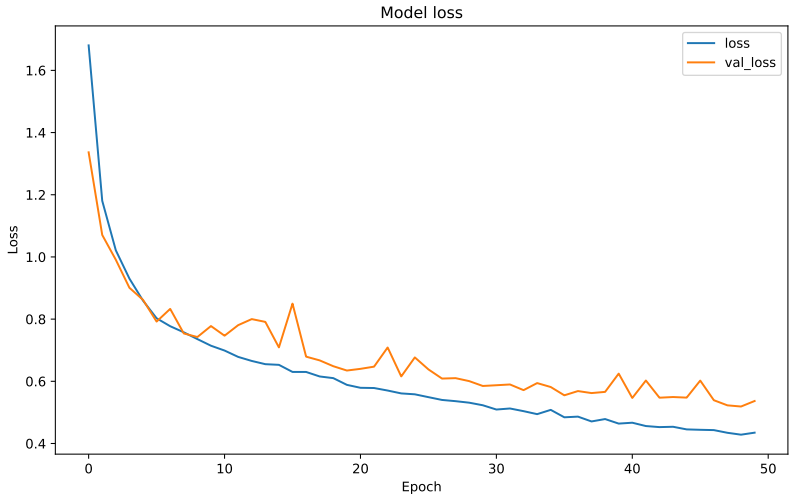
$$\text{INPUT} \to \text{CONV2D} \to \text{RELU} \to \text{MAXPOOL} \to \text{CONV2D} \to \text{RELU} \to \text{MAXPOOL} \to \text{FLATTEN} \to \text{DENSE} \to \text{OUTPUT}.$$

Com o modelo definido, compilamos a rede criada, utilizando o método de otimização `Adam`[13] visto no Capítulo 3 e a função de perda *Cross Entropy* utilizada para multi-classificação:

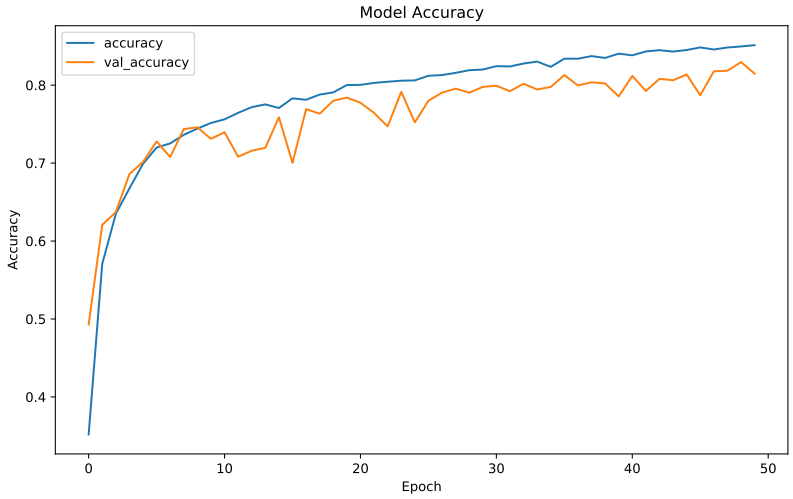
$$C(w,b) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log a(x^{(i)}; w, b).$$

Após a criação da rede, implementamos o treinamento. Definimos o número de épocas em 50, onde o número de iterações por época é dado $\lceil (0.8 \cdot m)/B \rceil$, sendo m a quantidade de dados presentes no conjunto de treinamento e B o tamanho do mini-lote, no nosso caso, 64. Com o modelo treinado e validado, verificamos a acurácia/acerto porcentual nos dados de teste e plotamos os gráficos referentes a função de perda e acurácia encontrando algo semelhante ao presente na Figura 20, pois a geração de pesos é randômica e o resultado pode variar um pouco de execução para execução.

[11] mantém o tamanho original da matriz de entrada na matriz de saída
[12] https://www.tensorflow.org/guide/keras/functional
[13] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam



(a) Função perda, EuroSAT.



(b) Acurácia, EuroSAT.

Figura 20: (a) comparação entre a função de perda nos dados de treinamento e de validação; (b) comparação entre a acurácia do modelo nos conjuntos de treinamento e validação, ambos do modelo com 50 épocas.

Pela Figura 20(a), é possível observar que o modelo obteve melhora na redução da função de perda até ao final das 50 épocas, onde *"loss"* representa a função de perda nos dados de treinamento e *"val_loss"* a perda no conjunto de validação. Note também que elas andam próximas até a parada. Na Figura 20(b) a acurácia do modelo é crescente, chegando em $\approx 90\%$ para modelo de treinamento (denotada de *"accuracy"* no gráfico) e $\approx 80\%$ no conjunto de validação (denotada de *"val_accuracy"* no gráfico).

No Apêndice B vimos como implementar uma rede do zero até seu treinamento. Os códigos são uma abordagem simples para o conjunto de dados da EuroSAT. A partir deles, foram realizados testes com os métodos apresentados no Capítulo 3 de modo a verificar qual tem o melhor desempenho no conjunto. Foi utilizada uma taxa de aprendizagem inicial pré-definidas pelos modelos presentes no TF, com exceção do método SGD com Momento e ADAGrad. As taxas padrão utilizadas são, para o método SGD, $\eta = 0,01$, e para RMSProp e Adam iguais a $\eta = 0,001$. Para o SGD com Momento utilizamos $\eta = 0,001$, e para o ADAGrad, $\eta = 0,01$. Os testes foram realizados através da plataforma *Google Colaboratory*, conhecida como "Google Colab". Foram utilizados *notebooks* Jupyter, executados na nuvem com GPU gratuitas (Nvidia K80s, T4s, P4s ou P100s). Infelizmente, não há como escolher qual tipo de GPU o Colab utiliza, portanto executamos os códigos na GPU disponível no momento.

