Universidade Estadual de Campinas

Institute of Mathematics, Statistics and Scientific Computing

(IMECC - Unicamp)

Projeto Computacional 2:

Classificação de Dígitos do *MNIST* Utilizando Redes Neurais com os métodos do *gradiente descendente*, *gradiente conjugado* e *checagem do gradiente*

Aluno : Gabriel Borin Macedo || RA : 197201
Aluno : Matheus Araujo Souza || RA : 184145
Aluno : Raul Augusto Teixeira || RA : 205177

Agosto 2020

Resumo

Nosso projeto visa o estudo dos algoritmos e implementação dos métodos de redes neurais e suas aplicações para reconhecimento de conjuntos do modelo MNIST, obtivemos vários resultados em diferentes aspectos, e podemos então escolher no final o melhor modelo o gradiente descente, também adaptamos nosso código para várias camadas internas, além disso também implementamos outro método de descida o Gradiente conjugado, método esse que se saiu muito bem para os exemplos de teste porem tem resultados muito custosos computacionalmente.

1 Estrutura matemática

1.1 Redes Neurais do tipo *perceptron*

Redes neurais são modelos matemáticos usados para predição de valores, localização de algum objeto ou para classificação de objetos a partir de um dado de entrada (*input*). Nos quais, as entradas para o modelo podem ser de valores discretos, por exemplo valores discretos de uma área de um terreno, ou até mesmo uma imagem em padrões *RGB*. As redes neurais são compostas por unidades denominadas neurônios, nas quais funcionam recebendo dados da forma $x_1, x_2, ..., x_n$. Note que esse processamento é uma combinação linear dos dados de entrada multiplicados pelos pesos de cada neurônio, seguido pela aplicação de uma *função de ativação f(v)*. A seguinte imagem representa essa ideia

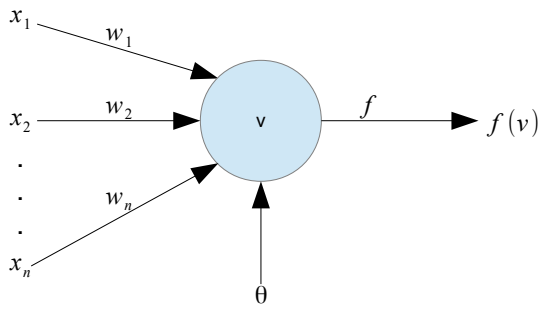


Figure 1: Representação de um perceptron (rede neural de um neurônio apenas).

e que essa combinação linear é da forma

$$v = \sum_{i=1}^{n} x_i w_i + \theta \tag{1}$$

em que w_i é o $i-$ésimo peso associado a cada conexão neural e θ é uma constante. Essa estrutura composta de apenas 1 neurônio (imagem 2) é denominada *perceptron* na literatura e θ é chamado de *bias*.

O parâmetro de *bias* serve para facilitar ou dificultar a "ativação" de um neurônio. Considere, por exemplo, o caso da função *step*:

$$f(v) = \begin{cases} 0 & \text{se Eq(1)} \leq t \\ 1 & \text{caso contrário} \end{cases}$$

Onde t é um parâmetro arbitrário escolhido na implementação do modelo. Além disso, um fator interessante sobre a influência do **bias** nesse modelo é que se um perceptron possui como função de ativação a função *step* e um valor de *bias* grande, ele tem grandes chances de produzir 1 como o resultado final.

1.2 Redes neurais com mais camadas e modelos totalmente conectados

Uma rede composta por vários neurônios estes podem ser combinados para resolver um problema, fazendo com que aprendam pesos e *biases* que realizem uma predição ou classificação com o menor erro possível. Uma boa estratégia para fazer um bom modelo é garantir que uma pequena mudança nos pesos implique em uma pequena variação do *output* final da rede.

Essa estratégia serve como base para que o modelo modifique seus pesos visando minimizar seu erro e obter assim uma máxima precisão na tarefa de predição ou classificação. Esse procedimento de mudança dos pesos para minimizar o erro é chamado de *aprendizado* (*learning*).

Para esse trabalho, é interessante adotar a função *f dá camada de saída* sendo a *função logística* ou também conhecida na literatura como *Softmax*. Essa função foi escolhida pois a *função logística* apresenta um excelente desempenho para modelos que precisam classificar mais de 2 classes de objetos. Além disso, será implementado apenas redes neurais com camadas escondidas que contém a mesma quantidade de neurônios.

1.3 Regressão logística

Esse tipo de técnica é utilizada quando é necessário classificar objetos em um conjunto de $k-$classes, apenas utilizando uma faixa de valor entre 0 e 1. Um aspecto interessante sobre essa técnica é que toda essa teoria foi baseada utilizando a metodologia da regressão linear e não linear e baseando em alguns teoremas estatísticos. Dessa forma, é definido a função g sendo

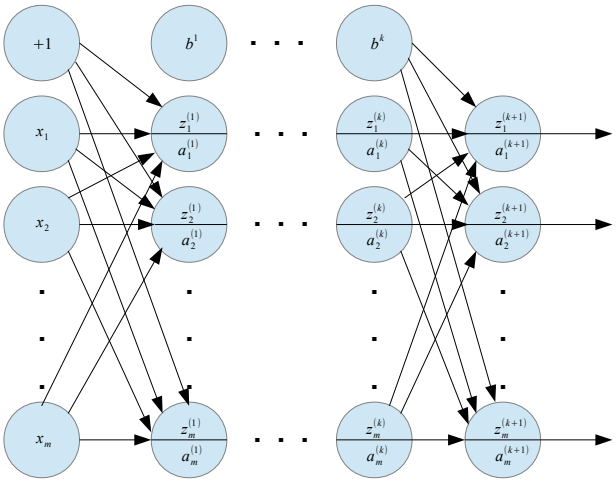


Figure 2: Representação de uma rede neural totalmente conectada que possui $k-$camadas escondidas com $m-$neurônios em cada e que classifica $m-$classes.

$$g(\Theta^T X) = \frac{1}{1 + e^{-(\Theta^T X)}}$$

onde Θ é a matriz dos pesos e X é a matriz de *design* e é da forma $X_{n \text{ x } m}$. Além disso, é definido a função de perda utilizando o teorema do *Maximum likelihood estimation*, temos a expressão

$$P(y = a|X; \theta) = (g(\Theta^T X))^a (1 - g(\Theta^T X))^{1-a}$$

na qual será útil para realizarmos a classificação das multi-classes.

1.3.1 Predição da classe dos dados

Se tivermos $k-$classes para classificarmos, iremos tomar $k-$classificadores que tentam melhor se aproxima para aquela classe do objeto. Com isso, temos a função para cada classe será da forma

$$h_\theta^{(i)}(X) = P(y = i|X; \theta); \; \forall \; i \in [1, k] \text{ e } i \in N.$$

Além disso, é criado um vetor onde cada índice dele corresponde a classe do objeto classificado. Ou seja, seja o vetor $v_{classe} = [1, 2, 3, \ldots, k]$ e o vetor das probabilidade de cada classificador sendo $v_{prob} = [h_\theta^{(1)}(X), h_\theta^{(2)}(X), h_\theta^{(3)}(X), \ldots, h_\theta^{(k)}(X)]$.

