Introdução à otimização aplicada ao aprendizado de máquina supervisionado

Edital:	**Edital Piic 2020/2021**
Grande Área do Conhecimento (CNPq):	**Ciências Exatas e da Terra**
Área do Conhecimento (CNPq):	**Matemática / Matemática Aplicada**
Título do Projeto:	**Métodos Computacionais em Otimização (9403/2019)**
Título do Subprojeto:	**Introdução à otimização aplicada ao aprendizado de máquina supervisionado**
Professor Orientador:	**Leonardo Delarmelina Secchin**
Estudante:	**Matheus Becali Rocha**

Resumo

Compõem o que chamamos de aprendizado de máquina, modelos matemáticos (no caso, um grafo – chamado "rede neural") e algoritmos capazes de otimizar os parâmetros da rede de modo a predizer 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 típicos usados para otimizar os dados da rede (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 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 a 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.

Palavras-chaves: Otimização. Aprendizado de máquina supervisionado. Método do gradiente estocástico. Inteligência Artificial. MNIST.

1 Introdução

O surgimento do aprendizado de máquina ocorreu com a primeira ideia sobre neurônio artificial, por volta dos anos 40, proposta por Warren McCulloch e Walter Pitts. Tratava-se de um modelo simplificado. Em 1958 surgiram os *perceptrons*, idealizados por Frank Rosenblatt que se inspirou no *Modelo McCulloch-Pitts*. Os *perceptrons* são classificadores binários que consistem em fluir suas entradas $x_1, x_2, \ldots$ para um único valor de saída $f(x)$, seguindo o modelo de propagação para frente (*feedforward*). Um *perceptron* é um neurônio artificial onde sua função de ativação é dada por uma função de degrau. Uma rede neural pode ser interpretada como um grafo, cujo fluxo flui da camada de entrada à camada de saída. Podemos pensar em uma rede neural como uma aplicação $x \to f(x)$, onde cada coordenada de x é o valor de um neurônio artificial na camada de entrada e $f(x)$ é a saída. Nesse sentido, é possível mostrar que qualquer função contínua pode ser arbitrariamente aproximada por uma rede neural.

No ramo da inteligência artificial existem inúmeras subdivisões. Algumas delas:

- Aprendizado de Máquina Supervisionado – modelos treinados utilizando dados externos para os quais sabemos a resposta para dada entrada. A rede então antevê a resposta à dados desconhecidos na fase de treinamento. Comumente utilizado para dados previamente rotulados (por exemplo, reconhecimento de tipos de carros);

- Aprendizado de Máquina Não Supervisionado – modelos que não necessitam que o usuário o supervisione, sem uso de dados de treinamento. Buscam descobrir padrões a partir do próprio comportamento do ambiente. Comumente utilizado para dados não rotulados (por exemplo, detecção de anomalias);

- Aprendizado por Reforço – modelos onde um agente opera em um ambiente e deve aprender a operar usando *feedback* (por exemplo, processamento de linguagem natural).

Esta pesquisa teve como ideia principal o estudo de modelos e técnicas de otimização utilizadas no aprendizado de máquina supervisionado. O método do gradiente estocástico surge como principal ferramenta na área, sendo de extrema valia para os primeiros estudos na área. Mais detalhes podem ser vistos em [Bottou et al., 2018].

Nesta pesquisa, buscamos comparar como diferentes métodos de otimização se comportam no treinamento de um modelo para classificação de imagens, onde foi utilizado um conjunto de dados pré-processado e de livre acesso MNIST [LeCun et al., 1998]. MNIST é um conjunto de imagens de caracteres numéricos de 0 a 9, dispostos em imagens de tamanho (28, 28) em escala cinza 8-bits. Foram utilizados os seguintes métodos:

- Gradiente estocástico (*Stochastic gradient descent*, SGD);

- Gradiente estocástico com momento (*Stochastic Gradient Descent with Momentum*, SGDM);

- Gradiente adaptativo (*Adaptive Gradient Algorithm*, AdaGrad);

- "Propagação da Raiz Quadrada Média" (*Root Means Square Propagation*, RMSProp);

- "Momento adaptável estimado" (*Adaptive Moment Estimation*, Adam).

Cada método acima foi proposto para diferentes finalidades. SGD e SGDM são métodos que utilizam a taxa de aprendizagem (isto é, o tamanho do passo na direção de $-\nabla f$) constante e os demais utilizam taxa de aprendizagem variável. Eles serão apresentados mais adiante. Também foi utilizado um conjunto de 32 imagens próprias, para verificar como a rede se comporta com dados de origem distinta da MNIST.