Na Figura 21 podemos visualizar graficamente a diminuição da função de perda do modelo em cada estágio do processo de treinamento da rede. O eixo vertical está em escala logarítmica para evidenciar erros próximos de zero. Os métodos foram executados com 100 épocas e mini-lotes de tamanho 64. Ao analisarmos o gráfico, com exceção do ADAGrad, os métodos obtiveram uma diminuição considerável da função de perda. Observa-se que o ADAGrad foi o método que obteve o pior desempenho, com erro consideravelmente alto ao longo das épocas; ele conseguiu reduzir de forma contínua o erro percentual até a época 60, porém bastante demorada para a quantidade de iterações propostas e se estagnou a partir da época 80. Ao contrário, Adam e RMSProp obtiveram considerável diminuição do erro. Evidentemente, caso executássemos os métodos por mais épocas, essa diminuição tenderia a continuar.

Pela Tabela 3 podemos verificar a porcentagem de acertos entre os dados de treinamento e validação/teste. Todos os métodos não passaram de 7% de disparidade entre os conjuntos. Note que apesar da lenta convergência do método ADAGrad, ele obteve uma diferença entre treinamento e teste relativamente pequena.

% de acertos	**SGD**	**SGD com Momento**	**ADAGrad**	**RMSProp**	**Adam**
Treinamento	85,01	85,74	80,42	90,61	90,94
Validação	81,85	81,30	78,19	84,78	84,44
Teste	81,18	81,96	78,37	83,96	83,92

Tabela 3: Percentual de acertos nos conjuntos de treinamento, teste e validação da EuroSAT.

Podemos concluir dos testes realizados no conjunto de dados EuroSAT que, em relação aos acertos no conjunto de treinamento, o Adam obteve o melhor resultado com 90,94%,

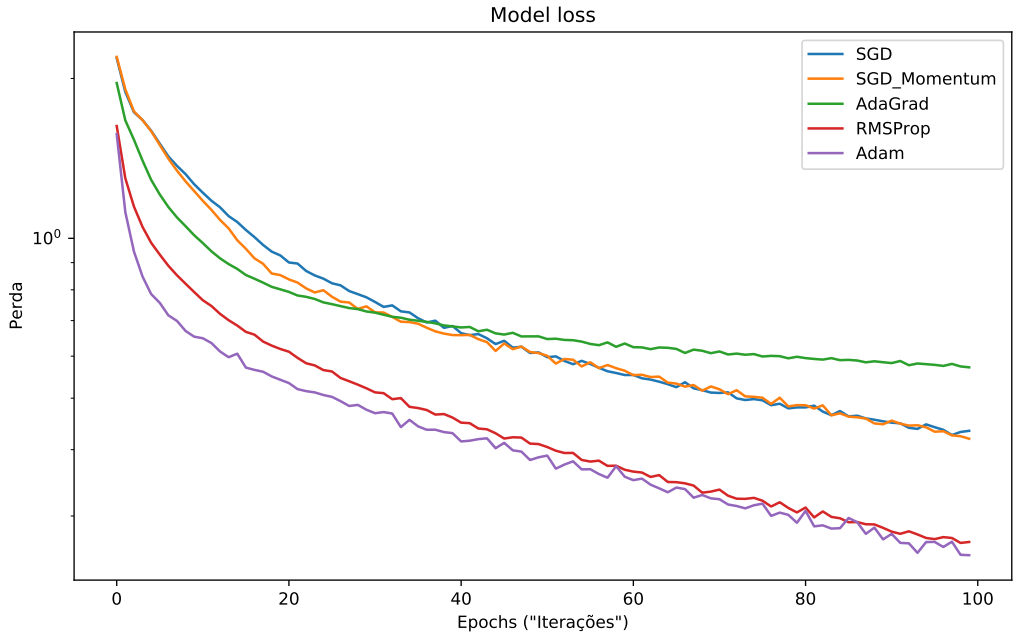


Figura 21: Gráfico comparativo entre os métodos com 100 *epochs* no conjunto EuroSAT.

seguido do RMSProp com 90,61%. Porém, no conjunto de testes e validação, o último obteve uma leve vantagem sobre o Adam. A acurácia sobre os dados de validação pode ser visualizada na Figura 22. O ADAGrad não teve um bom desempenho. Um possível motivo é que como cada termo adicionado à soma acumulada é positivo, ela continua aumentando durante a fase de treinamento, fazendo com que a taxa de aprendizagem diminua muito rapidamente, ocorrendo a estagnação do método.

4.3.2 Beans

- **Descrição do problema:** O conjunto de dados Beans consiste em imagens de folhas de feijões fotografadas no campo usando câmeras de celular, com intuito de montar uma rede para detectar folhas saudáveis ou com doenças. Os dados foram anotados por especialistas do *National Crops Resources Research Institute* (NaCRRI) em Uganda e coletados pelo laboratório de pesquisa Makerere AI.

- **Características da base de dados:** As imagens contêm bandas de frequência ópticas RGB, codificadas como imagem `.JPEG` de tamanho $500 \times 500 \times 3$. Possuem 3 classes, sendo 2 de doenças e uma saudável. As doenças descritas incluem mancha angular da folha (*angular leaf spot*) e ferrugem do feijão (*bean rust*). Ao todo, o conjunto possui 1.296 imagens sendo 428 de classe saudável, 432 da doença mancha