Dessa forma, temos que a classe prevista é da forma $\arg\max(v_{prob})$ que corresponde ao índice do vetor da classe procurada.

1.3.2 Função de perda para redes neurais

Visto em sala de aula, sabemos que cada neurônio de saída faz uma *regressão logística* que é definida pela equação

$$J(\Theta) = \frac{-1}{m}[\sum_{i=1}^{m} y^{(i)} \log{(h_\theta(X^{(i)}))}(1 - y^{(i)}) \log{(1 - h_\theta(X^{(i)}))}].$$

A partir desse resultado, temos que a função de custo da rede é descrita sendo a soma do custo de todas as K saídas. Portanto, é feito a equação

$$J(\Theta) = \frac{-1}{m}[\sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log{(h_\theta(X^{(i)})_j)}(1 - y_j^{(i)}) \log{(1 - h_\theta(X^{(i)})_j)}]$$

onde $(h_\theta(X^{(i)})_j)$ é a $j-$ésima saída da função de hipótese e y_j é o $j-$ésimo componente do vetor de saída de y. Com isso, podemos reescrever a fórmula da função de perda apenas para um único dado que é da forma

$$l(X^{(i)}, \Theta) = -[y^{(i)} \log{(h_\theta(X^{(i)}))}(1 - y^{(i)}) \log{(1 - h_\theta(X^{(i)}))}] \tag{2}$$

essa função será importante para descrever o processo do *gradiente conjugado*.

1.4 Vetorização das redes neurais

É possível implementar um processo no qual toda a rede neural é vetorizada. Dessa forma, melhorando ainda mais o desempenho das redes neurais. Esse procedimento é feito de seguinte forma : um *output* (dado de saída) de um neurônio da rede neural na $l-$ésima camada de um modelo pode ser descrito, para qualquer $l \geq 1$, sendo

$$z^{(l)} = \Theta^{(l-1)} a^{(l-1)} + b^{(l-1)}$$
$$a^{(l)} = f(z^{(l)})$$

Onde $z^{(l)}$ é a combinação dos dados antes de entrar na função $f(z^{(l)})$, $a^{(l)}$ é o vetor de ativação da camada l e b^l é o vetor de *bias* da camada l. Além disso, temos que a camada de entrada é representada pelo valor 0 e o vetor de ativação é descrito como $x^{(i)} = a^{(0)}$, ou seja

$$a^{(0)} = x^{(i)} = \begin{vmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_m^{(i)} \end{vmatrix}$$

Por fim, temos que $s = \theta^q a^q$, com q sendo o índice da última camada do modelo.

1.5 Atualização de pesos em redes neurais

Para este trabalho, será utilizado as metodologias do *gradiente descendente*, do *gradiente conjugado* para atualização dos pesos da rede neural durante a fase de treinamento e da *checagem do gradiente para uma rede neural menor*. Além disso, será comparado os desempenhos desses modelos.

1.5.1 *Gradiente descendente* e sua vetorização

Nesse técnica, todos os pesos θ serão atualizados começando do fim do modelo e indo até o *layer* de *input* da rede. Para a última camada do modelo, iremos definir que

$$\delta_i^{(q)} = a_i^{(q)} - y.$$

onde $\delta_i^{(q)}$ é o erro do neurônio i na camada q, $a_i^{(q)}$ é a saída do neurônio i na camada l, y é o vetor com o valor verdadeiro da classificação e q é a última camada do modelo.

Agora, ainda é necessário calcularmos os valores de $\delta_i^{(l)}$ das outras camadas. Dessa forma, temos que

$$\delta_i^{(l)} = (\sum_{j=1}^{n_{l+1}} \theta_{ij}^{(l)} \delta_j^{(l+1)}) \frac{\mathrm{d}g(z_i^{(l)})}{\mathrm{d}z_i^{(l)}}$$

onde n_{l+1} corresponde a quantidade de neurônios da camada $l+1$. Além disso, lembrando que foi visto em aula a dedução da derivada de $\frac{\mathrm{d}g(z_i^{(l)})}{\mathrm{d}z_i^{(l)}}$ que é da forma

$$\frac{\mathrm{d}g(z_i^{(l)})}{\mathrm{d}z_i^{(l)}} = g(z_i^{(l)})(1 - g(z_i^{(l)})) = a_i^{(l)}(1 - a_i^{(l)}).$$

Assim, se substituirmos o valor dessa derivada em $\delta_i^{(l)}$, é possível obter a relação

$$\delta_i^{(l)} = (\sum_{j=1}^{n_{l+1}} \theta_{ij}^{(l)} \delta_j^{(l+1)}) a_i^{(l)}(1 - a_i^{(l)}).$$

Ainda é possível vetorizar o processo do cálculo dos $\delta_i^{(l)}$. Com isso, temos a equação vetorizada

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* a^{(l)} .* (1 - a^{(l)}).$$

Onde $.*$ é o produto ponto-a-ponto e a equação vetorizada será a equação que será utilizada para o método gradiente, devido a sua praticidade e agilidade durante as operações. Por fim, é definido a derivada da função de perda sendo

$$(\frac{\partial J(\Theta)}{\partial \Theta})_{lij} = a_j^{(l)} \delta_i^{(l+1)}.$$

Note que esse resultado é válido apenas para um *dataset* composto de um único dado. Entretanto, o processo de atualização dos pesos e *bias* ainda serão os mesmo. Com isso, é necessário algumas mudanças para o funcionamento em um conjunto de dados. Assim, dado um conjunto de dados que é da forma $(x^{(i)}, y^{(i)})$ onde $x^{(i)}$ é o dado de entrada e $y^{(i)}$ é a resposta do dado $x^{(i)}$. Com isso, temos o algoritmo

Algorithm 1: Pseudo - Código para o método *gradiente descendente*

Dado o conjunto de dados $\{(x^{(0)}, y^{(0)}, \ldots, (x^{(m)}, y^{(m)}))$
$\Delta_{ij}^{(l)} = 0 \ \forall \ l, i, j$
for *i=0:m ; i = i + 1* **do**
  $a^{(0)} = x^{(i)}$
  $\delta^{(q)} = a^{(q)} - y$
  for *l=q - 1:1 (quantidade de camadas)* **do**
    $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* a^{(l)} .* (1 - a^{(l)})$
    $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
    for *j=0:MAX_j* **do**
      if *Se $j \neq 0$* **then**
        $D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \theta_{ij}^{(l)}$
      else
        $D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$
      end
      $(\frac{\partial J(\Theta)}{\partial \Theta})_{ij} = D_{ij}^{(l)}$.
    end
  end
end

Por fim, os pesos e os bias serão atualizados seguindo a seguinte regra

$$\Theta = \Theta - \epsilon \frac{\partial J(\Theta)}{\partial \Theta},$$

sendo ϵ a taxa de aprendizado. Neste trabalho, será adotado $\epsilon = 0.001$

Esse método usualmente é o mais escolhido como metodologia de atualização dos pesos. Isso se deve pois esse algoritmo apresenta uma série de vantagens em relação a outros algoritmos minimizadores. Algumas dessas vantagens são as seguintes : apresenta uma convergência mais estável e com um erro menor comparado com outros algoritmos minimizadores, é possível vetorizar esse método, a escolha dos pesos que minimizam o erro são mais diretos para se encontrar e computacionalmente é mais eficiente comparado com outros algoritmos.