2 Objetivos

O objetivo geral deste subprojeto foi estudar técnicas em aprendizado de máquina supervisionado. Foram traçados e alcançados os seguintes objetivos específicos:

- Estudo dos conceitos básicos de IA/aprendizado de máquina, tendo como guia o artigo de Bottou, Curtis e Nocedal (2018). Textos auxiliares foram utilizados, como o livro de Goodfellow, Bengio e Courville (2016);

- Estudo/revisão de conceitos básicos de probabilidade necessários (variáveis aleatórias, distribuições de probabilidade, esperança, etc.). A referência foi o livro de James (2010);

- Implementação e testes computacionais visando a resolução/estudo de uma aplicação. A referência de partida é o recente artigo de Gambella, Chaddar e NaoumSawaya (2021).

3 Embasamento Teórico

Os elementos de fundamentação teórica deste subprojeto foram retirados de [Bottou et al., 2018], [Gambella et al., 2021], [LeCun et al., 1998] e [Sun et al., 2019]. Através destes elementos, foi construída uma base para realização das pesquisas sobre otimização aplicada ao aprendizado de máquina supervisionado e com, base em [Nielsen, 2015], obtemos conhecimento sobre a classificação de dígitos manuscritos. Isso possibilitou uma compreensão inicial sobre o assunto para uma posterior aprofundamento. Com base na bibliografia, vale destacar um trecho de extrema importância que serviu de partida para a pesquisa:

> "Quais foram os métodos de otimização mais bem-sucedidos para o aprendizado de máquina em grande escala e por quê?" Texto de tradução livre. [Bottou et al., 2018]

Inúmeros estudos sobre métodos de otimização e algoritmos levaram à referência [Sun et al., 2019], que introduziu diversos modelos para tentar responder à pergunta proposta acima. Sun et al., 2019 propôs uma análise sobre os métodos clássicos e modernos da otimização para o aprendizado de máquina. Uma explicação sobre como a matemática e o aprendizado de máquina andam juntos é dada por Bottou et al., 2018:

> "Um dos pilares do aprendizado de máquina é a otimização matemática, que, neste contexto, envolve o cálculo numérico de parâmetros para um sistema projetado para tomar decisões com base em dados ainda não vistos." Texto de tradução livre. [Bottou et al., 2018]

De acordo com Gambella et al., 2021, no contexto de classificação, a relação entre entrada e saída são descritas por uma distribuição de probabilidade $P(x,y)$. Seguindo essa mesma ideia, Bottou et al., 2018 diz que os dados de treinamento seguem essa mesma função de probabilidade. A importância de uma rede neural é visualizada na fala de Gambella et al., 2021:

> "Redes neurais com uma única camada e com um número finito de unidades podem representar qualquer função contínua multivariada." Texto de tradução livre. [Gambella et al., 2021]

Por fim, vale ressaltar que graças às bases teóricas, foi possível conduzir o estudo e sanar possíveis dúvidas que surgiram ao longo da pesquisa.

4 Metodologia

O desenvolvimento da pesquisa foi acompanhado/direcionado pelo orientador através de reuniões semanais. O trabalho foi conduzido com base no estudo, apresentação e discussão de artigos científicos, os principais [Bottou et al., 2018], [Gambella et al., 2021], [LeCun et al., 1998] e [Sun et al., 2019]. Uma aplicação foi proposta e realizada, via implementação em Python 3 de uma rede neural junto aos métodos de otimização propostos. As propostas foram atestadas e corrigidas na fase dos testes numéricos.

5 Resultados e Discussão

5.1 Problema alvo – classificação de caracteres numéricos

O problema de classificação de caracteres numéricos consiste em treinar o computador para que ele consiga reconhecer dígitos escritos à mão. No contexto do aprendizado de máquina supervisionado, é necessário que tenhamos

um banco de dados 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). Detalhamos o modelo na Seção 5.2.

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. Esse cálculo é realizado por uma técnica conhecida como *Backpropagation*. Essa técnica é fundamentada na regra da cadeia, e será explicada na Seção 5.3.

No processo de treinamento, precisamos entrar com os dados de treinamento e propagá-los pela rede até a camada de saída. O fluxo é guiado usando as chamadas funções de ativação. São funções não lineares associadas aos neurônios da rede e servem como “moduladoras” do fluxo. Dentre as funções de ativação, consideramos a Sigmoid e a ReLU (ver Seção 5.3).