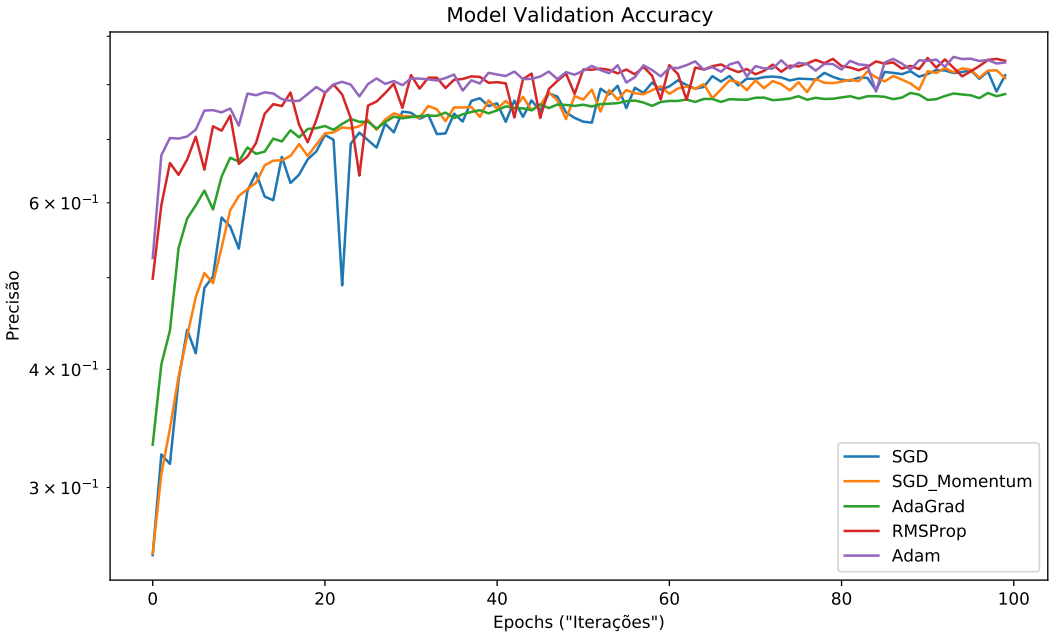


Figura 22: Gráfico comparativo sobre a precisão entre os métodos com 100 *epochs* no conjunto de validação da EuroSAT.

angular da folha e 436 da doença ferrugem do feijão. Uma pequena amostra de seu conjunto pode ser visualizada na Figura 23.

- **URL da base de dados:** https://github.com/AI-Lab-Makerere/ibean.
- **Base de dados no *TensorFlow*:** https://www.tensorflow.org/datasets/catalog/beans.

Dado as características da Beans (LAB, 2020), efetuamos alguns testes sobre os dados. A construção passo a passo do código é semelhante à observada na seção anterior, com algumas variações. Definimos uma função de pré processamento e repetimos o processo de embaralhamento e separação dos dados visto anteriormente, aqui o tamanho de mini-lote utilizado é igual a 64. Montamos nossa rede neural convolucional com 3 camadas de convolução e *pooling*, uma de neurônios totalmente conectados e a saída contendo 3 neurônios com a função de ativação *softmax*. A rede pode ser vista da seguinte maneira:

$$\text{INPUT} \to \text{CONV2D} \to \text{RELU} \to \text{MAXPOOL} \to \text{CONV2D} \to \text{RELU} \to \text{MAXPOOL} \to$$
$$\text{CONV2D} \to \text{RELU} \to \text{MAXPOOL} \to \text{FLATTEN} \to \text{DENSE} \to \text{OUTPUT}.$$

Repare que a rede criada possui mais camadas de convolução e *pooling* que a da Seção 4.3.1. Como o tamanho da entrada é bastante elevado, mesmo com a redução de

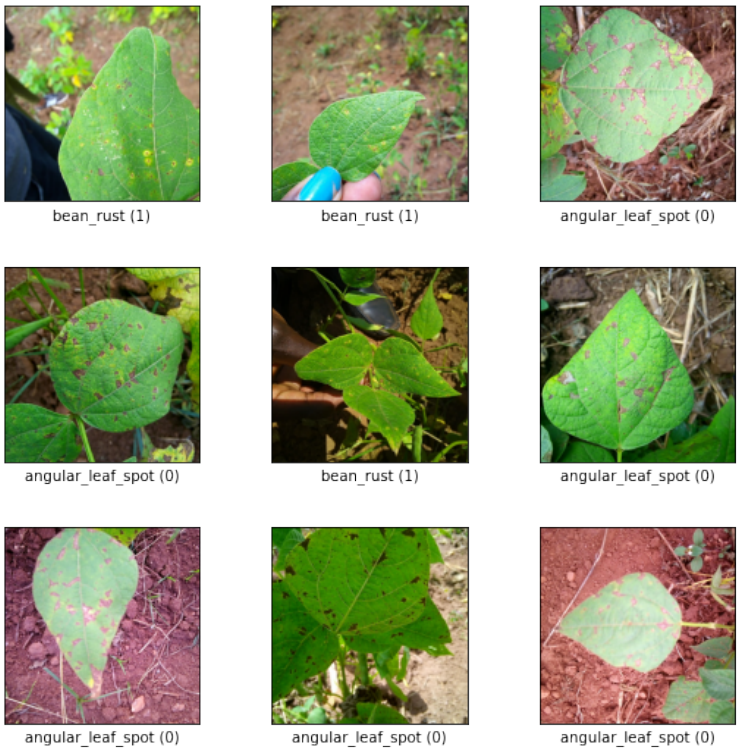


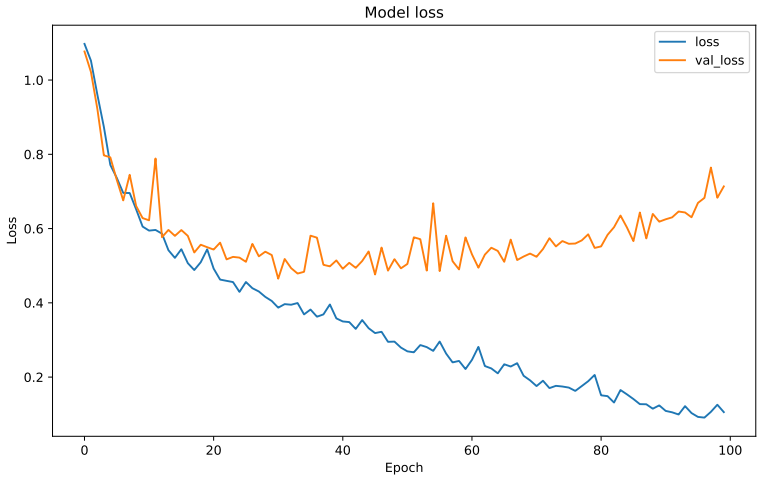
Figura 23: Exemplos de imagens presentes no *dataset* Beans.

$500 \times 500 \times 3$ para $250 \times 250 \times 3$, precisamos de mais camadas até convertermos para uma de neurônios totalmente conectados, aqui reduzimos do tamanho $250 \times 250 \times 3$ para até $4 \times 4 \times 64$ na nossa última camada de *Max Pooling*, chegando na camada *Flatten*[14] com $4 * 4 * 64 = 1024$ neurônios. Por fim, após o modelo ter sido montado, compilamos e treinamos com 100 épocas e plotamos os gráficos como na Seção 4.3.1.