As desvantagens do *gradiente descendente* é que os pesos podem convergir para pontos de mínimos ou de celas que atrapalham na acurácia do modelo.

1.5.2 Método do *gradiente conjugado* para atualização dos pesos da rede

O princípio desse método é resolver um sistema linear de equações ou que apresentam equações quadráticas.

Assim, a função que queremos minimizar será a função $J(\Theta)$. Entretanto, iremos calcular a função de custo em intervalos de t-dados denominados como *mini-batch* (subconjunto dos dados de treinamento que contém t dados). Assim, dado um subconjunto de dados X, a função de perda nesse subconjunto é da forma

$$L(\Theta) = \frac{1}{|X|} \sum_{x \in X} l(x, \Theta)$$

onde $|X|$ é a cardinalidade do subconjunto de dados $|X|$ (em outras palavras, a quantidade de dados em X) e $l(x, \Theta)$ é a função de perda para um único dado descrita pela Eq[2]

Entretanto, antes de realmente começar o algoritmo. É necessário utilizar um algoritmo que irá ajudar no processo do cálculo do *gradiente conjugado*. A seguir, é definido o cálculo da matriz pré-condicional que será da forma

Algorithm 2: Pseudo - Código para o cálculo da matriz pré-condicionada (MPC)

Dado Θ, Θ_{ant}, $\nabla L(\Theta)$, $\nabla L_{ant}(\Theta_{ant})$, e o passo t
if $t = 0$ **then**
  $H_{diag} = Id$ (Matriz identidade)
else
  $y = \nabla L(\Theta) - \nabla L_{ant}(\Theta_{ant})$
  $s = \Theta - \Theta_{ant}$
  $H_{diag} = H_{diag} + \frac{Diag(yy^T)}{y^T s} - \frac{H_{diag}\ Diag(ss^T)\ H_{diag}^T}{s^T\ H_{diag}\ s}$
$\Theta_{ant} = \Theta$
$\nabla L_{ant}(\Theta_{ant}) = \nabla L(\Theta)$
retorne $H_{diag}^{-1}, \Theta_{ant}, \nabla L_{ant}(\Theta_{ant})$

Dessa forma, temos o algoritmo do *gradiente conjugado* que será da forma

Algorithm 3: Pseudo - Código para o cálculo do *gradiente conjugado*.

$M^{-1}, \Theta_{ant}, \nabla L_{ant}(\Theta_{ant}) = \text{MPC}(\Theta, 0, -\nabla L(\Theta), 0, 0)$
$s = M^{-1}r$
$d = s$
$\delta_{novo} = r^T d$
$\nabla L_{ant}(\Theta_{ant}) = \nabla L(\Theta)$
for $t = 0{:}t_{max}$; $t = t + 1$ **do**
  $\Theta = \Theta + \epsilon d$
  $r = -\nabla L(\Theta)$
  $\delta_{ant} = \delta_{novo}$
  $\delta_{aux} = r^T s$
  $M^{-1}, \Theta_{ant}, \nabla L_{ant}(\Theta_{ant}) = \text{MPC}(\Theta, \Theta_{ant}, \nabla L(\Theta), \nabla L_{ant}(\Theta_{ant}), t)$
  $s = M^{-1}r$
  $\delta_{novo} = r^T s$
  if *update = PolakRibiere [6]* **then**
    $\beta = \dfrac{\delta_{novo} - \delta_{aux}}{\delta_{ant}}$
  else
    $\beta = \dfrac{\delta_{novo}}{\delta_{ant}}$(atualização via *FletcherReaves* [5])
  if $\beta < 0$ **then**
    $\beta = 0$
  else
    $\beta = 1$
  $d = s + \beta d$

A grande vantagem desse algoritmo está em sua convergência em menos passos comparado com o *gradiente descendente*. Além disso, o *gradiente conjugado* procura o ponto mínimo através de uma reta, que em certos casos, ajuda na convergência.

Entretanto, pelo fato de que para cada movimento, o gradiente não retorna ao passo anterior (caso o valor atual seja pior comparado com o valor antigo) e pelo alto custo computacional, devido ao cálculo de matrizes inversas e de outras operações matemáticas custosas. O método do *gradiente conjugado* é inferior comparado ao *gradiente descente*.

1.5.3 *Checagem do gradiente* em redes neurais

Nesse método, utilizamos o método da *checagem do gradiente* para analisar o comportamento do gradiente e verificar se realmente está convergindo ao valor certo. A ideia é aproximar o gradiente $\dfrac{\partial J(\Theta)}{\partial \Theta}$ pela *secante*. Assim, temos que

$$\frac{\partial J(\Theta)}{\partial \Theta} \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

e com esse procedimento, é possível ainda vetorizar esse processo. Basta que transformas as matrizes de pesos $\Theta^{(l)}$ em vetores e assim obtemos $\theta = [\theta_1, \theta_2, \ldots, \theta_n]$. Dessa forma, o gradiente é aproximado pela seguinte equação

$$\frac{\partial J(\Theta)}{\partial \Theta_1} = \frac{J(\Theta_1 + \epsilon, \Theta_2, \ldots, \Theta_n) - J(\Theta_1 - \epsilon, \Theta_2, \ldots, \Theta_n)}{2\epsilon}$$
$$\frac{\partial J(\Theta)}{\partial \Theta_2} = \frac{J(\Theta_1, \Theta_2 + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \Theta_2 - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial J(\Theta)}{\partial \Theta_n} = \frac{J(\Theta_1, \Theta_2, \ldots, \Theta_n + \epsilon) - J(\Theta_1, \Theta_2, \ldots, \Theta_n - \epsilon)}{2\epsilon}.$$

Com isso, é possível fazer a atualização dos pesos sendo

$$\frac{\partial J(\Theta)}{\partial \Theta} = \begin{vmatrix} \frac{\partial J(\Theta)}{\partial \Theta_1} \\ \frac{\partial J(\Theta)}{\partial \Theta_2} \\ \vdots \\ \frac{\partial J(\Theta)}{\partial \Theta_n} \end{vmatrix}.$$

Por fim, a atualização dos pesos e *bias* serão da forma

$$\Theta = \Theta - \alpha \frac{\partial J(\Theta)}{\partial \Theta},$$

onde os valores de α e ϵ são escolhidos pelo usuário durante a implementação do código.

Com isso, é possível verificar se a implementação do *gradiente descendente* está correta e após verificar se este valor está próximo do obtido pelo *gradiente descendente*, essa checagem deve ser removida do código. É recomendado remover essa checagem pois esse algoritmo é extremamente lento e possui uma baixa taxa de convergência, servindo apenas como um ferramental de checagem do gradiente, como o próprio nome do método sugere.

2 Implementação das redes neurais e discussão dos resultados obtidos

Nesse projeto, foi implementado três redes neurais utilizando a linguagem de programação *python*. Onde cada modelo, respectivamente, utiliza os algoritmos do *gradiente descendente*, do *gradiente conjugado* e da *checagem do gradiente* para treinamento dos pesos e *bias*.