Quando o algoritmo de otimização declara parada, a função de erro é 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, dentro de uma margem de acerto considerável (o ideal, claro, é 100% de acerto).

Um banco de imagem bastante conhecido é a MNIST ([LeCun et al., 1998]), que tem ao todo 70 mil de tamanho 28×28 *pixels* imagens divididas em um conjunto de treinamento com 60 mil imagens e um conjunto de teste com 10 mil imagens. É um subconjunto do banco de dados de caracteres e formulários manuscritos do Instituto Nacional de Padrões e Tecnologia dos EUA (NIST). Na Figura 1, podemos ver alguns dígitos presentes nesse conjunto de dados.

Figura 1: MNIST.

A entrada da rede neural é dada por um vetor x, que corresponde aos $28 \times 28 = 784$ *pixels* da imagem e fui até a camada de saída, o vetor $\hat{y}$ de tamanho $(1, 10)$. Originalmente os rótulos das imagens são apenas o valor presente na imagem, precisamos converter esses valores para podermos comparar com o nosso $\hat{y}$. Para isso, utilizamos a estratégia denominada *One-Hot-Encoding*, que transforma o rótulo da imagem para um vetor de tamanho $(1, 10)$ da forma $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$, onde 1 corresponde ao valor presente no rótulo.

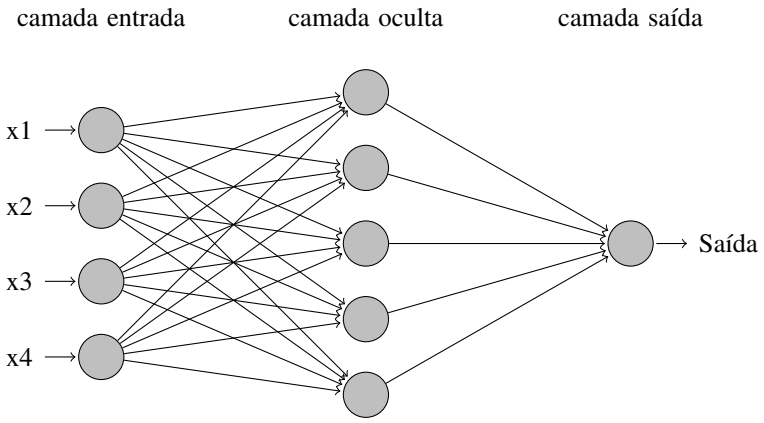
5.2 Redes Neurais

Uma rede neural é uma “réplica computacional” dos neurônios do cérebro humano e é um subconjunto de aprendizagem de máquina (*machine learning*), formada pelos neurônios artificiais. Os dois neurônios mais conhecidos e

importantes para o entendimento de redes neurais são os perceptrons e o sigmoid. Podemos escrever a saída desses neurônios de maneira geral por

$$\hat{y} = g\left(\sum_{i=1}^{m} w_{ij}x_i + b_i\right)$$

onde w e x são vetores cujas componentes são chamadas **peso** e **entrada**, b um vetor, chamado **viés** e g a função de ativação não-linear, aqui podendo ser a sigmoid ou a de degrau, presente no perceptron. A diferença entre os neurônios sigmoid para o neurônios perceptron está em sua função ativação, o perceptron ao sofrer uma pequena variação nos pesos e viés sofrem grandes mudanças ao longo da rede, diferentemente, a função de ativação os neurônios sigmoid é suave, possibilitando que pequenas alterações não tenham grande influência nas decisões ao longo da rede. Atualmente os perceptrons não são muito utilizados em rede neurais, mas sim como um método de pesar evidências para tomar decisões, presentes em portas lógicas. Uma rede neural artificial é dada por diversos neurônios artificiais dispostos em diferentes camadas, como no esquema a seguir:



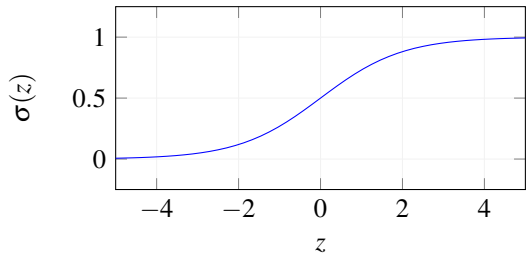
Na figura acima, a camada de entrada recebe um vetor x. Ao neurônio i da camada l são associados um vetor de pesos $w_{ij}^{[l]}$ e o viés $b_i^{[l]}$. Esses objetos são iniciados aleatoriamente, e ajustados por um algoritmo de otimização ao minimizar a função de erro. Os dados fluem, passando pela camada oculta, e avança até a camada de saída com os parâmetros ajustados, retornando a previsão do modelo.