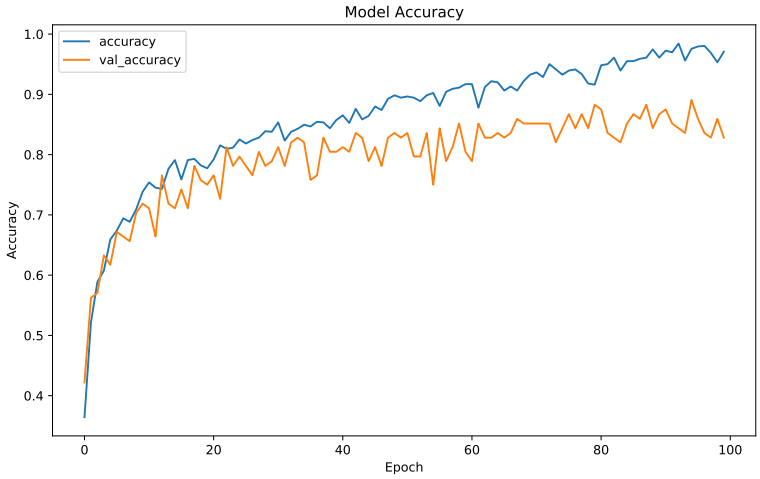
Podemos observar na Figura 24 um comportamento diferente do usual, a função de perda aplicada nos dados de validação obteve uma queda, mas logo depois, a partir da época 20, começou a subir voltando próximo ao patamar inicial. O fenômeno que está ocorrendo é o *overfitting*, onde a rede está se ajustando perfeitamente aos dados de treinamento e ao testar com dados nunca vistos, não consegue realizar o acerto.

Para contornarmos esse problema existem algumas técnicas úteis. A primeira é realizar uma parada prematura no treinamento da rede. Definimos uma porcentagem da função de perda da validação ("*val_loss*") esperada e ao ser atingida o problema é encerrado. No nosso caso, definindo 50%, a rede teria sido executada por ≈ 20 épocas, obtendo um resultado insatisfatório, como podemos observar pelo na Figura 24(b). A segunda técnica, como constatado na Seção 3.3, é aumentar a quantidade de dados somente no conjunto de treinamento, mantendo os de validação e teste padrões. Para aumentar os

[14]Camada que achata nossa matriz para um vetor



(a) Função perda do modelo e da validação da Beans.



(b) Acurácia do modelo e da validação da Beans.

Figura 24: Gráficos do modelo treinado com 100 *epochs*.

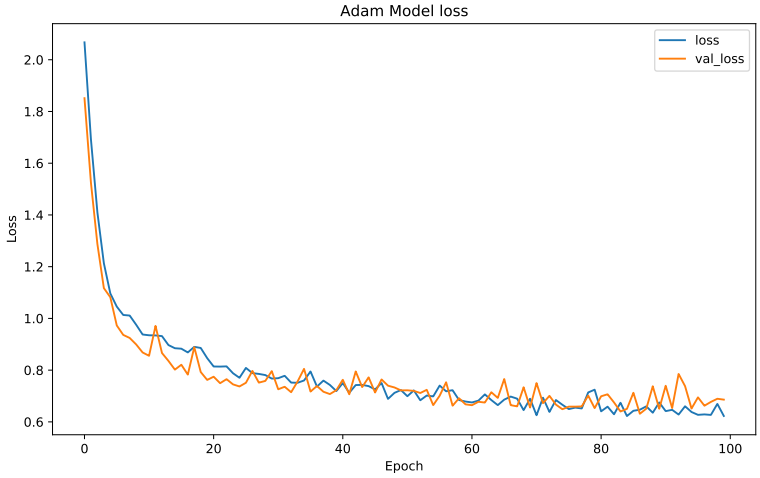
dados, podem ser feitas rotações, mudanças no contraste, no brilho, dentre outras. Essa técnica é útil quando não conseguimos obter mais dados semelhantes aos de treinamento. A terceira técnica é a regularização ℓ_1, onde adicionamos uma pequena penalização $\lambda||z||_1$ à função de custo $C(w,b)$ do modelo. Por exemplo, no caso do erro quadrático médio teríamos a função de custo:

$$C(w,b) = \frac{1}{2m}\sum_{i=0}^{m} ||a(x^{(i)}; w, b) - y^{(i)}||^2 + \lambda||(w,b)||_1.$$

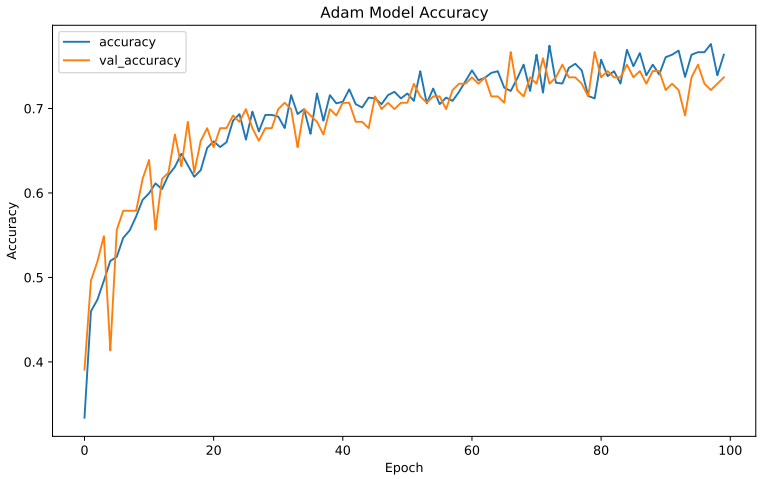
Na regularização ℓ_1 os valores dos pesos tendem a ser zero e pesos menores reduzem os impactos dos neurônios intermediários, diminuindo a complexidade do modelo e evitando o *overfitting*. Existem diversas outras técnicas para evitar o *overfitting*, como *Dropout* e

regularização ℓ_2, entretanto não serão tratadas neste trabalho.