Por fim, foi verificado a acurácia de cada modelo na tarefa de classificação utilizado como *dataset* do *MNIST*. Esse conjunto de dados é composto por 5000 imagens de dimensão 20 x 20 em tom de preto e branco de números escritos à mão e que variam dos números 0 à 9. Além disso, são separados 500 dados para cada classe de número. Para maior facilidade para implementação, cada imagem foi transformada em um vetor de dimensão 1 x 400 e todas as 500 imagens de cada classe foram adicionada em um arquivo de extensão *.mat*(em um arquivo de matriz do *Matlab*) de 5000 x 400 e os valores estão salvo nos padrões do *Octave/MATLAB* ao invés do padrão de texto *ASCII*. Por fim, todos os códigos implementado nesse trabalho foram baseados na aula 9 de [2] e os bias de cada camada assume valor 1. Ou seja, é somado 1 antes de aplicar a função f. Com f sendo a função *Softmax* na última camada e f a sendo a função *sigmoid* nas demais camadas.

2.1 Inicialização dos pesos

Para todas as redes neurais, foi adotado a seguinte inicialização dos pesos

$$\alpha(l-1,l) = \sqrt{\frac{6}{n_{l-1}+n_l}}$$
$$\Theta^{(l-1)} = (B_\theta * 2 * \alpha(l-1,l)) - \alpha(l-1,l)$$

para todo $l \geq 1$. Com n_l, n_{l-1} sendo a quantidade de neurônios, respectivamente, das camada l e $l-1$ e B_θ é uma matriz de dimensão n_{l-1} x n_l e que cada elemento da $(b_\theta)_{ij}$ matriz B_θ estão entre 0 e 1. Ou seja, $(b_\theta)_{ij} \in [0,1]$.

2.2 Implementação de uma rede neural que utiliza a *checagem do gradiente*

Para esta parte, foi criado um código de nome *BackpropagationChecagemGradiente.py* Nele, foi implementada uma função que calcula a média da diferença entre o vetor gradiente e a aproximação por secantes e imprime esse valor a cada iteração do Gradiente Descendente. Dessa forma, foi montando uma rede como sugerida pelo problema, com 3 unidades de entrada, 5 na camada escondida e 3 na saída, e utilizando 3 exemplos de treinamento gerados aleatoriamente, o programa imprimiu o seguinte, ao realizar 5 iterações do Gradiente Descendente:

Diferença na aproximação: 0.13197527471135098
Diferença na aproximação: 0.054938988255387206

Diferença na aproximação: 0.017403946624392947
Diferença na aproximação: 0.013742906848079267
Diferença na aproximação: 0.009941982262297288

Analisando as aproximações, é possível perceber que ao longo das iterações do *Gradiente Descendente*, as aproximações por secantes se aproximam mais dos valores obtidos pelo cálculo do gradiente normal. Desse modo, pode-se afirmar que o cálculo dos gradientes do programa está correto.

2.3 Implementação de uma rede neural que utiliza o *gradiente descendente*

Utilizando a linguagem de programação *Python* e baseando-se nas notas de aula, fizemos a implementação do algoritmo. Dividindo sempre cada pedaço do código em funções como um método para facilitar as chamadas destas no nosso algoritmo. Nosso código na primeira versão já funcionava para n exemplos de treinamento e isso foi usado para testar outros métodos que foram pedidos. O funcionamento do algoritmo também ocorre normalmente para qualquer número de classes, sendo esse também um recurso interessante para reduzir o número de exemplos de treinamento, e trabalhar com grupos menores do conjunto *MNIST*. Uma das maiores dificuldades que encontramos foi a implementação de novas camadas. Para resolver isso, no início foi preciso utilizar muitos recursos de depuração para encontrar os problemas de dimensionamento que ocorreram dentro da nossa função *ComputeCost*. Como tática, começamos implementando camadas internas manualmente, e fazendo vários testes de execuções para elas, depois pegando a relação de recorrência que ocorre nas camadas mais internas começamos a construir os *loops* para automatizar o sistema de criação de camadas.

Depois da especificação do número de neurônios internos, número de classes e quantidade de elementos de entrada, que na realidade já são a quantidade de parâmetros da nossa imagem, fazemos uma concatenação nas dimensões especificadas anteriormente, com os elementos gerados aleatoriamente das nossas matrizes de *thetas*, então chamamos a função *gradientDescent*, nela pegamos o vetor coluna da nossa concatenação e repartimos novamente para cada *theta* do nosso código, aqui é um ótimo momento para uma pausa para uma explicação, no inicio podemos definir o *numLayers*, mas na realidade o número de camadas do nosso código é definido pela quantidade de *thetas*, ativadores e *delta*. Por fim, nossas derivadas, então o número de camadas é consequência de todos os fatores que estamos aumentando no nosso código. Dentro do nosso *gradientDescent* vai ocorrer a nossa atualização dos *thetas* e também vai chamar a nossa *computeCost*, dentro dele nós novamente precisamos colocar os valores dentro de nossas matrizes, assim que fazemos isso temos nossas matrizes de *thetas*, logo depois calculamos nossos ativadores e e fazemos o cálculo da nossa função de custo, assim podemos ir para o cálculo do nossos *deltas* e logo depois definimos nossos *Grads*, assim voltamos para o método *gradientDescent* e atualizamos os *thetas*, quando terminamos o número de iterações necessárias voltamos para a

nossa *main* e vamos para o método *prediction*. Esse é um resumo do nosso código, claro dentro dele existem outras funções que devem ser executadas para os cálculos dos ativadores e deltas como *sigmoid* e a *sigmoidGradient*.

Algum dos resultados obtidos foram os seguintes :

Alguns resultados obtidos para 1 camada interna, introdução manual:
$inputLayerSize = 400$
$hiddenLayerSize = 20$
$numLabels = 10$
$alpha = 0.8$
$iteraçõe = 800$
$Training\ Set\ Accuracy = 93.89999999999999\%$
$- - -514.4781627655029seconds - -$

Alguns resultados obtidos para 2 camadas internas, introdução manual:
$inputLayerSize = 400$
$hiddenLayerSize = 20$
$numLabels = 10$
$alpha = 0.8$
$iteraçõe = 1500$
$Training\ Set\ Accuracy = 76.34\%$
$- - -1418.296094417572seconds|$

Alguns resultados obtidos para 2 camadas internas, introdução manual:
$inputLayerSize = 400$
$hiddenLayerSize = 25$
$numLabels = 10$
$alpha = 0.8$
$iteraçõe = 1600$
$Training\ Set\ Accuracy = 86.88\%$
$- - -5323.1008000060431seconds|$(para esse teste a máquina ficou em *standby* por um longo tempo, o tempo deve ser desconsiderado).