5.2.1 Funções de ativação

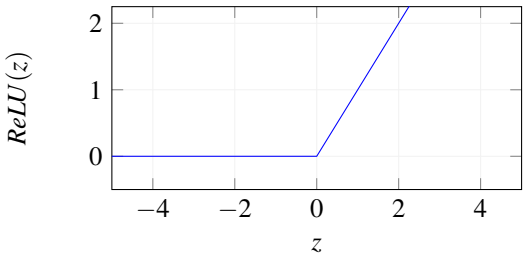
Uma função de ativação ou função de transferência, tem, dentre outras, a finalidade de evitar o acréscimo progressivo dos valores de saída ao longo das camadas da rede, visto que tais funções possuem valores máximos e mínimos contidos em intervalos determinados. De certa forma, elas "modulam" o fluxo para uma correta saída. A Figura 2 traz algumas das principais funções de ativação, cujas derivadas são exibidas no quadro abaixo.

Nome	Sigmoid (σ)	*ReLU*	Tan. hip. (tanh)	LeakyReLU (*lReLU*)
Expressão	$\dfrac{1}{1+e^{-z}}$	$\max(0,z)$	$\dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$\max(10^{-1}z,z)$
Derivada	$\sigma(z)(1-\sigma(z))$	$\begin{cases} 1, & z>0 \\ 0, & \text{c.c.} \end{cases}$	$1-\tanh(z)^2$	$\begin{cases} 1, & z>0 \\ 10^{-2}, & \text{c.c.} \end{cases}$

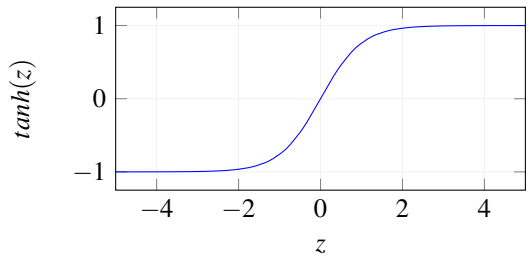
Figura 2: Funções de ativação.



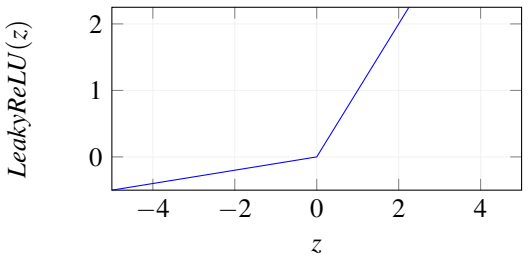
(a) Função ativação Sigmoid.



(b) Função ativação ReLU.



(c) Função ativação tangente hiperbólica.



(d) Função de ativação Leaky ReLU.

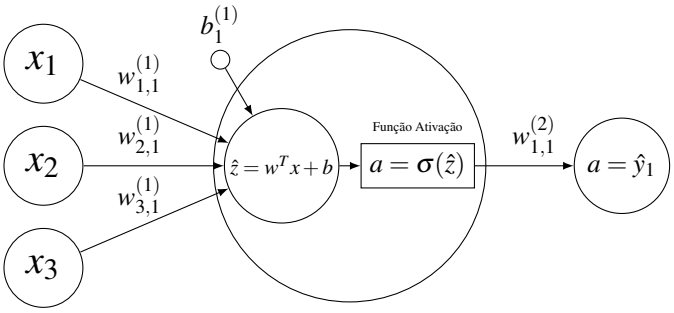
5.3 Treinamento da rede neural

O processo de treinamento de uma rede neural consiste em atualizarmos os pesos e viés da rede, para que na próxima iteração ela consiga nos retornar melhores resultados. 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 o esquema mais usado: esquemas de gradiente.

5.3.1 Propagação pela rede (*feedforward*)

O propósito do *feedforward* 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. Nele são realizados duas operações fundamentais, de maneira recursiva 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, nossas previsões). A Figura 3 ilustra como é realizado o processo dentro de um neurônio.

Figura 3: Propagação do fluxo para cálculo de (w,b) (*feedforward*).



Entre as camadas ocultas pode-se definir funções de ativação diferentes. Por exemplo, uma rede pode ser estruturada como sendo (ENTRADA $\rightarrow$ ReLU $\rightarrow$ Sigmoid $\rightarrow$ SAÍDA).