No nosso modelo, utilizamos as seguintes ideias: aumento na quantidade de dados e a regularização ℓ_1. Realizamos uma rotação na imagem em todos os lados e adicionamos uma mudança em seu contraste aumentando a quantidade de dados. Todas as técnicas para o aumento de dado são efetuadas de forma randômica. Executamos algumas funções importantes para normalizar os dados de treino, como dividir em mini-lotes e embaralhamento dos dados. Efetuado as alterações necessárias no conjunto de dados, aplicamos a regularização ℓ_1 na camada densa no nosso modelo. Por fim, após o modelo ter sido montado com as novas implementações, compilamos, treinamos com 100 épocas e por fim, plotamos os gráficos. Nota-se na Figura 25 que o modelo apresenta uma taxa menor no



(a) Função perda do modelo e da validação.



(b) Acurácia do modelo e da validação.

Figura 25: Gráficos do modelo treinado em 100 *epochs* com regularização ℓ_1 e aumento de dados.

conjunto de treinamento em relação à observada na Figura 24, porém a `val_loss` manteve o decréscimo junto à `loss`. Evitando assim o *overfitting* dos dados, possibilitando um treinamento mais longo e preciso.

A partir das técnicas mostradas acima para redução do *overfitting*, realizamos testes com os métodos apresentados no Capítulo 3 de modo a verificar qual tem o melhor desempenho no conjunto. Utilizamos as taxas de aprendizagem iniciais pré-definidas para os métodos presentes no *TensorFlow*, com exceção do método ADAGrad. As taxas padrão são, para o método SGD e SGD com Momento iguais a $\eta = 0,01$, e ADAGrad, RMSProp e Adam iguais a $\eta = 0,001$. Empregamos para o ADAGrad uma taxa diferente do padrão, $\eta = 0,01$.

Na Figura 26 podemos visualizar graficamente a diminuição da função de perda do modelo em cada estágio do processo de treinamento da rede. O eixo vertical está em escala logarítmica para evidenciar erros próximos de zero. Os métodos foram executados com 100 épocas. Ao analisarmos o gráfico, todos os métodos obtiveram uma diminuição considerável da função de perda. Note que o SGD foi o método obteve o pior desempenho, com erro consideravelmente alto ao longo das *epochs*. Ao contrário, Adam obteve um percentual considerável de melhora na diminuição do erro, chegando ao final com uma grande diferença entre os demais métodos. Sua função de perda ficou em 0,3426 enquanto o método mais próximo, RMSprop alcançou 0,7137.

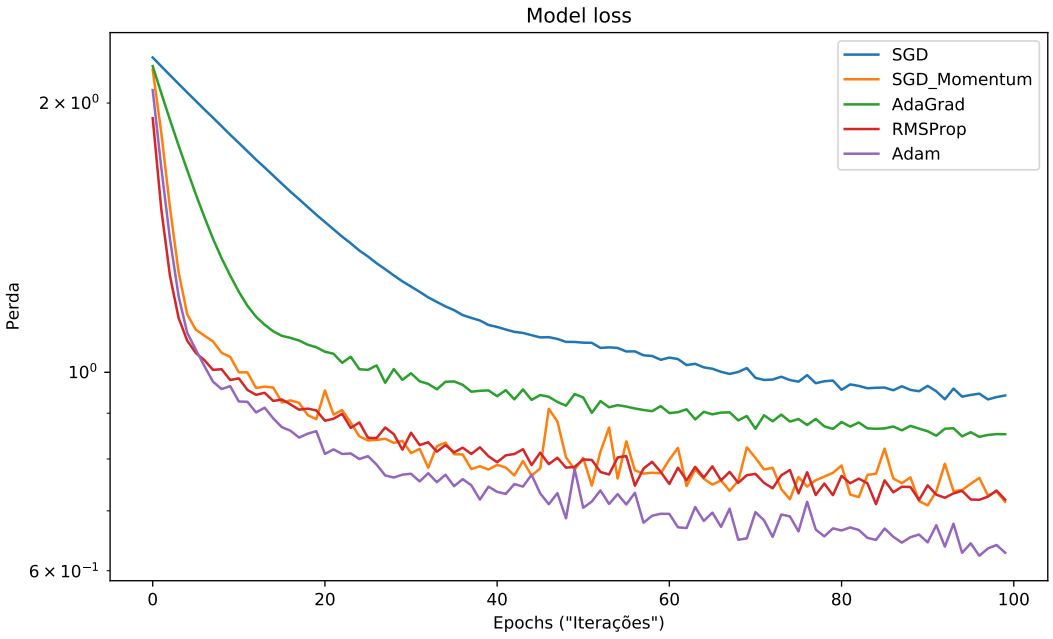


Figura 26: Gráfico comparativo entre os métodos com 100 *epochs* no conjunto Beans.

Pela Tabela 4 podemos verificar a porcentagem de acertos nos dados de treinamento, validação e teste. Observe que ao utilizarmos as técnicas para evitar o *overfitting*, a disparidade das porcentagens de acertos ficou bastante pequena. Note que o Adam obteve a maior taxa de acertos no conjunto de validação, mas ao utilizarmos o conjunto de teste, o SGD com Momento alcançou a maior porcentagem de acertos. Adam se equiparou a RMSProp.

% de acertos	SGD	SGD com Momento	ADAGrad	RMSProp	Adam
Treinamento	56,15	72,85	60,94	73,34	75,49
Validação	58,65	76,69	60,15	71,43	77,44
Teste	61,71	75,00	66,40	74,21	74,21

Tabela 4: Tabela referente a porcentagem de acertos dos métodos nos conjuntos de treinamento, teste e validação da Beans.

Concluímos com os testes realizados que, em relação aos acertos no conjunto de treinamento, o Adam obteve o melhor resultado com 75,49% e logo em seguida o RMSProp com 73,34%. No conjunto de validação, o Adam obteve o melhor resultado com 77,44% e logo em seguida o SGD com momento com 76,69%. A acurácia sobre os dados de validação pode ser visualizada na Figura 27. O SGD chegou a apenas 56,15% no conjunto de treinamento, uma porcentagem relativamente baixa.