Alguns resultados obtidos para 2 camadas internas, introdução manual:
$inputLayerSize = 400$
$hiddenLayerSize = 25$
$numLabels = 10$
$alpha = 0.8$
$iteraçõe = 1500$
$Training\ Set\ Accuracy = 67.56\%$
$- - -1495.8373906612396seconds|$

Alguns resultados obtidos para 3 camadas internas, introdução manual:
$inputLayerSize = 400$
$hiddenLayerSize = 20$
$numLabels = 10$
$alpha = 3.0$

$$iteraçõe = 2000$$
$$Training\ Set\ Accuracy = 96.8\%$$
$$- - -2311.5979058742523seconds - -$$

Para a implementação para número de camadas internas maiores do que2,nesse caso, começamos a fazer os testes para 3 camadas internas. Mas os resultados que vamos mencionar aqui valem para todas as camadas internas. O modelo do código continua o mesmo, uma pergunta que deve ser respondida é *porque apenas introduzimos para duas camadas internas, ao olharmos o modelo das camadas internas na estrutura do código?*É possível perceber que existe uma relação de recorrência interna, tanto nos ativadores quanto nos *deltas*. Então, só precisamos encaixar de maneira correta a quantidade de elementos de entrada e saída pertencentes aos *thetas* e começar a seguir um modelo estrutural interno entre a segunda camada interna. Esse método obteve êxito, como mostrado na sua implementação manual. Depois de algumas simulações, foi verificado que o método estava funcionando, no qual mostrava depois de algumas iterações mostrava uma convergência muito lenta. Como a precisão era muito baixa, basicamente todos os resultados eram classificados como um único valor. Em alguns testes, foi obtido alguns resultados com 20% de acurácia. A seguir, é possível verificar alguns resultados para o código de n camadas escondidas de forma interativa ao invés da manual

Alguns resultados obtidos para 3 camadas internas, introdução iterativa:
$$inputLayerSize = 400$$
$$hiddenLayerSize = 20$$
$$numLabels = 10$$
$$alpha = 0.8$$
$$iteraçõe = 1600$$
$$Training\ Set\ Accuracy = 10.0\%$$
$$- - -1725.501962184906seconds|$$

Alguns resultados obtidos para 3 camadas internas, introdução iterativa:
$$inputLayerSize = 400$$
$$hiddenLayerSize = 20$$
$$numLabels = 10$$
$$alpha = 0.8$$
$$iteraçõe = 1500$$
$$Training\ Set\ Accuracy = 10.0\%$$
$$- - -1525.960108757019seconds - -$$

Após algumas tentativas de varredura de código e de substituição direta dos valores que eram operados dentro do *loop*. Foi aumentado o número de neurônios como tentativa de convergência.

Alguns resultados obtidos para 3 camadas internas, introdução iterativa:
$$inputLayerSize = 400$$
$$hiddenLayerSize = 50$$

$$numLabels = 10$$
$$alpha = 0.5$$
$$iteraçõe = 2000$$
$$Training\ Set\ Accuracy = 10.0\%$$
$$- - -1525.960108757019seconds - -$$

Logo depois começamos a verificar as saídas e velocidade de diminuição do *JHistory*.

Na implementação desse código, foi definido como código manual o algoritmo que foi implementado manualmente com número de camadas e código iterativo,em que foi construímos as relações de recorrência para $n > 2$ camadas internas. Assim, podemos ver uma aproximação muito grande do código que convergiu para aquele que não convergiu. Mesmo assim, depois de algumas iterações chegamos em um mínimo local, com convergência lenta. A seguir, é possível visualizar alguns dos resultados dos valores da função de perda.

Para $\alpha = 3.0$ e 20 neurônios, 3 camadas internas
Valor implementado com *looping* : [6.938773404768356, 7.2630379456884775, 5.480711481875639, 3.960403266070936, 3.270428953944134, 3.2636640893877393, 3.2619156729346765, 3.26122771112502, 3.2608936026713304, 3.2608175679514697, 3.2607392833337294, 3.2607280018925398, 3.260699091182187, 3.2606905349336364, 3.2606729514606143, 3.2606623518927584, 3.260647980998898, 3.260636170866475, 3.26062288026128, 3.2606105616981176, 3.260597679834129, 3.2605851774233456, 3.260572468581173, 3.2605599146733732]
Valor implementado manualmente: [6.898121845497133, 7.166009882098611, 3.521947508296392, 3.2851678270862377, 3.2680530061311255, 3.259481011540431, 3.257564960713688, 3.2568936271547058, 3.256699074026682, 3.2566310817942985, 3.256604553039253, 3.2565905131520547, 3.256580561047487, 3.2565719496561973, 3.2565638177872462, 3.2565558680206625, 3.2565480022874866, 3.256540182493972, 3.256532395572568, 3.256524635363577]

Além disso, foi feito o teste para mais neurônios. Para esse resultado em específico, obtivemos uma melhora na acurácia em comparação aos resultados anteriores para o código de uma camada. No caso, a melhora foi de 1% para um pequeno grupo de iterações.

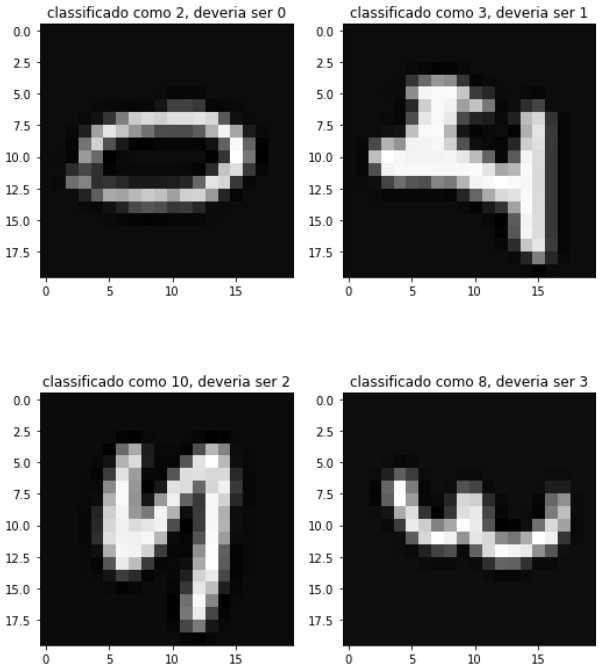
Para $alpha = 0.5$ e 40 neurônios, 3 camadas internas

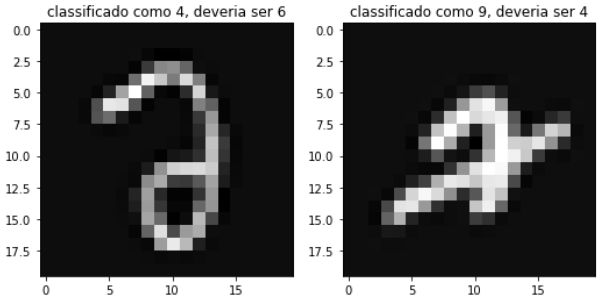
Valor implementado com *looping* :
[7.007578970731205, 3.251410950611318, 3.2513163353111416, 3.251292022880333, 3.2512857690321044, 3.2512841510341013, 3.2512837222998385, 3.251283598516133, 3.2512835529065303, 3.2512835273343668, 3.25128350689903, 3.2512834877823136, 3.2512834690061494, 3.2512834503201464, 3.2512834316602564,