Após o processo de propagação pela rede, ocorre o cálculo de gradientes por *backpropagation*, onde percorremos novamente a rede, contudo no sentido contrário. Neste processo, aplicamos 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.

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

As funções de erro mais conhecidas na literatura são o **erro quadrático médio**, que será considerada na próxima subseção, e a **regressão logística**, para mais detalhes [Nielsen, 2015]. 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.

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

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

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

Aqui, o ponto

$$z = (w_{11}, w_{21}, \ldots, w_{784,1}, w_{12}, \ldots, w_{784,2}, \ldots, w_{1m}, \ldots, w_{784,m}, b_1, \ldots, b_m) \in \mathbb{R}^{m+1)\times 784}$$

foi computado percorrendo a rede da camada de entrada até à camada de saída (*feedforward*), conforme a Figura 3. O algoritmo *backpropagation* com a função de ativação Sigmoid σ (dada na Seção 5.2.1), é descrito como segue.

Definimos o erro δ_j^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 j dependem apenas de $z_j^{[L]}$, e o resultado é dado por

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

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

$$\delta_i^{[L]} = \sum_j \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial \sigma(z_j^{[L]})}{\partial z_j^{[L]}} = \sum_j \frac{\partial C}{\partial a_j^{[L]}} \sigma'(z_j^{[L]}).$$

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 $z_j^{[l+1]} = \sum_i w_{ji}^{[l+1]} a_i^{[l]} + b_j^{[l+1]} = \sum_i w_{ji}^{[l+1]} \sigma(z_i^{[l]}) + b_j^{[l+1]}$. Derivando $z_j^{[l+1]}$ obtemos

$$\frac{\partial z_j^{[l+1]}}{\partial z_i^{[l]}} = w_{ji}^{[l+1]} \sigma'(z_i^{[l]}) \implies \delta_i^{[l]} = \sum_j w_{ji}^{[l+1]} \sigma'(z_i^{[l]}) \delta_j^{[l+1]}.$$

Agora com os erros $\delta_i^{[l]}$ de todas as camadas da rede neural, conseguimos calcular as derivadas

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

5.3.3 Cálculo eficiente do gradiente (*back-propagation*) – regressão logística

A função de Custo da Regressão logística é definida por:

$$C(w,b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log a(x^{(i)}; w, b) + (1 - y^{(i)}) \log(1 - a(x^{(i)}; w, b)) \right]. \tag{2}$$

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, 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,

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

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

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

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

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

Juntando as equações e fazendo o mesmo para as derivadas parciais com respeito à b, obtemos

$$\frac{\partial C}{\partial w_{ij}^{[l]}} = \sum_{i=1}^{m} \left(a_i^{[l]} - y_i \right) x_i^{[l]} \quad \text{e} \quad \frac{\partial C}{\partial b_i^{[l]}} = \sum_{i=1}^{m} \left(a_i^{[l]} - y_i \right) \cdot 1.$$

5.4 Métodos para treinamento baseados em gradientes

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. No contexto de aprendizado de máquina, η_k é chamado *taxa de aprendizagem*. Os métodos mais utilizados consistem em variações do esquema anterior. A seguir uma rápida revisão dos principais algoritmos.

5.4.1 Método do gradiente estocástico – SGD

O método SGD ([Bottou et al., 2018]) busca evitar que todos os dados de treinamento sejam utilizados de uma só vez como no método do gradiente, pois do contrário resultaria em processo com alto custo computacional. 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 entra escolhas aleatórias de parte dos gradientes da soma em (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. Este subconjunto pode ser apenas um único termo ou um conjunto pequeno dos dados. O SGD é descrito no Algoritmo 1.

Algoritmo 1 Método do gradiente estocástico – SGD

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Faça $k = 0$.
para $k = 1, \ldots, k_{\max}$ **faça**
    Selecione os lotes $B_0, \ldots, B_{p-1}$ dos dados de tamanho m
    $z^{k,0} = z^k$
    para $i = 0, \ldots, p-1$ (número de lotes) **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$, onde C_j é o termo em (1) referente à $j \in B_i$
        Incremente os pesos e vieses: $z^{k,i+1} = z^{k,i} - \eta \nabla C_{B_i}(z^{k,i})$
    fim para
    Atualize os pesos e vieses: $z^{k+1} = z^{k,p}$
fim para
Retorne os pesos e vieses finais $z^k = (w^k, b^k)$.