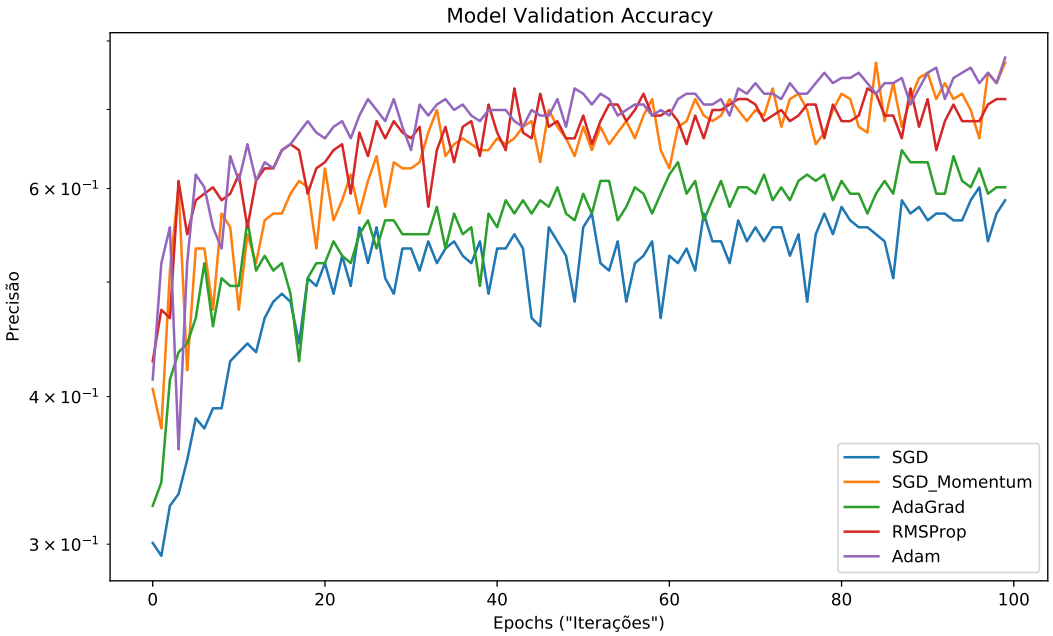


Figura 27: Gráfico comparativo sobre a precisão entre os métodos com 100 *epochs* no conjunto de validação da Beans.

O conjunto de dados Beans é interessante pois, por conter um número razoavelmente pequeno de dados (apenas 1.296 imagens), é possível observar o *overfitting* na prática, levando ao estudo de técnicas para contornar o problema. Mesmo assim, os resultados obtidos aqui provavelmente não são os melhores possíveis, visto que há muitas outras técnicas que podem ser aplicadas para melhorar ainda mais o treinamento.

Conclusões

Neste trabalho realizamos um estudo introdutório das redes neurais, apresentando redes neurais artificiais e convolucionais, além de tratar de modelos e técnicas de otimização para o aprendizado de máquina supervisionado com foco em classificação de imagens. Apresentamos o método do gradiente estocástico e a sua prova de convergência. Para tal, introduzimos conceitos fundamentais de probabilidade. Abrangemos variações do método do gradiente estocástico, como gradiente estocástico com momento, ADAGrad, RMSProp e Adam.

Comentamos sobre o problema de reconhecimento de caracteres numéricos com o conjunto MNIST e como tratar o *underfitting* e *overfitting*. Fornecemos códigos para um início do estudo sobre o tema utilizando a plataforma *TensorFlow*, de forma que o leitor possa implementar sua primeira rede neural.

Realizamos testes com os métodos discutidos para descobrirmos qual obtinha o melhor desempenho em determinada rede e conjunto de dados. Nos testes com redes neurais totalmente conectadas, onde utilizamos o conjunto de dados MNIST, o algoritmo que obteve o melhor desempenho foi o Adam, tendo $\approx 98{,}1\%$ de acerto nos dados de teste. Em redes neurais convolucionais, onde utilizamos os conjuntos de dados EuroSAT e Beans, o método que se saiu melhor foi o RMSProp para a EuroSAT e SGD com Momento para a Beans, tendo respectivamente 83,96% e 75,00% de acerto nos dados de teste.

Por fim, concluímos que dedicar um tempo aos estudos sobre os diferentes métodos de otimização visando obter melhores resultados para o conjunto de dados proposto é de grande valia, já que cada método pode obter um desempenho diferente conforme o tamanho e a categoria de rede utilizada, da taxa de aprendizagem em cada método e do tamanho do mini-lote.

Referências Bibliográficas

ABADI, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>.

ALARIE, B.; NIBLETT, A.; YOON, A. H. How artificial intelligence will affect the practice of law. *University of Toronto Law Journal*, University of Toronto Press, v. 68, n. supplement 1, p. 106–124, 2018.

BAXT, W. G. Application of artificial neural networks to clinical medicine. *The lancet*, Elsevier, v. 346, n. 8983, p. 1135–1138, 1995.

BOTTOU, L.; CURTIS, F. E.; NOCEDAL, J. Optimization methods for large-scale machine learning. *Siam Review*, SIAM, v. 60, n. 2, p. 223–311, 2018.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, v. 12, n. 7, 2011.

DUMOULIN, V.; VISIN, F. *A guide to convolution arithmetic for deep learning*. 2018. ArXiv:1603.07285.

HEBB, D. O. *The organisation of behaviour: a neuropsychological theory*. [S.l.]: Science Editions New York, 1949.

HELBER, P. et al. Introducing eurosat: A novel dataset and deep learning benchmark for land use and land cover classification. In: *IGARSS 2018 - 2018 IEEE International Geoscience and Remote Sensing Symposium*. [S.l.: s.n.], 2018. p. 204–207.

JAMES, B. R. *Probabilidade: Um curso intermediário*. Rio de Janeiro: IMPA, 2004.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. [S.l.: s.n.], 2015.

LAB, M. A. Bean disease dataset. January 2020. Disponível em: <https://github.com/AI-Lab-Makerere/ibean/>.

LECUN, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, Ieee, v. 86, n. 11, p. 2278–2324, 1998.

LECUN, Y.; CORTES, C. MNIST handwritten digit database. 2010. Disponível em: <http://yann.lecun.com/exdb/mnist/>.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

NIELSEN, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015. Disponível em: <http://neuralnetworksanddeeplearning.com/>.