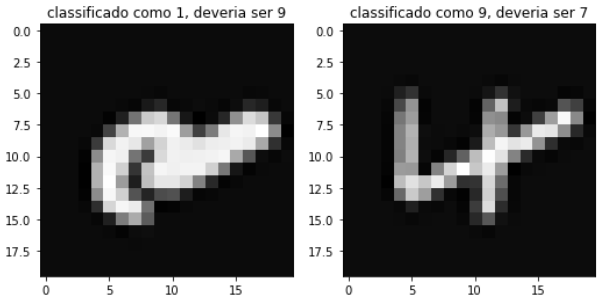
3.251283413010147, 3.2512833943656663, 3.2512833757257713,
3.25128335709019, 3.251283338458866, 3.2512833198317943,
3.2512833012089524, 3.251283282590352, 3.251283263975985]
Valor implementado manualmente: [7.018239577184086, 3.251485403553457,
3.251334499353232, 3.2512993365336644, 3.251290869605539,
3.251288761697265, 3.251288209224541, 3.2512880444440033,
3.25128797748891, 3.251287935408547, 3.251287899685545,
3.251287865595418, 3.251287831929134, 3.251287798376781,
3.251287764858837, 3.251287731354881, 3.2512876978596745,
3.2512876643718767, 3.2512876308911243, 3.2512875974173343,
3.251287563950492, 3.25128753049057, 3.2512874970375885,
3.2512874635915257, 3.251287430152386, 3.2512873967201728,
3.2512873632948747, 3.251287329876504, 3.251287296465046,
3.2512872630605036, 3.251287229662875]

Assim, é notável que existe realmente um ponto de mínimo local em que, logo após um número de iterações, nossos dois métodos acabam se estabilizando. O problema que encontramos de não convergência do método de n camadas, com $n > 2$ e de forma automática, pode ser decorrente a erros numéricos consequentes dos números de *loops* que tivemos que fazer com vários tensores. Além disso, podemos atribuir ao número muito pequeno de dados para o treinamento da nossa rede neural em comparação ao número muito grande de camadas internas.

Por fim, foi plotado alguns classificações que um dos modelos realizou.







Onde todas essas fotos são do seguinte modelo

Para 2 camadas internas e modelo feito manualmente :
$inputLayerSize = 400$
$hiddenLayerSize = 20$
$numLabels = 10$
$alpha = 0.8$
$iteraçõe = 1500$
$Training\ Set\ Accuracy = 76.34\%$
$---1418.296094417572seconds---$

2.4 Implementação de uma rede neural que utiliza a *checagem do gradiente*

Primeiramente, foi criado um programa para uma camada interna na rede neural com nome *1camadainternaGradienteConjugado.* Neste caso, foi variado o número de neurônios na camada interna, os valores de λ, o número de iterações e a tolerância mínima de aproximação do gradiente. O melhor resultado que forneceu a maior precisão, mantendo o tempo de execução em no máximo 10 minutos foi o seguinte: 5 neurônios internos; $\lambda = 1$; número máximo de iterações = 100 e tolerância de aproximação do gradiente = $10-3$. Devido à demora na execução do Gradiente Conjugado, tivemos que abaixar o número de exemplos de treinamento para 10 exemplos de treinamento, com um exemplo de cada

dígito. Com esses parâmetros, obtivemos os seguintes resultados com a execução do programa:

Optimization terminated successfully.
Current function value: 3.202833
Iterations: 45
Function evaluations: 204534
Gradient evaluations: 99
Training Set Accuracy: 80.0%
$---300.4320583343506 seconds---$

A acurácia do Gradiente Conjugado, como mostrada para 10 exemplos de treinamento, foi de 80%. É uma ótima acurácia, porém a execução para 5000 exemplos mostrou-se demasiadamente demorada.

Além disso, foi implementamos o algoritmo do *Gradiente Conjugado* no programa de n camadas, utilizando para isso 10 exemplos de treinamento no programa *backpropagationGradienteConjugado.py.*. Com isso, foi testado várias combinações de números de neurônios em cada camada escondida, λ, número de iterações e tolerância de aproximação do gradiente, mantendo 3 camadas escondidas. Porém o programa sempre retorna uma acurácia de 10% e classifica os dígitos como se todos fossem somente um dígito. Por exemplo, para 5 neurônios nas camadas internas , $\lambda = 1$, número de iterações = 100 e tolerância de aproximação de 10^{-3} para o gradiente, obtivemos o seguinte *output*:

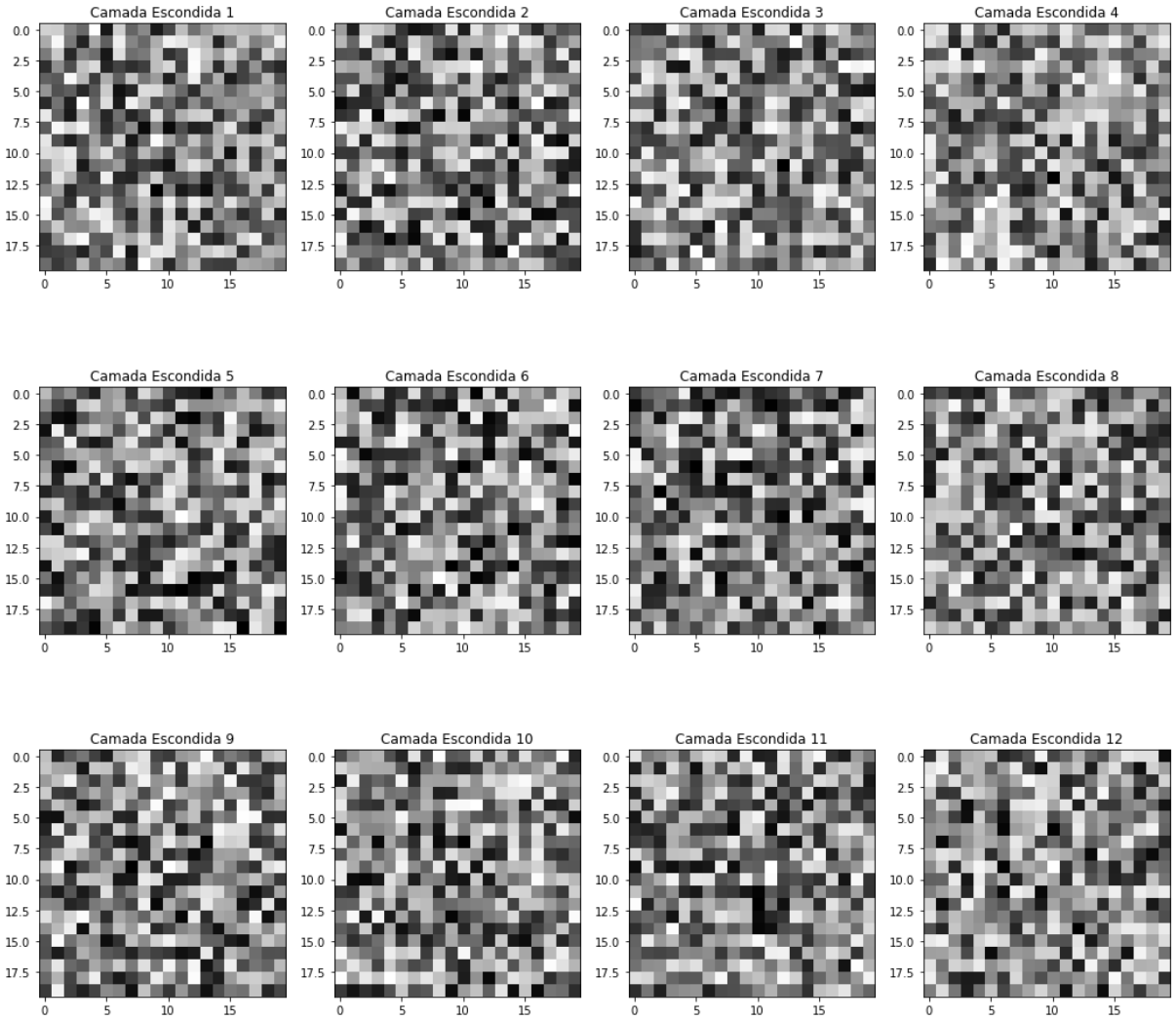
Optimization terminated successfully.
Current function value: 3.251081
Iterations: 8
Function evaluations: 42520
Gradient evaluations: 20
Training Set Accuracy: 10.0%
—115.74599957466125 seconds—