Caso $|B_k| = 1$ para todo k chamamos o método simplesmente de gradiente estocástico, caso $1 < |B_k| < m$ chamados o método de SGD em lote e por fim, caso $|B_k| = m$ para todo k, recaímos no método do gradiente clássico sobre (1).

5.4.2 Método do gradiente estocástico com momento

O SGDM, proposto por [Sutskever et al., 2013], utiliza a ideia do momento presente na física, que simula a trajetória de um objeto em movimento. É proposto com o intuito de contornar um dos problemas presentes no SGD: convergência a mínimos locais ruins e pontos de sela. Acelerando a convergência e reduzindo as oscilações presentes no SGD, seu algoritmo é descrito no Algoritmo 2.

Algoritmo 2 SGD com Momento

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $V^{-1} = 0$.
para $k = 1, \ldots, k_{\max}$ **faça**
    Selecione os lotes $B_0, \ldots, B_{p-1}$ dos dados de tamanho m
    $z^{k,0} = z^k, \quad V^{k,-1} = V^k$
    para $i = 0, \ldots, p-1$ (número de lotes) **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$, onde C_j é o termo em (1) referente à $j \in B_i$
        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 $z^k = (w^k, b^k)$.

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, enquanto se $\beta > 0$, V^k leva em consideração a velocidade na iteração anterior. Usualmente tomamos $\eta = 0{,}001$ e $\beta = 0{,}9$.

5.4.3 AdaGrad

O AdaGrad, proposto por [Duchi et al., 2011], é um método com taxa de aprendizado adaptativa. 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. Assim, ele evita o ajuste manual da taxa de aprendizagem e acelera a convergência.

Algoritmo 3 AdaGrad

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $G^{-1} = 0$.
para $k = 1, \ldots, k_{\max}$ **faça**
  Selecione os lotes $B_0, \ldots, B_{p-1}$ dos dados de tamanho m
  $z^{k,0} = z^k, \quad G^{k,-1} = G^k$
  para $i = 0, \ldots, p-1$ (número de lotes) **faça**
    Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$, onde C_j é o termo em (1) referente à $j \in B_i$
    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: $z^{k,i+1} = z^{k,i} - \frac{\eta}{\sqrt{G+\varepsilon}} \nabla C_{B_i}(z^{k,i})$
  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 $z^k = (w^k, b^k)$.

G é o quadrado da norma dos gradientes acumulados até a iteração atual. No algoritmo, adotamos os parâmetros $\eta = 0,001$ e $\varepsilon = 10^{-6}$. 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 $G \approx 0$.

5.4.4 RMSProp

O RMSProp é um algoritmo não publicado proposto por, Geoff Hinton, que entra com a ideia de melhorar o AdaGrad. Ele não acumula todos os gradientes e foca somente nos gradientes em um certo tempo. Sua principal abordagem é através da propagação média móvel quadrática para realizar o cálculo do momento. Ele contorna o problema de aprendizagem presente no final do AdaGrad, que conforme o passar das iterações do método AdaGrad, o gradiente se torna muito grande, levando a taxa de aprendizagem a zero. O algoritmo é semelhante ao AdaGrad, com uma pequena modificação em G, que adicionamos β visto na seção 5.4.2 na soma das normas dos gradientes. Assim, a alteração ficaria como: $G = \beta G + (1-\beta) \left\| \frac{\partial C_{B_k}(z^k)}{\partial z} \right\|^2$.

5.4.5 Adam

O Adam, proposto por [Kingma & Ba, 2014], é um método que utiliza taxa de aprendizagem adaptativa. Ele mescla as ideias principais dos algoritmos RMSProp e SGD com Momento, isto é, emprega taxa adaptativa escalada pelas normas dos gradientes calculados e o momento. 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.

Os parâmetros de referência são $\eta = 0,001$, $\beta_1 = 0,9$, $\beta_2 = 0,999$ e $\varepsilon = 10^{-8}$.

5.4.6 Outros métodos

Existem diversos outros métodos interessantes que não foram utilizados como ferramenta principal dos estudos, tais como: *Nesterov Accelerated Gradient Descent* (NAG), utiliza o mesmo conceito que o SGD com Momento, com a diferença de que ao invés de atualizar o gradiente atual, ele realiza um passo para frente utilizando o momento anterior para atualizar o próximo gradiente. *Stochastic Average Gradient* (SAG), proposto para melhorar a velocidade de convergência, porém só pode ser aplicado em funções de perda suaves e convexas; *Stochastic*