RIBEIRO, A.; KARAS, E. *Otimização Contínua: Aspectos Teóricos e computacionais*. [S.l.]: CENGAGE DO BRASIL, 2013. ISBN 9788522115013.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, American Psychological Association, v. 65, n. 6, p. 386, 1958.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. *nature*, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986.

RUSSELL, S.; NORVIG, P. Artificial intelligence: a modern approach. 2002.

SUN, S. et al. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, IEEE, v. 50, n. 8, p. 3668–3681, 2019.

SUTSKEVER, I. et al. On the importance of initialization and momentum in deep learning. In: PMLR. *International conference on machine learning*. [S.l.], 2013. p. 1139–1147.

TIELEMAN, T.; HINTON, G. et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, v. 4, n. 2, p. 26–31, 2012.

ZHAO, W.; DU, S.; EMERY, W. J. Object-based convolutional neural network for high-resolution imagery classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, v. 10, n. 7, p. 3386–3396, 2017.

APÊNDICE A – Código exemplo

```
import numpy as np

#Define Funcao Sigmoid
def sigmoid(x):
    return 1 / (1+np.exp(-x))

#Define Derivada da Funcao Sigmoid
def sigmoid_prime(x):
    return sigmoid(x) * (1 - sigmoid(x))

#Cria os pesos (w, b) e entradas (x, y)
parametros = { "w1": np.array([[0.1,0.4],[-0.5, 0.1],[0.9, 0.5]]),
               "w2": np.array([[0.8, -0.12,0.3]]),
               "b1": np.array([[-0.3],[0.82],[0.03]]),
               "b2": np.array([[-0.8]]) }

x = np.array([[0.5],[0.2]])
y = np.array([[0.9]])

#Funcao feedforward
def feedforward(a0, parametros):
    x = np.dot(parametros["w1"], a0) + parametros["b1"]
    a1 = sigmoid(x)
    x = np.dot(parametros["w2"], a1) + parametros["b2"]
    a2  = sigmoid(x)
    return a2

#Calculo do Backpropagation
def backprop(x, y, parametros):
    biases = (parametros["b1"], parametros["b2"])
    weights = (parametros["w1"], parametros["w2"])
    nabla_b = list(np.zeros(b.shape) for b in biases)
    nabla_w = list(np.zeros(w.shape) for w in weights)
```

```
    # feedforward
    activation = x
    activations = [x]
    z_error = []

    for b, w in zip(biases, weights):
        z = np.dot(w, activation) + b
        z_error.append(z)
        activation = sigmoid(z)
        activations.append(activation)

    # backpropagation
    o_error = (activations[-1] - y)  # erro na output
    o_delta = o_error * sigmoid_prime(z_error[-1])  # aplicando a
                                        derivada do sigmoid no erro
    print(f"Erro da Output: {o_delta}")
    nabla_b[-1] = o_delta
    nabla_w[-1] = np.dot(o_delta, activations[-2].T)  # (output -->
                                        hidden)
    z2_delta = np.dot(weights[-1].T, o_delta)*sigmoid_prime(z_error[-2])
    nabla_w[-2] = np.dot(z2_delta, activations[-3].T)
    nabla_b[-2] = z2_delta

    grad = {'W': nabla_w,
            'b': nabla_b}

    return grad

def treinamento(x,y,parametros):
    for i in range(0, 100):
        A2 = feedforward(x, parametros)
        grad = backprop(x, y, parametros)

        for l in range(0,2):
            parametros["w" + str(l+1)] = parametros["w" + str(l+1)] - 0.
                                                5 * grad['W'][l]
            parametros["b" + str(l+1)] = parametros["b" + str(l+1)] - 0.
                                                5 * grad['b'][l]


    return parametros
```

APÊNDICE B – Códigos das implementações em *Tensorflow*

```
# Codigo exemplo tensorflow
# importando as bibliotecas
import tensorflow as tf
from tensorflow import keras
import numpy as np

def house_model(y_new):
    xs = np.array([1, 4, 0, 3, 5, 8, 9, 2, 4], dtype=float)
    ys = np.array([1, 2.5, .5, 2, 3, 4.5, 5, 1.5, 2.5], dtype=float)
    model = tf.keras.Sequential([
            keras.layers.Dense(1, input_shape=[1])])
    model.compile(optimizer="sgd", loss="mean_squared_error")
    model.fit(xs, ys, epochs = 100)
    return model.predict(y_new)[0]

# Classificar o valor de uma casa com 7 quartos
classification = house_model([7.0])
print(classification)
```

O código foi executado no ambiente Colab, recomendado que a cada função após cada comentário com ## seja uma célula no Colab, buscando a utilização esperado do código.

Código referente a testes com o dataset EuroSAT:

```
## Importamos as bibliotecas necessarias
import tensorflow as tf
import tensorflow.keras.layers as tfl
from tensorflow import keras
import tensorflow_datasets as tfds
import pandas as pd
```

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

## Separacao dos dados
DATA_DIR = "/eurosat/input/"
(ds_train, ds_valid, ds_test), metadata = tfds.load("eurosat",
    split=["train[:80%]", "train[80%:90%]", "train[90%:]"],
    data_dir = DATA_DIR,
    with_info=True,
    as_supervised=True)

## Funcao de pre-processamento
def preprocess(image, label):
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)
    return image, label

## Tamanho do lote
BATCH_SIZE = 64
## Ajusta o valor dinamicamente no tempo
AUTO = tf.data.experimental.AUTOTUNE
## Tamanho do conjunto de treino para realizar o embaralhamento
SHUFFLE_BUFFER = int(metadata.splits['train'].num_examples * 0.8)

## Aplicando as normalizacoes nos conjuntos
ds_train = (ds_train.map(preprocess, AUTO)
                    .cache()
                    .shuffle(SHUFFLE_BUFFER)
                    .repeat()
                    .batch(BATCH_SIZE)
                    .prefetch(AUTO))
ds_valid = (ds_valid.map(preprocess, AUTO)
                    .cache()
                    .batch(BATCH_SIZE)
                    .prefetch(AUTO))
ds_test = (ds_test.map(preprocess, AUTO)
                    .cache()
                    .batch(BATCH_SIZE)
                    .prefetch(AUTO))

## Criacao do modelo convolucional EuroSAT
def convolutional_model(input_shape):
    input_img = keras.Input(shape=input_shape)
    #Primeira camada CONV e POOL
```

```
        Z1 = tfl.Conv2D(16, 3, strides = 2, padding = 'same')(input_img)
        A1 = tfl.ReLU()(Z1)
        P1 = tfl.MaxPool2D(pool_size = (2, 2), padding = 'same')(A1)
        #Segunda camada CONV e POOL
        Z2 = tfl.Conv2D(32, 3, strides = 2, padding = 'same')(P1)
        A2 = tfl.ReLU()(Z2)
        P2 = tfl.MaxPool2D(pool_size = (2, 2), padding = 'same')(A2)
        #Camada FC
        F = tfl.Flatten()(P2)
        outputs = tfl.Dense(10, activation="softmax")(F)
        model = tf.keras.Model(inputs=input_img, outputs = outputs)
        return model

    ## Copilando o modelo
    eurosat_model = convolutional_model((64, 64, 3))
    eurosat_model.compile(optimizer='adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])
    eurosat_model.summary()

    ## Treinamento do modelo
    STEPS_PER_EPOCH = int(metadata.splits['train'].num_examples*0.8)//64
    history = eurosat_model.fit(ds_train,
                            epochs=50,
                            validation_data=ds_valid,
                            steps_per_epoch=STEPS_PER_EPOCH)

    ## Acuracia do modelo
    loss, acc = eurosat_model.evaluate(ds_test)
    print("Accuracy", acc)

    ## Plotagem dos graficos
    df_loss_acc = pd.DataFrame(history.history)
    df_loss = df_loss_acc[['loss','val_loss']]
    df_acc = df_loss_acc[['accuracy','val_accuracy']]
    df_loss.plot(title='Model loss').set(xlabel='Epoch',ylabel='Loss')
    df_acc.plot(title='Model Accuracy').set(xlabel='Epoch',ylabel='
                                        Accuracy')
```

Código referente a testes com o dataset Beans:

```
    ## Importamos as bibliotecas necessarias
    import tensorflow as tf
```

```
import tensorflow.keras.layers as tfl
from tensorflow import keras
import tensorflow_datasets as tfds
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

## Separacao dos dados
DATA_DIR = "/beans/input/"
(ds_train, ds_valid, ds_test), metadata = tfds.load("beans",
    split = ['train', 'validation', 'test'],
    data_dir = DATA_DIR,
    with_info = True, as_supervised = True)

## Funcao de pre-processamento
IMG_SIZE = 250
def preprocess(image, label):
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)
    image = tf.image.resize(image, [IMG_SIZE, IMG_SIZE])
    return image, label

def augment(image_e_label, seed):
    image, label = image_e_label
    image, label = preprocess(image, label)
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    image = tf.image.random_contrast(image, lower=0.0, upper=1.0)
    return image, label

## Tamanho do lote
BATCH_SIZE = 64
## Ajusta o valor dinamicamente no tempo
AUTO = tf.data.experimental.AUTOTUNE
## Tamanho do conjunto de treino para realizar o embaralhamento
SHUFFLE_BUFFER = int(metadata.splits['train'].num_examples * 0.8)

## Garante que cada imagem no conjunto de dados
## fique associada a um valor unico
counter = tf.data.experimental.Counter()
ds_train = tf.data.Dataset.zip((ds_train, (counter, counter)))

## Aplicando as normalizacoes nos conjuntos
ds_train = (ds_train.map(augment, AUTO)
```

```
                    .cache()
                    .shuffle(SHUFFLE_BUFFER)
                    .repeat()
                    .batch(BATCH_SIZE)
                    .prefetch(AUTO))

ds_valid = (ds_valid.map(preprocess, AUTO)
                    .cache()
                    .batch(BATCH_SIZE)
                    .prefetch(AUTO))

ds_test = (ds_test.map(preprocess, AUTO)
                    .cache()
                    .batch(BATCH_SIZE)
                    .prefetch(AUTO))

## Criacao do modelo convolucional Beans
def convolutional_model(input_shape):
    input_img = keras.Input(shape=input_shape)
    #Primeira camada CONV e POOL
    Z1=tfl.Conv2D(16, 4, strides = 1, padding = 'same')(input_img)
    A1=tfl.ReLU()(Z1)
    P1=tfl.MaxPool2D(pool_size=8, strides=2, padding='same')(A1)
    #Segunda camada CONV e POOL
    Z2=tfl.Conv2D(32, 2, strides = 1, padding = 'same')(P1)
    A2=tfl.ReLU()(Z2)
    P2=tfl.MaxPool2D(pool_size=4, strides=4, padding='same')(A2)
    #Terceira camada CONV e POOL
    Z3=tfl.Conv2D(64, 2, strides = 1, padding = 'same')(P2)
    A3=tfl.ReLU()(Z3)
    P3=tfl.MaxPool2D(pool_size=2, strides=8, padding='same')(A3)
    #Camada FC
    F=tfl.Flatten()(P3)
    outputs = tfl.Dense(3, activation="softmax", kernel_regularizer=
                                            'l1')(F)
    model = tf.keras.Model(inputs=input_img, outputs = outputs)
    return model

## Copilando o modelo
beans_model = convolutional_model((IMG_SIZE, IMG_SIZE, 3))
beans_model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
```

```
beans_model.summary()

## Treinamento do modelo
STEPS_PER_EPOCH = int(metadata.splits['train'].num_examples) //
                                        BATCH_SIZE
history = beans_model.fit(ds_train,
                          epochs=50,
                          validation_data=ds_valid,
                          steps_per_epoch=STEPS_PER_EPOCH)

## Acuracia do modelo
loss, acc = beans_model.evaluate(ds_test)
print("Accuracy", acc)

## Plotagem dos graficos
df_loss_acc = pd.DataFrame(history.history)
df_loss = df_loss_acc[['loss','val_loss']]
df_acc = df_loss_acc[['accuracy','val_accuracy']]
df_loss.plot(title='Model loss').set(xlabel='Epoch',ylabel='Loss')
df_acc.plot(title='Model Accuracy').set(xlabel='Epoch',ylabel='
                                        Accuracy')
```