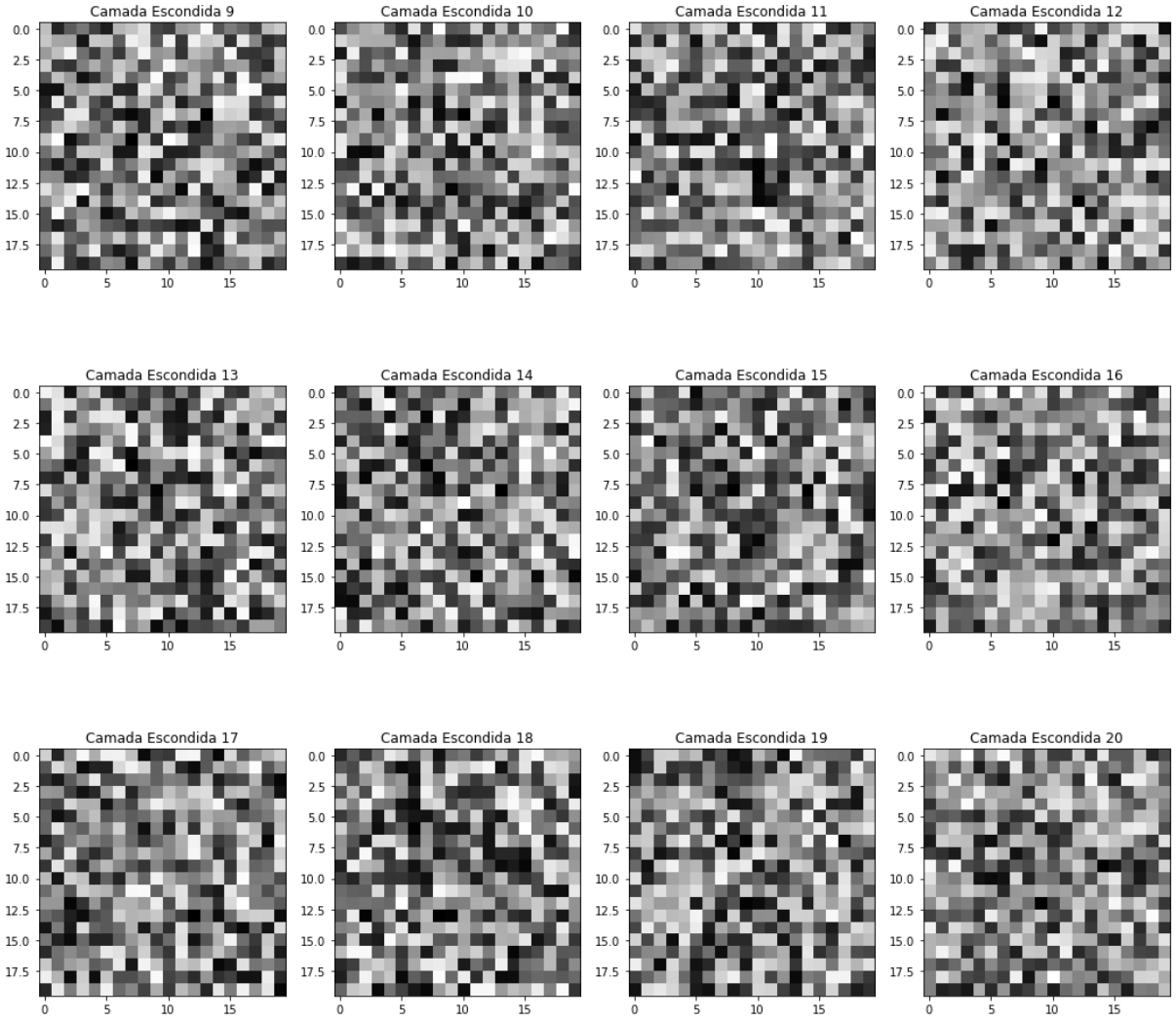
Acreditamos que 3 camadas internas não funcionam para o problema com 10 exemplos de treinamento. Com isso, tentamos utilizar mais exemplos de treinamento, como 30 exemplos, mas não surtiu efeito, pois obtivemos apenas mais tempo de execução para a mesma classificação errada que aproxima todos os exemplos somente com o dígito 1. Além disso, foi aumentado os números de exemplos de treinamento aumentaria exponencialmente o tempo de execução do programa

2.4.1 Imagem para cada unidade escondida

Para essa parte, foi criado o programa *backpropagationPlotImagens*, que é baseado no algoritmo para n camadas escondidas previamente implementado nesse trabalho. Neste novo programa, foi implementado uma função denominada *showImage(theta1)* que plota a imagem de cada camada escondida em função de

theta1. Para o código com 500 iterações do *gradiente descendente*, obtivemos as seguintes imagens para as três primeiras camadas escondidas:





2.5 Divisão de erro, validação e teste

Para esse problema, criamos o programa *v34TreinoValidacaoTeste.py*, que é baseado no código de duas camadas internas. No qual divide os exemplos do *dataset MNIST* em teste, validação e treino de acordo com as porcentagens dadas pelo programador. Entretanto, não tivemos tempo suficiente para implementar a função em nosso algoritmo para n camadas internas.