Algoritmo 4 Algoritmo Adam

Inicialize os pesos e vieses $z^0 = (w^0, b^0)$ randomicamente. Tome $G^{-1} = 0$ e $V^{-1} = 0$.
para $k = 1, \ldots, k_{\max}$ **faça**
    Selecione os lotes $B_0, \ldots, B_{p-1}$ dos dados de tamanho m
    $z^{k,0} = z^k, \quad G^{k,-1} = G^k, \quad V^{k,-1} = V^k$
    para $i = 0, \ldots, p-1$ (número de lotes) **faça**
        Calcule $\nabla C_{B_i}(z^{k,i}) = \frac{1}{|B_i|} \sum_{j \in B_i} \nabla C_j(z^{k,i})$, onde C_j é o termo em (1) referente à $j \in B_i$
        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: $z^{k,i+1} = z^{k,i} - \frac{\eta}{\sqrt{\hat{G}^{k,i}+\varepsilon}}\hat{V}^{k,i}$
    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 $z^k = (w^k, b^k)$.

Variance Reduction Gradient (SVRG), foi proposto para melhorar o desempenho da otimização de modelos complexos. Mais detalhes em [Sun et al., 2019].

5.5 Testes numéricos

Os testes numéricos foram realizados na linguagem Pyhton 3 e executados em um servidor contendo um, Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz 10 núcleos (20 threads), 160Gb RAM. Para a construção do código próprio, foi tomado como referência os códigos de Michael A. Nielsen (`http://neuralnetworksanddeeplearning.com`) e de Kyohei Sahara (`https://github.com/schwalbe10/ThinkageDeepLearning`). Para geração dos pesos iniciais, foi utilizado a técnica de geração aleatória com uma semente fixa para comparação justa entre os métodos.

A Tabela 1 contém os tempos e a precisão de cada método ao ser executado com os dados de testes que foram retirados da própria MNIST, ao todo 10.000 exemplos ou 14,28% do total de imagens nunca observadas pela rede. Todos os testes foram realizados com $\eta = 0.001$ e com a camada oculta variando entre 300, 200 e 100, na tabela representada pela sigla HL.

Tabela 1: Resultados dos testes na MNIST.

Método	**HL**	**Tempo (s)**	**Acerto (%)**	**Método**	**HL**	**Tempo (s)**	**Acerto (%)**
SGD	300	7608.94	89.73	RMSProp	300	9739.17	97.63
	200	5382.59	89.84		200	6938.55	97.38
	100	3304.11	89.89		100	4107.94	97.17
SGDM	300	8827.15	94.14	Adam	300	12334.39	98.07
	200	6240.75	94.52		200	8548.00	97.65
	100	3718.60	94.65		100	4975.84	97.27
AdaGrad	300	8682.42	94.84				
	200	6271.35	94.72				
	100	3742.94	93.92				

Note que nos métodos SGD e SGD com Momento, ao aumentarmos as camadas ocultas, a porcentagem de acertos diminui. Isso se deve ao fato da taxa de aprendizagem ser constante, ao se aproximar do mínimo local o passo pode ser grande e acabar por se afastar do mínimo local. Para contornarmos esse problema, uma solução é diminuir a taxa de aprendizagem, porém acarretaria uma convergência lenta. Ao avaliarmos os métodos com taxa de aprendizagem variável, esse problema não ocorre, tendo uma melhora em todos os métodos presentes.

Na Figura 4 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 (*epoch*, as iterações externas dos métodos). O eixo vertical está em escala logarítmica para evidenciar erros próximos de zero. Os métodos foram executados com 300 camadas ocultas 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.