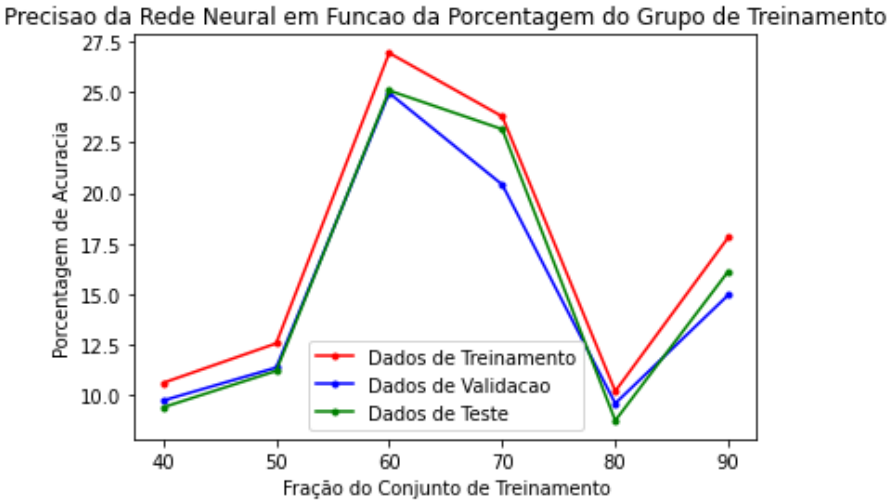
Após o *gradiente descendente* ser aplicado nos exemplos de treinamento com 20 neurônios por camada interna e 800 iterações do *gradiente descendente*. Foi feito o teste com os exemplos de validação e teste e obtivemos os seguintes resultados:

Training Set Accuracy: 26.934673366834172%
Validation Set Accuracy: 24.95049504950495%
Testing Set Accuracy: 25.07462686567164%
$---373.1127655506134 seconds---$

Podemos observar que as acurácias para cada grupo ficaram bem parecidas. O que mostra uma consistência no treinamento e que nosso modelo não está errado.

2.6 Testes com Frações diferentes

Nesse exercício, utilizamos o programa anterior e variamos os fracionamentos entre conjuntos de treinamento, validação e teste em frações 40%/30%/30%, 50%/25%/25%, 60%/20%/20%, 70%/15%/15%, 80%/10%/10% e 90%/5%/5%. Mantendo o número de iterações em 800 e o número de neurônios das camadas internas em 20, obtivemos acurácias com as quais geramos o seguinte gráfico no programa *grafico.py*:



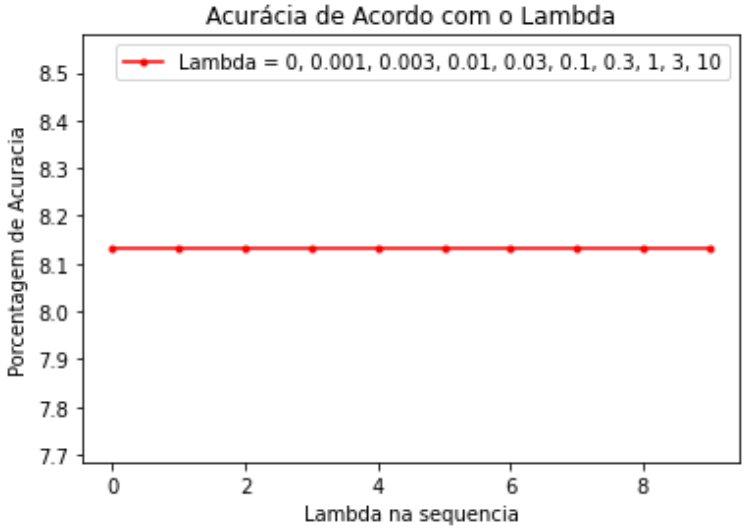
Ao Analisar o gráfico, vemos que houve uma acurácia maior para o fracionamento dos dados de treinamento perto da faixa de 60% a 70%. A baixa acurácia se deve provavelmente à baixa quantidade de dados de treinamento. Quanto à baixa acurácia com mais exemplos de treinamento, é provável que o número de iterações do *Gradiente Descendente* tenha sido muito baixa em relação ao número de exemplos de treinamento e/ou o número de neurônios nas camadas internas seja muito baixo. Há a possibilidade também de o *Gradiente Descendente* tentar aproximar todos os dados ao mesmo tempo de tal modo que acaba não tendo uma acurácia suficiente.

2.7 Testes com Lambdas Diferentes

Nessa parte, a implementação foi baseada nos algoritmos de rede neural com 2 camadas internas da letra *A*. Assim, foi feito um *loop* que testa os valores sugeridos de $\lambda = 0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10$. Esses valores foram escolhidos como uma recomendação do professor para serem utilizados nesse relatório. O programa calcula o custo J utilizando o λ e verifica a acurácia do conjunto de validação para cada *theta* obtido. Por fim, o modelo seleciona o λ que deu a maior acurácia de todas, calculando em seguida a acurácia do conjunto de testes para o *theta* obtido por este λ.

Para 2 camadas internas, 100 iterações máximas e 20 neurônios em cada camada interna, obtivemos os seguintes resultados:

Validation Set Accuracy for $\lambda = 0.0$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 0.001$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 0.003$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 0.01$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 0.03$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 0.1$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 0.3$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 1.0$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 3.0$: [[8.13165537]]%
Validation Set Accuracy for $\lambda = 10.0$: [[8.13165537]]%
Best Lambda: $\lambda = 0.0$
Testing Set Accuracy: 9.649122807017543%
$---459.84608459472656 seconds --$



Como não tivemos tempo para rodar com um número maior de iterações

e neurônios na rede, todos os λ, obtiveram a mesma acurácia para o conjunto de validação, de 8.13%, implicando em uma acurácia de 9.65% no conjunto de teste. O λ selecionado foi $\lambda = 0$ e o gráfico permite observar os valores de acurácias de acordo com os Lambdas. O correto seria rodar para 25 neurônios internos e 1600 iterações, mas isto levaria muito tempo, provavelmente algumas horas. Entretanto, o programa funciona bem e realiza a tarefa.

3 conclusão

Pelo testes, foi verificado que o melhor resultado foi para as redes neurais treinadas utilizando o método do *gradiente descendente*. Devido a maior precisão e melhor custo computacional comparado com os outros modelos. Além disso, é possível utilizar um maior número de exemplo de treinamento sem afetar tanto o custo computacional. O único problema encontrado nesse relatório para as redes neurais treinadas com o *gradiente descendente* está pelo fato que a implementação iterativa do método não funcionou corretamente. No qual apresentou uma taxa de acerto em torno de 10%. Porém, para os códigos das redes neurais implementada manualmente com *gradiente descendente*, foi obtido um grande exito nas predições. Com a melhor predição de treinamento em torno de 97%.

References

[1] Saurabh Adya, Vinay Palakkode, and Oncel Tuzel. Nonlinear conjugate gradients for scaling synchronous distributed DNN training. *CoRR*, abs/1812.02886, 2018.

[2] Joao B. Florindo. Ms960 : Tópicos especiais em processamento de imagens. `https://www.youtube.com/playlist?list=PLGwGFVrptiyRmFoDWxruNGgTu2cSPnTLX`, 2020.

[3] Aurelien Geron. Hands-on machine learning with scikit-learn and tensorflow: Concepts, tools, and techniques to build intelligent systems, 2017.

[4] Jonathan Hui. Conjugate gradient. `https://jonathan-hui.medium.com/rl-conjugate-gradient-5a644459137a`.

[5] Barak Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6, 02 1970.

[6] E. Polak and G. Ribiere. Note sur la convergence de méthodes de directions conjuguées. *ESAIM: Mathematical Modelling and Numerical Analysis - Modélisation Mathématique et Analyse Numérique*, 3(R1):35–43, 1969.

[7] Naveen Venkatesan. Using logistic regression to create a binary and multiclass classifier from basics. `https://towardsdatascience.com/using-logistic-regression-to-create-a-binary-and-multiclass-classifier-from-basics-26f5e`