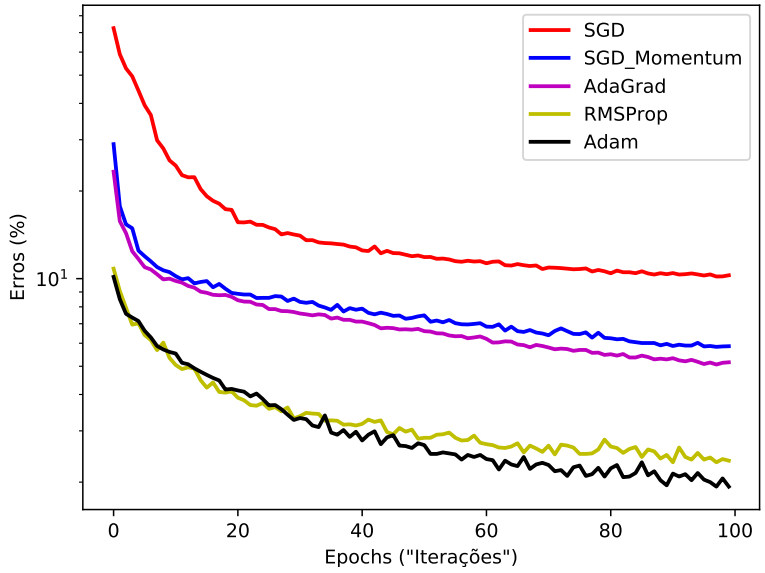


Figura 4: Gráfico comparativo entre os métodos em 300 HL e 100 epochs

6 Conclusões

Na subseção 5.5, apresentamos os resultados e alguns comparativos entre os métodos. Podemos concluir que o método que mais se destacou em relação à eficácia foi o Adam com 300 HL, 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 HL se destaca, com $\approx 97{,}3\%$ de acerto no caso de teste e $\approx 1{,}38$ horas de execução.

Realizamos testes com 32 dígitos próprios. No método Adam obtivemos uma porcentagem de acertos igual a 71.88%. A rede não conseguiu identificar os dígitos 9, talvez pelo fato da escrita diferir dos dados de treinamento, algo plausível.

Na Figura 5, temos uma visualização dos acertos referentes a dois números presentes no conjunto de testes da MNIST, dados retirados da rede treinada com o método SGD com 300 HL. Nota-se que a rede se confunde mais com o número 4 e 7 ao tentar reconhecer o digito 9. Isso ocorre pois dependendo da escrita, podem acabar ficando muito parecidos e talvez a rede não tenha muitos dados para treinar nesse tipo de caso. 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.

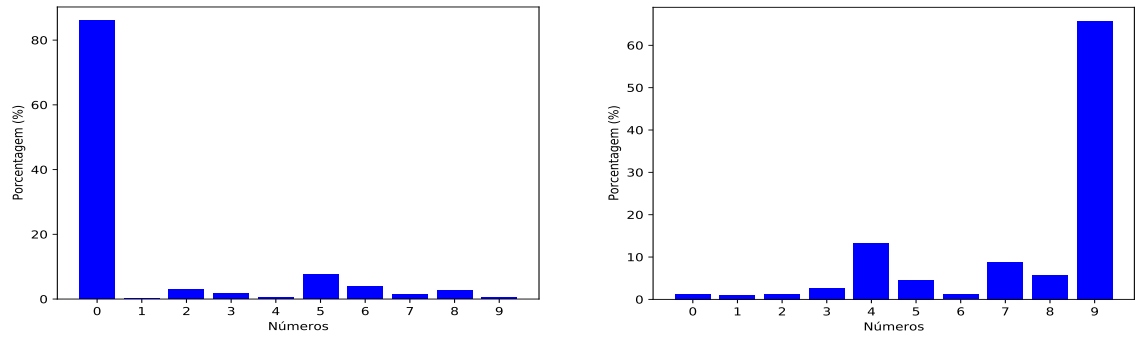


Figura 5: Repostas da rede para MNIST. À esquerda, respostas à 0; à direita, respostas à 9.

Por fim, o projeto contribuiu bastante para minha formação acadêmica. Através dele realizei inúmeros cursos livres visando o aprendizado de novas técnicas, tais como, rede convolucionais e processamento de linguagem natural (PLN). Além disso, foi realizado um seminário *online*, aberto aos alunos de graduação dos cursos de exatas da UFES/São Mateus sobre o tema estudado.

6.1 Trabalhos futuros

Para trabalhos futuros, buscaremos a elaboração do Trabalho de Conclusão de Curso e iremos mais a fundo no aprendizado de máquina. Utilizaremos novos *datasets* e introduziremos redes convolucionais com a ajuda da API de código aberto Tensorflow, um ambiente de código aberto para criar modelos e simulações em *Machine Learning*.

Agradecimentos

Este trabalho teve o apoio da FAPES (processo 116/2019), e da UFES (edital 003/2019 DP/PRPPG, FAP linha I).

Referências Bibliográficas

[Bottou et al., 2018] Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311.

[Duchi et al., 2011] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

[Gambella et al., 2021] Gambella, C., Ghaddar, B., & Naoum-Sawaya, J. (2021). Optimization problems for machine learning: A survey. *European Journal of Operational Research*, 290(3):807–828.

[Kingma & Ba, 2014] Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Nielsen, 2015] Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

[Sun et al., 2019] Sun, S., Cao, Z., Zhu, H., & Zhao, J. (2019). A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681.

[Sutskever et al., 2013] Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR.