P2-MS960

Bryan Alves do Prado - 195171
Hugo Ricardo Ribeiro Matarozzi - 155743
Isabella Mi Hyun Kim - 170093

Outubro 2021

Conteúdo

1 Redes Neurais **3**
- 1.1 Panorama Geral . . . 3
- 1.2 História . . . 4
- 1.3 Forward-Propagation e Back-Propagation . . . 6
- 1.4 Aplicações . . . 7

2 Parte I: Redes Neurais **8**
- 2.1 Introdução . . . 8
- 2.2 A Questão da Escolha do Número de Camadas Escondidas e do Número de Neurônios . . . 8
- 2.3 Análise do Problema 1: . . . 10
  - 2.3.1 Testes com Alta Taxa de Aprendizado . . . 11
  - 2.3.2 Testes com Baixa Taxa de Aprendizado . . . 12
  - 2.3.3 Visualização . . . 13
- 2.4 O Algoritmo Desenvolvido Para a Rede Neural . . . 14
- 2.5 O Algoritmo da Checagem de Gradiente . . . 17
- 2.6 O Método do Gradiente Conjugado . . . 19
- 2.7 Análise Acerca dos Resultados Obtidos Com o Gradiente Conjugado . . . 20
- 2.8 O Algoritmo do Gradiente Conjugado . . . 24
- 2.9 Visualização de Imagens Para Cada Unidade Escondida . . . 26
- 2.10 Anexos da Parte I . . . 28

3 Parte II: Validação e Seleção de Modelos **32**
- 3.1 Introdução . . . 32
- 3.2 Importância de Avaliar um Modelo . . . 32
- 3.3 A Divisão em Treino, Teste e Validação . . . 33
- 3.4 O Algoritmo Para a Divisão dos Grupos . . . 34
- 3.5 Curvas de Aprendizado . . . 35
- 3.6 Escolha Automatizada do λ . . . 36
- 3.7 Algoritmo Para a Automatização da Escolha de λ . . . 37
- 3.8 Anexos da Parte II . . . 38

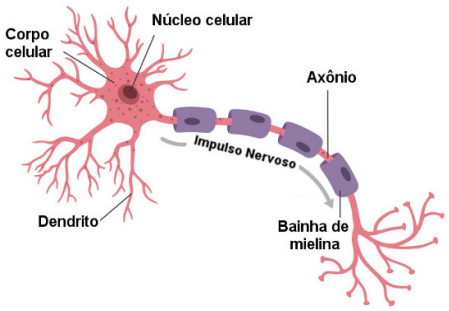
4 Conclusão **40**

5 Bibliografia: **41**

1 Redes Neurais

1.1 Panorama Geral

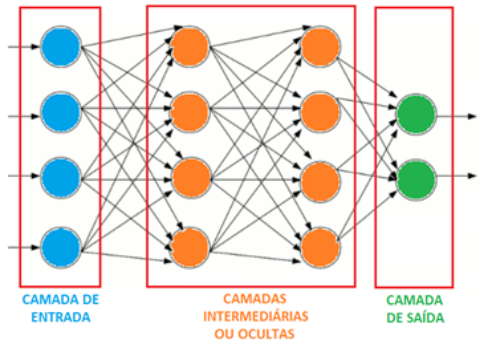
Podemos definir o conceito de redes neurais como técnicas computacionais, aplicadas principalmente no âmbito de aprendizado de máquinas, baseadas em modelos matemáticos fortemente inspirados na estrutura e modo de atuação do sistema nervoso de organismos inteligentes que adquirem conhecimento através da experiência. Tal sistema é formado por neurônios, células complexas com papel extremamente importante na definição do comportamento humano, seu aprendizado, entre outros. A seguir, um exemplo de neurônio:



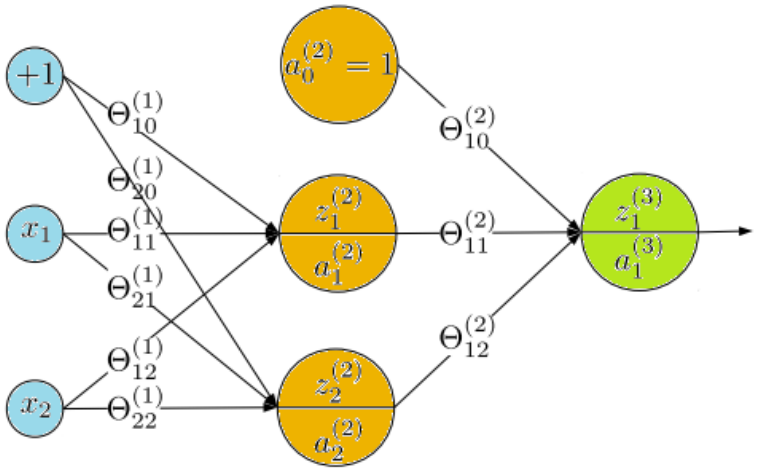
Quando tratamos de tal modelo no âmbito matemático, estamos basicamente falando de uma divisão em três grupos de camadas que atuam de forma específica:

- camada de entrada, onde os dados são oferecidos à rede;
- camadas intermediárias (mais conhecidas como camadas escondidas/ocultas), responsáveis pelo processamento e extração de características dos dados;
- camada de saída, responsável por computar e apresentar o resultado final.

A seguir, temos um exemplo de como tal modelo se organiza:



Conseguimos observar que possuímos ligações entre as camadas da rede neural: estas são conhecidas como pesos e indicam um valor de influência na saída da unidade. De forma simplificada, consideremos uma camada n e uma camada $n+1$: o peso incidiria sobre a saída da camada n, e tal saída atuaria como uma entrada da camada $n+1$. Consideremos então que em cada camada da rede neural teremos um número i de neurônios (que pode ser variável), e em cada neurônio teremos uma soma ponderada z composta pelos pesos anteriormente citados multiplicados por seus respectivos valores de entrada. Tal valor z será utilizado em uma função de ativação a, de forma que para uma função sigmoide, por exemplo, teríamos algo como $a = g(z) = \frac{1}{1+e^{-z}}$. A seguir, um exemplo de como fica uma rede neural nestas considerações:

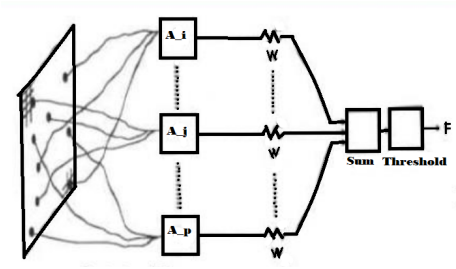


1.2 História

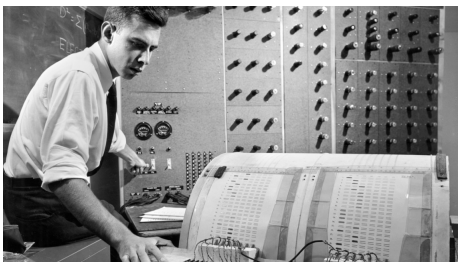
Quando adentramos na história do desenvolvimento e concepção de redes neurais, acabamos por observar alguns acontecimentos que indicam grandes rupturas e evoluções em relação a conceitos, técnicas e tecnologias já estabelecidas.

Começando na década de 1940, temos Warren McCulloch e Walter Pits criando o primeiro modelo computacional para redes neurais baseado em matemática e lógica; em seguida, na década de 1950, Frank Rosenblatt cria o algoritmo Perceptron, com a função de reconhecimento de padrões baseada em uma rede neural simples de duas camadas.

Tal algoritmo foi publicado no paper "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain" de 1958. Esses dois pontos são de extrema importância e servem como pontos de ignição para um amplo estudo e desenvolvimento posterior da área. A seguir, uma representação de como era o algoritmo Perceptron original e uma foto de Frank Rosenblatt:

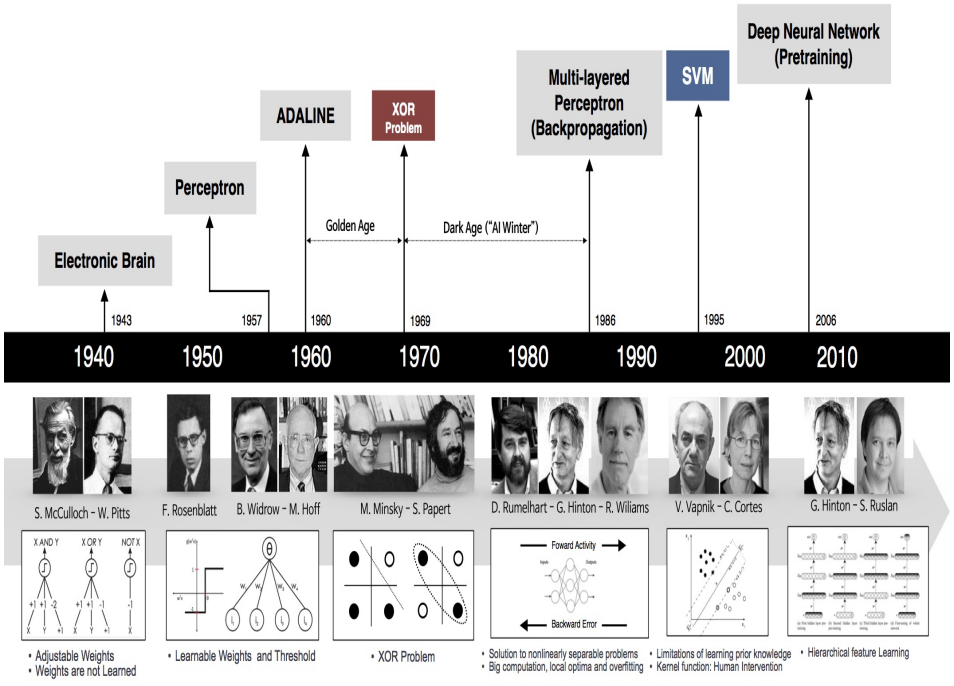


(a) Perceptron



(b) Frank Rosenblatt

Em seguida, na década de 1980, temos Kunihiko Fukushima propondo a Neoconitron, uma rede neural de hierarquia multicamadas utilizada no reconhecimento de caligrafia e padrões. Posteriormente, na mesma década, temos a criação de redes neurais profundas, porém com tempo de execução da ordem de dias, inviabilizando o projeto. Com o passar do tempo, a tecnologia progrediu de forma extremamente acentuada assim como os modelos e técnicas de redes neurais, levando a um ponto de grande ruptura em 2012 quando algoritmos de reconhecimento artificiais conseguiram alcançar uma taxa de desempenho comparável à humana. A partir desse ponto, temos um proliferação imensa de aplicações de redes neurais, principalmente se utilizando de Deep Learning. A seguir, um diagrama com os principais marcos da história do desenvolvimento de redes neurais:



1.3 Forward-Propagation e Back-Propagation

Algoritmos extremamente importantes para as redes neurais artificiais, e seu desenvolvimento, o Forward-Propagation é responsável por percorrer a rede neural desde sua camada de entrada até a camada de saída (como o próprio nome sugere), por meio do uso do valor calculado na função de ativação como valor de entrada para a próxima camada, a partir da primeira camada escondida (no caso da primeira camada, é utilizada a própria matriz X). Seu funcionamento pode ser exemplificado, para uma rede de quatro camadas (2 ocultas):

Algorithm 1 Forward-Propagation

1: $a^{(1)} = X$
2: $z^{(2)} = \theta^{(1)} a^{(1)}$
3: $a^{(2)} = g(z^{(2)}) + a^{(2)}_{(0)}$
4: $z^{(3)} = \theta^{(2)} a^{(2)}$
5: $a^{(3)} = g(z^{(3)}) + a^{(3)}_{(0)}$
6: $z^{(4)} = \theta^{(3)} a^{(3)}$
7: $a^{(4)} = h_\theta(x) = g(z^{(4)})$

O Back-Propagation, por sua vez, é um algoritmo que percorre a rede neural pelo caminho reverso, ou seja, da camada de saída até a camada de entrada, melhorando progressivamente os pesos utilizados nas camadas da rede durante o percurso, através do cálculo e uso do gradiente. De forma geral, o algoritmo se comporta da seguinte forma:

Algorithm 2 Back-Propagation

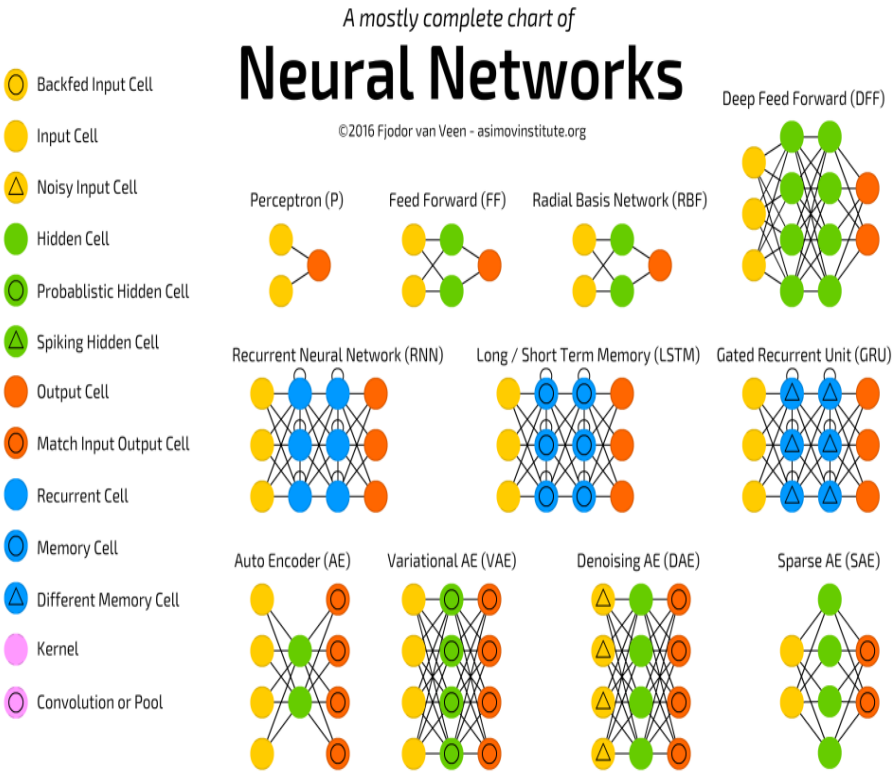
1: $\Delta^l_{ij} := 0$
2: **for** $i = 1 : m$ **do**
3: $\quad a^{(1)} := X^{(i)}$
4: $\quad \delta^{(L)} := a^{(L)} - Y^{(i)}$
5: $\quad \delta^{(l)} = (\theta^{(l)})^T \delta^{(l+1)} * a^{(l)} * (1 - a^{(l)})$, para $l = (L-1) : 2$
6: $\quad \Delta^l_{ij} := \Delta^l_{ij} + A^l_J \delta^{l+1}_i$
7: **end for**
8: $D^l_{ij} := \frac{1}{m}\Delta^l_{ij} + \lambda\theta^l_{ij}$, se $j \neq 0$
9: $D^l_{ij} := \frac{1}{m}\Delta^l_{ij}$, se $j = 0$
10: $\frac{\partial}{\partial \theta^l_{ij}} J(\theta) = D^l_{ij}$

1.4 Aplicações

Dadas suas amplas variedades de modelos, técnicas e abordagens, as redes neurais são amplamente aplicadas em problemas de áreas e setores diversos da sociedade, citando por exemplo a utilização na área médica para a identificação de doenças como câncer, nas redes sociais para a identificação de indivíduos em fotos, no mercado financeiro para a emissão ou não de crédito, além de mineração de dados, reconhecimento de caracteres e imagens, entre outros.

Com a ascensão da inteligência artifical e o amplo movimento de automatização de inúmeros setores da sociedade, as redes neurais passam a se estabeler fortemente e, de certa forma, passam a moldar parte de nosso futuro, gerando debates acerca de seus "limites" e amplo impacto na sociedade. Um tema abordado de forma interessante nos livros "Sapiens: Uma Breve História da Humanidade" e "Homo Deus" do professor e historiador israelense Yuval Noah Harari é a questão de empregos, postos de trabalho que poderão sumir e novos que irão surgir (e necessitarão de uma qualificação considerável de seus ocupantes, situação que pode se tornar grave em países subdesenvolvidos).

A seguir, alguns modelos de redes neurais e suas formas de organização. Sua diversidade nos permite visualizar bem sua ampla versatilidade e capacidade de aplicação em inúmeros âmbitos:



2 Parte I: Redes Neurais

2.1 Introdução

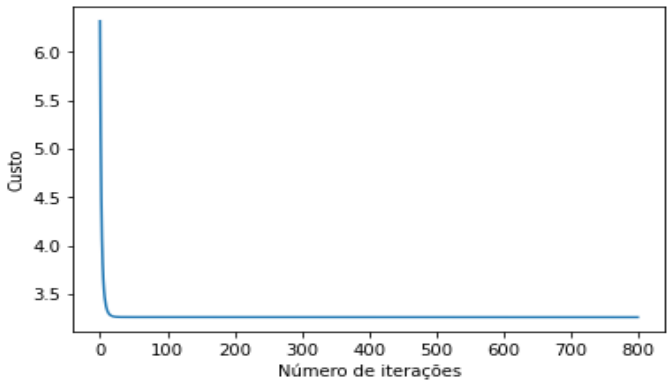
O projeto trata sobre a aplicação de redes neurais no tópico de identificação de dígitos. Para tal, desenvolvemos um modelo que permite ao usuário selecionar o número de camadas, número de "neurônios" nas camadas escondidas e outros valores como a taxa de aprendizado e o valor de λ. Assim, temos como resultado o número total de dígitos classificados corretamente, a porcentagem de eficácia do modelo com os parâmetros oferecidos pelo usuário e a visualização dos dígitos que foram erroneamente classificados.

2.2 A Questão da Escolha do Número de Camadas Escondidas e do Número de Neurônios

Em um primeiro momento, tendemos a imaginar que a escolha de um maior número de camadas escondidas (Hidden Layers) tende a ser benéfica para um modelo de rede neural, porém isso não é verdade. Quando estamos tratando de algoritmos que utilizam métodos de aprendizado baseados em gradientes, podemos presenciar um fenômeno conhecido como "Vanishing Gradient": tal fenômeno tem como causa a adição de muitas camadas na rede neural sem um tratamento e objetivo adequado, e pode ser causado principalmente pela escolha da função de ativação de forma errônea.

A função sigmoide gera derivadas muito pequenas, e quando executamos o Backpropagation, essas derivadas diminuem exponencialmente enquanto propagamos para as camadas iniciais, o que pode fazer com que os pesos sejam muito pouco alterados com o passar das iterações, diminuindo bruscamente a taxa de aprendizado do modelo.

Pudemos observar esse problema no nosso modelo desenvolvido ao executar um teste para uma rede de 6 camadas (4 camadas escondidas), com parâmetros de taxa de aprendizado 0.1, $\lambda = 1$, número de neurônios nas camadas escondidas igual a 25 e número de iterações igual a 800. com esse teste, obtivemos a seguinte curva de custo por iteração:

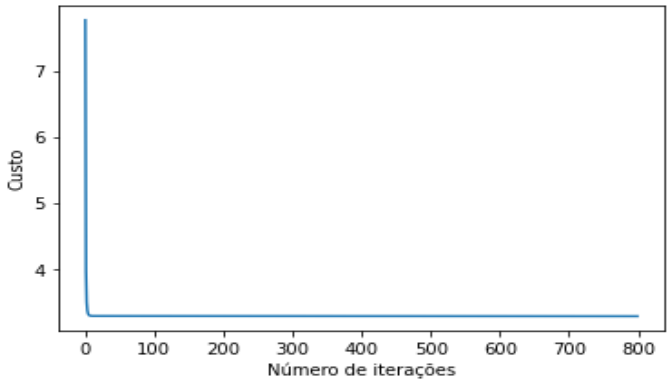


Como podemos observar, o custo reduziu bruscamente nas primeiras iterações, porém, em dado momento (por volta da iteração 21) sua taxa de redução diminuiu muito, tendendo a estagnação. Seguem os valores de custos obtidos nas 20 últimas iterações, onde é possível observar a taxa de redução muito baixa:

780	float64	1	3.2642624485929743
781	float64	1	3.264261018709759
782	float64	1	3.264259588936802
783	float64	1	3.264258159273641
784	float64	1	3.2642567297198117
785	float64	1	3.2642553002748502
786	float64	1	3.2642538709382922
787	float64	1	3.2642524417096777
788	float64	1	3.2642510125885345
789	float64	1	3.264249583574405
790	float64	1	3.2642481546668316
791	float64	1	3.2642467258653403
792	float64	1	3.264245297169472
793	float64	1	3.2642438685787627
794	float64	1	3.2642424400927514
795	float64	1	3.2642410117109777
796	float64	1	3.264239583432966
797	float64	1	3.2642381552582673
798	float64	1	3.264236727186413
799	float64	1	3.264235299216934

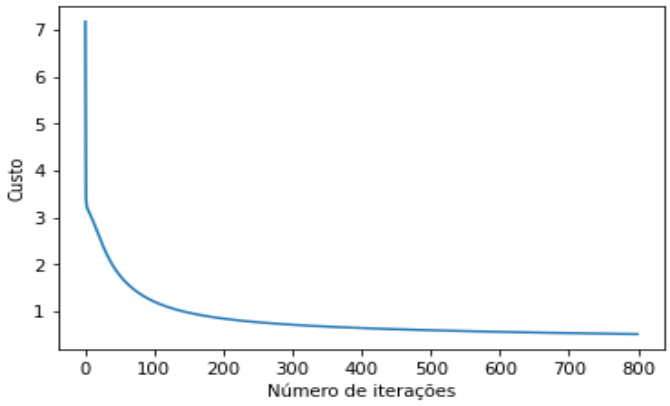
O resultado dessa baixa eficácia no treinamento do modelo ficou bem visível nos resultados obtidos nas previsões, onde foram classificados corretamente apenas 787 dígitos, o que equivale a 15.74% do total. Como observação, a utilização de outras funções de ativação, como a ReLU, pode amenizar e até solucionar o problema.

É interessante visualizarmos também como a quantidade de neurônios nas camadas escondidas pode impactar o treinamento do modelo. De forma geral, quanto maior for o número de neurônios melhor será o treinamento do modelo a cada iteração, porém o custo computacional tende a crescer e, com isso, o modelo tende a ficar mais lento. De forma geral, uma análise de forma a obter um equílibrio entre velocidade e qualidade do treinamento do modelo é útil, adequada e deve ser feita caso a caso. Para o caso das 6 camadas já retratado, com os mesmos parâmetros e apenas o número de neurônios aumentado para 100 por camada escondida, obtivemos 2111 dígitos classificados corretamente, o que representa 42.23% do total. A curva de custo porém também tendeu à estagnação em um dado momento, como podemos ver a seguir:



2.3 Análise do Problema 1:

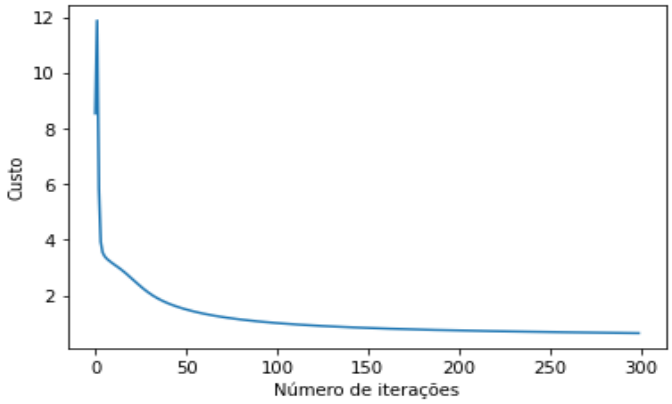
Para este problema, desenvolvemos um modelo de rede neural regularizada que permite ao usuário a escolha dos parâmetros gerais, como taxa de aprendizado e λ, mas também permite ao usuário a escolha de características estruturais da rede neural, como o seu número de camadas e a quantidade de neurônios em cada camada escondida. Utilizando os valores propostos pelo problema, ou seja, uma camada escondida com 25 neurônios, e utilizando parâmetros como a taxa de aprendizado igual a 0.8, $\lambda = 1$ e número de iterações igual a 800, obtivemos a seguinte curva de custos:



Como podemos observar, o custo se reduziu a um patamar muito baixo, o que se traduziu na alta eficácia do modelo na identificação de dígitos: foram classificados corretamente 4722 dígitos, o que equivale a 94.46% do total.

Também realizamos alguns testes para esse problema com diferentes parâmetros, como por exemplo uma taxa de aprendizado igual a 1, $\lambda = 1$, número de iterações igual a 300, somente uma camada escondida com número de neurônios igual a 100. Obtivemos 4623 dígitos corretamente classificados, equivalente a

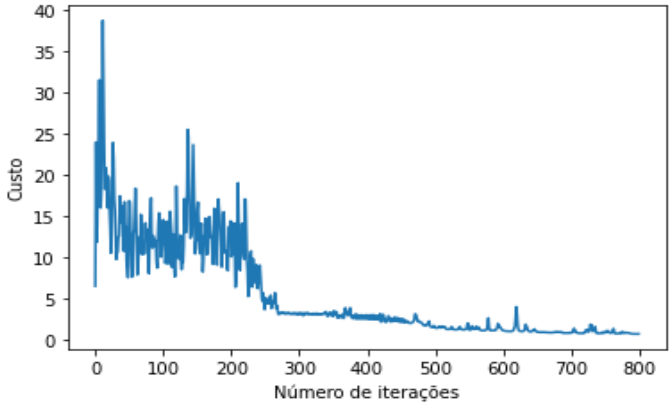
92.48% de acerto, com a seguinte curva de custo:



É possível reparar a existência de um pequeno repique em uma das primeiras iterações, que possívelmente ocorreu por conta do alto número de neurônios na camada escondida. Com os mesmos valores gerais e apenas o número de neurônios igual a 60 na camada escondida, obtivemos uma taxa de acerto de 95.38%, o que equivale a 4768 dígitos corretamente classificados.

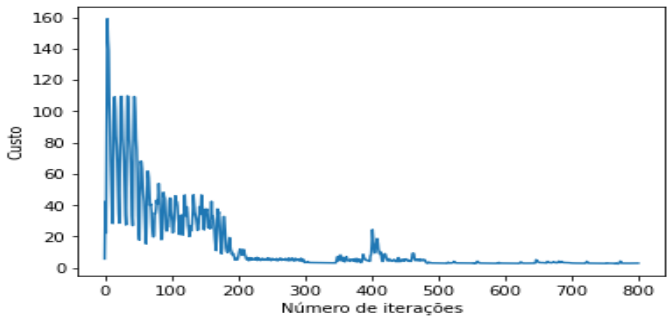
2.3.1 Testes com Alta Taxa de Aprendizado

Fizemos também um teste com um alto valor de taxa de aprendizado, no caso, 10, uma camada escondida com 25 neurônios, $\lambda = 1$ e 800 iterações. O resultado nos permitiu a visualização de um fenômeno interessante, que é a oscilação da curva de custo de maneira brusca e contínua:



O resultado final foi de 91.68% da classificações corretas, porém é vísivel que se trata de um modelo instável e que pode gerar problemas em aplicações fu-

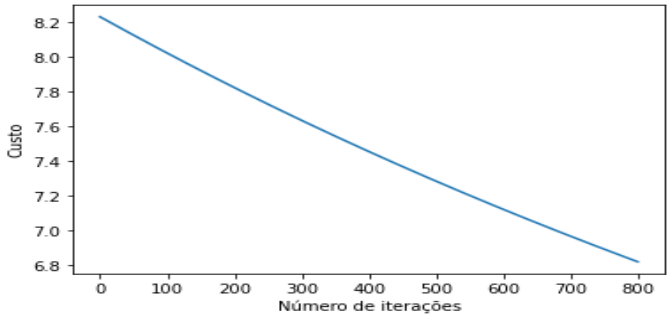
turas. Usando os mesmos parâmetros e alterando apenas a taxa de aprendizado para 20, obtivemos a seguinte curva de custo:



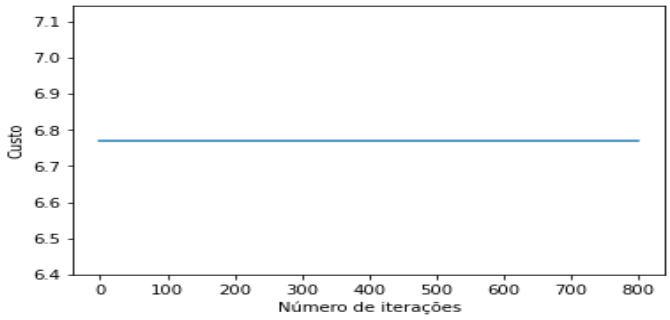
Apesar de ter um formato semelhante à curva anteriormente obtida, ela possuí picos e frequência de oscilação muito maiores, o que acabou por fazer com que sua taxa de acerto fosse baixa, no caso, 18.82%

2.3.2 Testes com Baixa Taxa de Aprendizado

Testes com uma taxa de aprendizado baixa, 0.0001, também foram realizados, e utiliziando os mesmos parâmetros dos casos anteriores, i.e., uma camada escondida, 25 neurônios, 800 iterações e $\lambda = 1$, o resultado foi de 9.98% de classificações corretas e a seguinte curva de custo:



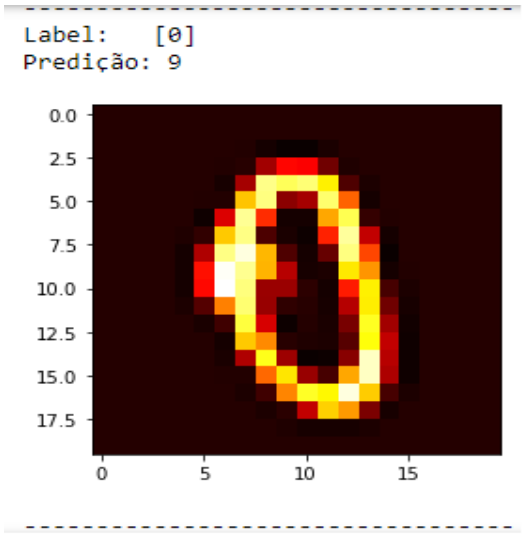
Por fim, com os mesmos parâmetros anteriores e apenas a taxa de aprendizado diminuída para 0.00000000000000000001, tivemos como resultado 12.44% de classificações corretas e obtivemos a seguinte curva de custo:



É importante ressaltarmos que para as situações extremas, como taxa de aprendizado muito alta ou muito baixa, número de iterações baixo, entre outros, os θ iniciais que são inicializados randomicamente podem ter uma influência considerável.

2.3.3 Visualização

A visualização das imagens erroneamente classificadas é uma escolha que o modelo oferece ao usuário, sendo apresentadas da seguinte forma:



Dessa forma, concluímos que o modelo satisfaz de forma adequada o problema e que, com a utilização dos parâmetros corretos, o resultado obtido é satisfatório e o tempo de execução é razoável. Trata-se assim de uma boa implementação no âmbito da identificação de dígitos.

No anexo se encontram mais aplicações do modelo, principalmente com diferentes números de camadas, valores de λ e quantidade de neurônios nas camadas escondidas.

2.4 O Algoritmo Desenvolvido Para a Rede Neural

Nosso foco para o desenvolvimento do algoritmo sempre foi torná-lo o mais geral possível, permitindo ao usuário a escolha dos parâmetros, e também torná-lo, na medida do possível, um algoritmo rápido e eficiente. Com isso, vetorizamos tudo que foi possível e testamos trecho a trecho do código para verificarmos se sua execução era correta. Assim, obtivemos o seguinte modelo:

```
import copy as cp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat

def teste():
    testex = mat["X"]
    m = testex.shape[0]
    n = testex.shape[1]
    nbr_classes = 10
    testex = np.append(np.ones([m,1]),testex,axis=1)
    testey=mat["y"]
    for x in range(testey.shape[0]):
        if testey[x]==10:
            testey[x]=0
    aux = []
    for i in range(m):
        aux.append(np.array([1 if testey[i] == j else 0 for j in range(nbr_classes)]))
    testey = np.array(aux).reshape(-1, nbr_classes)
    return testex, testey, m, n, nbr_classes

def Load():
    X = data_2.values
    m = X.shape[0]
    n = X.shape[1]
    nbr_classes = 10
    X = np.append(np.ones([m,1]),X,axis=1)
    y = data_3.replace(10,0).values
    aux = []
    for i in range(m):
        aux.append(np.array([1 if y[i] == j else 0 for j in range(nbr_classes)]))
    Y = np.array(aux).reshape(-1, nbr_classes)
    return X, Y, m, n, nbr_classes

def sigmoid(z):
    return 1/(1+np.exp(-z))

def sigmoidGradient(z):
    R=1/(1+np.exp(-z))
    return R*(1-R)

def CostFunction(nbr_classes,k,A,Y,m,thetas):
    J=0
    for h in range(nbr_classes):
        J = J + np.sum(-Y[:,h]*np.log(A[-1][:,h])-(1-Y[:,h])*np.log(1-A[-1][:,h]))
    J=(1/m)*J
    for g in thetas:
        J=J+(k/(2*m))*(np.sum(g[:,1:]**2))
    return J


def ForwardPropagation(X,b,thetas):
    A=[]
    aux=X
    m=X.shape[0]
    for i in range(0,b-2):
        a=sigmoid(aux @ thetas[i].T)
        a=np.hstack((np.ones((m,1)), a))
        A.append(a)
        aux=a
    A.append(sigmoid(A[-1] @ thetas[-1].T))
    return A

def BackPropagation(X,Y,A,thetas):
    grads=[]
    erro=[]
    F=A[-1]-Y
    grad=F.T @ A[-2]
    grads.append(grad)
    erro.append(F)
    error=F
    for i in range(len(thetas)-2,0,-1):
        error=np.multiply((error @ thetas[i+1][:,1:]),(sigmoidGradient(A[i-1] @ thetas[i].T)))
        grad=error.T @ A[i-1]
        erro.append(error)
        grads.append(grad)
    error=np.multiply((error @ thetas[1][:,1:]),(sigmoidGradient(X @ thetas[0].T)))
    grad= error.T @ X
    erro.append(error)
    grads.append(grad)
    return list(reversed(grads)),list(reversed(erro))

def GradientDescent(X,Y,m,thetas,alpha,p,k,n,nbr_classes):
    J=[]
    for i in range(p):
        e,grads,A,erro=Compute_Cost(X,Y,m,thetas,n,b,nbr_classes,k)
        for j in range(0,len(thetas)):
            thetas[j]=thetas[j]-alpha*grads[j]
        J.append(e)
    return thetas, J, A, grads, erro

def Compute_Cost(X,Y,m,thetas,n,b,nbr_classes,k):
    A=ForwardPropagation(X,b,thetas)
    J=CostFunction(nbr_classes,k,A,Y,m,thetas)
    grads,erro=BackPropagation(X,Y,A,thetas)
    for r in range(0,len(grads)):
        grads[r]=(1/m)*grads[r]
        grads[r]=grads[r]+(k/m)*np.hstack((np.zeros((thetas[r].shape[0],1)),thetas[r][:,1:]))
    return J,grads,A,erro

def Random_Weight(a,b):
    i=(6**(1/2))/((a+b)**(1/2))
    return np.random.rand(b,a+1)*(2*i)-i

def Theta_Initialization(b,a,X,Y):
    f=[]
    f.append(Random_Weight(X.shape[1]-1,a[0]))
    for i in range(1,b-2):
        f.append(Random_Weight(a[i-1],a[i]))
    f.append(Random_Weight(a[-1],Y.shape[1]))
    return f

def Prediction(Y,X,b,m,thetas):
    A=ForwardPropagation(X,b,thetas)
    j=np.argmax(A[-1],axis=1)
    np.array(j).reshape((m, 1))
    r=np.argmax(Y,axis=1)
    u=0
    for k in range(m):
      if j[k]==r[k]:
            u=u+1
    return j,r,u

def missFeedback(X, Y, pred):
    m = len(Y)
    y_labels = []
    for row in Y:
        y_labels.append(np.argmax(row))
    y_labels = np.array(y_labels).reshape((m, 1))
    for i in range(m):
        if not (y_labels[i] == pred[i]):
            print(f'Label: \t {y_labels[i]}')
            print(f'Predição: {pred[i]}')
            pixels = X[i, 1:].reshape((20, 20))
            plt.imshow(pixels, cmap='hot')
            plt.show()
```

E nosso "hub" de execução ficou da seguinte forma:

```
mat = loadmat(r'C:\Users\USUARIO\Desktop\Neural Network\ex3data1.mat')
testex,testey,testem,testen,teste_nbr_classes=teste()
data_2=pd.read_csv(r'C:\Users\USUARIO\Desktop\Neural Network\imageMNIST.csv')
data_3=pd.read_csv(r'C:\Users\USUARIO\Desktop\Neural Network\labelMNIST.csv')
X,Y,m,n,nbr_classes=Load()
b=int(input("Qual seria o número de camadas? "))
a = list(map(int,input("\nQual seria o número de neurônios em cada camada escondida? ").strip().split()))[:n]
thetas=Theta_Initialization(b,a,X,Y)
alpha=int(input("\nQual seria o número de iterações? "))
apren=float(input("\nQual seria a taxa de aprendizado? "))
lambd=float(input("\nQual seria o valor de lambda? "))
aux_theta,aux_grads,grads_aux=GradientChecking(lambd,b)
thetas,J,A,grads,erro=GradientDescent(X,Y,m,thetas,apren,alpha,lambd,n,nbr_classes)
pred,real,total=Prediction(Y,X,b,m,thetas)
print("\nO modelo obteve um total de ",total," classificações corretas" )
print("\nRepresentando uma taxa de eficácia de: ","{:.2f}".format((total/m)*100),"%")
l=int(input("\nGostaria de ver a curva de custo por iteração? (1 para sim, 2 para não) "))
q=int(input("\nGostaria de ver os dígitos que foram classificados incorretamente? (1 para sim, 2 para não) "))
if q==1:
    missFeedback(X,Y,pred)
if l==1:
    plt.plot(range(len(J)),J)
    plt.xlabel('Número de iterações')
    plt.ylabel('Custo')
    plt.show()
```

Primeiramente, é possível observar no "hub" de execução uma chamada de função que não foi retratada acima (checagem de gradiente), ela será tratada no tópico seguinte. Seguindo, começamos com a inicialização dos dados e a preparação dos mesmos (matriz X e Y) através da função "$Load$", para serem utilizados na rede neural.

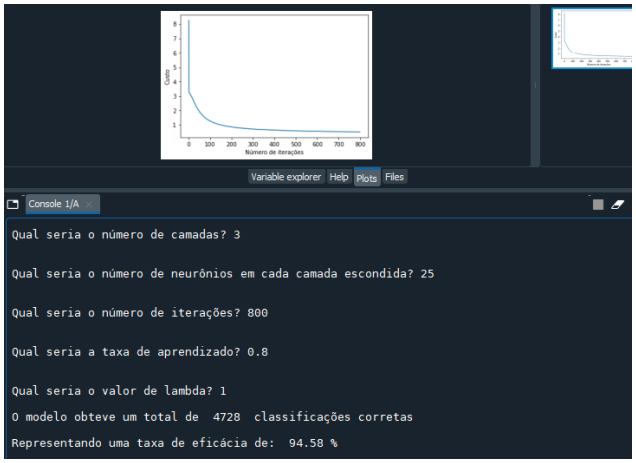
Com isso, obtemos do usuário o número de camadas e a quantidade de neurônios em cada camada escondida. É importante frisar que estamos lidando com o fato de que ao oferecer um número de camadas, isso se trata do número total. Em outras palavras ou seja, se o usuário der como entrada três camadas, se trata de uma camada de entrada, uma camada escondida e uma camada de saída.

Em seguida, começamos a preparação dos pesos através da função "$Theta\_Inicialization$", que com o auxílio de "$Random\_Weight$ inicializa os θ (pesos) de forma aleatória e com as dimensões adequadas para cada trecho da rede neural.

Com as entradas oferecidas pelo usuário no trecho seguinte, chamamos a função "$GradientDescent$", responsável por calcular os melhores pesos possíveis em um dado número de iterações. Tal função chama a cada iteração a função "$Compute\_Cost$", responsável por calcular o valor do custo J e os gradientes a cada iteração, isso com o auxílio da função "$ForwardPropagation$", que retorna a lista A com as matrizes dos valores de ativação; a função "$CostFunction$", que com a utilização de A, retorna o custo J e, por fim, a função "$BackPropagation$", responsável por calcular os gradientes de forma retroativa, ou seja, caminhando na rede neural de trás para frente.

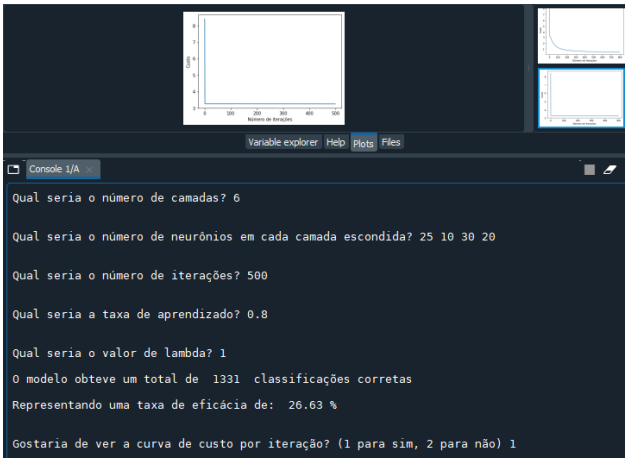
Com isso, ao finalizarmos a execução de "$GradientDescent$", teremos os melhores pesos possíveis obtidos no dado número de iterações, e assim, podemos através da função "$Prediction$" averiguar qual a quantidade e porcentagem de acertos da rede neural com os pesos obtidos. Em seguida, o usuário pode optar

por visualizar ou não as imagens que foram erroneamente classificadas através da função "$MissFeedback$". A seguir, um exemplo de como é executado o programa na IDE "Spyder":



Por fim, temos de citar também algumas funções "assistentes" que foram utilizadas, como por exemplo "$sigmoid$", responsável por atuar como função de ativação quando executamos o "$ForwardPropagation$", e "$sigmoidGradient$", que atua no cálculo do "$BackPropagation$".

Temos também a função "$teste$", utilizada para inicializarmos os dados ofertados no exemplo dado pelo professor em aula, e assim conseguimos verificar também com eles se nosso modelo gera uma boa taxa de predição. A seguir, um exemplo de como o programa funciona para um caso de 6 camadas:



Após averiguarmos que se trata da mesma base de dados do projeto, a função teste foi excluída.

2.5 O Algoritmo da Checagem de Gradiente

O método de Backpropagation é complexo e, se mal implementado, pode gerar "bugs", ou seja, pode fazer com que os gradientes calculados se distanciem consideravalmente dos gradientes reais. Para tal, desenvolvemos um algoritmo que calcula numericamente os gradientes utilizando a lógica de $\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$, sendo J a função que calcula o custo em um dado θ e ϵ é uma pequena perturbação, na ordem de 10^{-4}, que utilizamos para aproximar o gradiente numericamente.

Como se trata de um algoritmo que tem um custo computacional demasiado alto, inserir o mesmo no ciclo de iterações que utilizamos na rede "padrão" tornaria o processo todo extremamente lento. Com isso, desenvolvemos uma "sub rede neural" composta por uma matriz Y já no formato ideal para uma rede neural com dimensões 5×3, uma matriz X com dimensões 5×4 (A coluna com m linhas de 1 já está inclusa), um número de camadas escondidas e λ iguais aos ofertados pelo usuário e, por fim, um total de cinco neurônios por camada escondida. Assim, utilizando um caso de uma camada escondida e $\lambda = 1$, obtivemos o seguinte resultado:

	0	1	2	3	4	5
0	0.520695	0.450686	0.461371	0.602695	0.591656	0.545043
1	0.516916	0.558915	0.457565	0.4996	0.540924	0.469879
2	0.580687	0.556959	0.525433	0.528983	0.505288	0.526569

	0	1	2	3	4	5
0	0.520695	0.450028	0.460541	0.630992	0.637048	0.571684
1	0.516916	0.610329	0.495546	0.532828	0.568104	0.501906
2	0.580687	0.548094	0.517969	0.526244	0.504699	0.523772

	0	1	2	3
0	0.0815177	0.165546	0.192609	0.210548
1	0.0643166	0.139087	0.0836289	0.216447
2	0.110354	0.205878	0.151388	0.0444157
3	0.159944	0.0836175	0.250425	0.05856
4	0.126906	0.143425	0.193087	0.129808

	0	1	2	3
0	0.0737606	0.202188	0.214171	0.25065
1	0.0623071	0.165269	0.0907977	0.256457
2	0.110242	0.236688	0.176088	0.0471143
3	0.156568	0.0877953	0.288861	0.061598
4	0.126377	0.159827	0.222036	0.147009

O primeiro resultado se trata de uma comparação do gradiente obtido pelo Backpropagation (matriz superior), com o gradiente obtido pela checagem numérica, (matriz inferior) em relação ao peso referente ao "trecho" entre a camada escondida e a camada de saída. Já o segundo resultado se trata também de uma

comparação do gradiente obtido pelo Backpropagation (matriz superior) com o gradiente obtido pela checagem numérica (matriz inferior), desta vez em relação ao peso referente ao "trecho" entre a camada de entrada e a camada escondida. É visível que os resultados são bastante semelhantes, o que indica a ausência de "bugs" ou "má implementações" no Backpropagation, e muito provavelmente as diferenças sutis são geradas por erro de cálculos propagados.

A seguir, nosso código para a checagem de gradiente:

```
def GradientChecking(lambd,b):
    aux_theta=[]
    input_layer_size=3
    hidden_layer_size=5
    num_labels=3
    m=5
    aux_theta.append(AssistWeight(hidden_layer_size,input_layer_size))
    for i in range(1,b-2):
        aux_theta.append(AssistWeight(hidden_layer_size, hidden_layer_size))
    aux_theta.append(AssistWeight(num_labels, hidden_layer_size))
    aux_X=np.append(np.ones((m,1)),AssistWeight(m,input_layer_size-1), axis=1)
    n=aux_X.shape[1]-1
    aux_Y=np.matrix('1 0 0; 1 0 0; 0 1 0; 0 1 0; 0 0 1')
    aux_J,aux_grads,aux_A,aux_erro=Compute_Cost(aux_X,aux_Y,m,aux_theta,n,b,num_labels,lambd)
    epsilon=1e-4
    grads_aux=[]
    for p in range(0,len(aux_theta)):
        grads_aux.append(np.zeros(aux_theta[p].shape))
    cont=0
    for x in aux_theta:
        for y in range(0,x.shape[0]):
            for z in range(0,x.shape[1]):
                assist=cp.deepcopy(aux_theta)
                minnum=cp.deepcopy(x)
                minnum[y,z]=minnum[y,z]-epsilon
                maxnum=cp.deepcopy(x)
                maxnum[y,z]=maxnum[y,z]+epsilon
                assist[cont]=cp.deepcopy(minnum)
                costmin=Compute_Cost(aux_X, aux_Y, m, assist, n, b, num_labels, lambd)[0]
                minnum[y,z]=minnum[y,z]+epsilon
                assist[cont]=cp.deepcopy(maxnum)
                costmax=Compute_Cost(aux_X, aux_Y, m, assist, n, b, num_labels, lambd)[0]
                maxnum[y,z]=maxnum[y,z]-epsilon
                grads_aux[cont][y,z]=(costmax-costmin)/(2*epsilon)
                assist[cont]=x
        cont=cont+1
    for u in range(0,len(grads_aux)):
        grads_aux[u]=(1/m)*grads_aux[u]
        grads_aux[u]=grads_aux[u]+(lambd/m)*np.hstack((np.zeros((aux_theta[u].shape[0],1)),aux_theta[u][:,1:]))
    return aux_theta,aux_grads, grads_aux
```

E abaixo, a função que utilizamos para a inicialização aleatória das matrizes de peso θ e de X:

```
def AssistWeight(e,s):
    W=np.zeros((s,e))
    W=np.random.rand(W.shape[1],W.shape[0]+1)
    return W
```

Como é fácil de se observar, primeiro inicializamos os parâmetros iniciais da sub-rede com auxílio da função "$AssistWeight$", que inicializa aleatóriamente nossos pesos e a matriz X. Em seguida, fazemos todos os processos normais comuns à rede neural, obtendo ao final de tais processos os gradientes calculados pelo Backpropagation e, com isso, executamos nosso modelo para a checagem de gradiente.

Diferente do que se costuma observar nas implementações comumente encontradas na internet, onde se utiliza uma lógica de estrutura de vetor, nós utilizamos uma lógica de estrutura de matrizes e, com isso, nosso código percorre cada matriz de peso em sua totalidade: linhas e colunas, somando e subtraindo a cada iteração a perturbação ϵ. A partir desse processo, preenchemos os gradientes numericamente, resultando ao final em uma lista de matrizes de gradientes correspondentes a cada peso θ.

2.6 O Método do Gradiente Conjugado

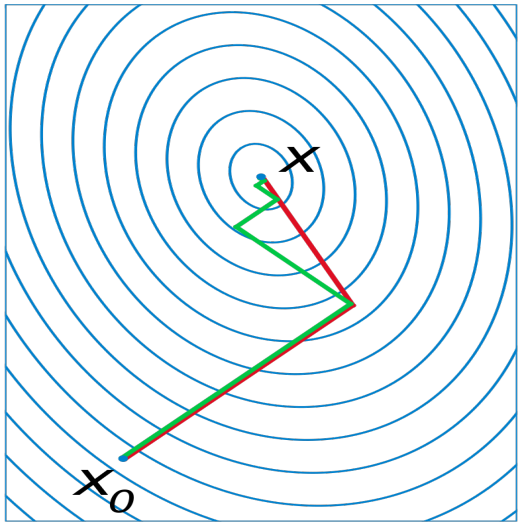
O método do gradiente conjugado se trata de um algoritmo baseado em uma solução numérica para um sistema de equações, definido por uma matriz positiva definida. De forma iterativa, sua lógica se baseia em encontrar um passo ótimo a cada iteração, gerando assim uma convergência em um menor número de passos.

Para um sistema da forma $Ax = b$ com A sendo uma matriz positiva definida, teremos:

Algorithm 3 Gradiente Conjugado

1: $x^{(1)} = aprox.inicial$
2: $d^{(1)} = r^{(1)}$
3: $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
4: $\alpha_k = -\frac{r^{(k)} \cdot d^{(k)}}{d^{(k)} \cdot Ad^{(k)}}$
5: $d^{(k+1)} = -r^{(k+1)} + \beta_k d^{(k)}$
6: $\beta^{(k)} = \frac{r^{(k+1)} \cdot Ad^{(k)}}{d^{(k)} \cdot Ad^{(k)}}$
7: $k = 1, 2, 3...$ *e* $r^{(k)} = Ax^{(k)} - b$

A seguir, a comparação de caminho de convergência entre o método de descida com taxa de aprendizado ótima (em verde), e o método de gradiente conjugado:



Como é possível observar neste exemplo, em um geral, o método de gradiente conjugado tende a convergir em um menor número de passos em relação ao método do gradiente de descida. É perceptível, porém, que seu custo computacional é consideravelmente mais elevado, dada a quantidade e tipo de cálculos executados, como vamos tratar no tópico em que abordaremos sua implementação.

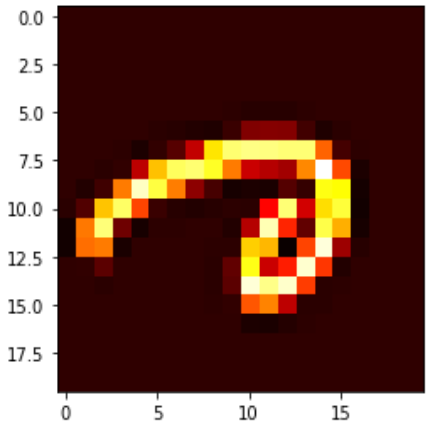
2.7 Análise Acerca dos Resultados Obtidos Com o Gradiente Conjugado

Conseguimos implementar com êxito o método do gradiente conjugado utilizando a biblioteca *scipy.optimize*. Nos testes iniciais, porém, onde executamos o algoritmo utilizando X e Y completos, nos deparamos com uma situação onde o código rodava "eternamente" (por um tempo demasiado alto completamente inviável para o modelo proposto), e assim constatamos o que já imaginávamos: o algoritmo do gradiente conjugado possui um custo computacional e, por consequência, um tempo de execução, extremamente altos.

Assim, decidimos restringir o número de exemplos para teste em um total de dez, um para cada valor de dígito (0-9). Para um caso de três camadas, com 10 neurônios na camada escondida, 100 iterações máximas, $\lambda = 1$ e gtol fixado em 10^{-4} nas opções da minimize, tivemos nove classificações corretas, representando uma taxa de eficácia de 90%, e o seguinte resultado retornado pela função *minimize*:

```
Optimization terminated successfully.
         Current function value: 3.142260
         Iterations: 83
         Function evaluations: 692328
         Gradient evaluations: 168
     fun: 3.1422601781544626
     jac: array([ 7.96616077e-05,  4.76837158e-07,  1.78813934e-07, ...,
        2.38120556e-05, -2.37226486e-05, -5.06639481e-06])
 message: 'Optimization terminated successfully.'
    nfev: 692328
     nit: 83
    njev: 168
  status: 0
 success: True
       x: array([ 1.78212949e+00,  4.97012523e-06,  1.97789462e-06, ...,
        1.14393018e-01, -1.16592851e-01, -2.17760755e-01])
```

Para este caso de teste, o erro ocorreu no dígito seis, que foi classificado pelo modelo como um nove:



Para um teste com três camadas, 20 neurônios na camada escondida, 200 iterações máximas, $\lambda = 1$ e gtol fixado em 10^{-4}, tivemos também nove classificações corretas, o que representa uma taxa de eficácia de 90%, e o seguinte resultado retornado pela função *minimize*:

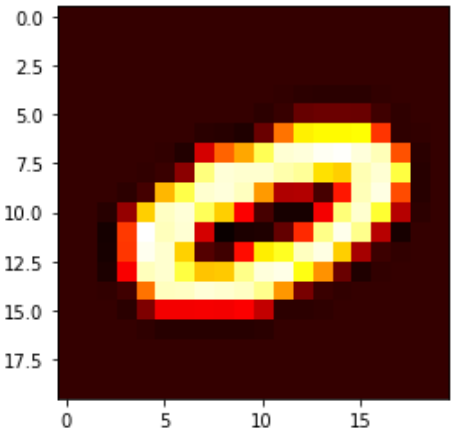
```
Optimization terminated successfully.
         Current function value: 3.108606
         Iterations: 125
         Function evaluations: 2049519
         Gradient evaluations: 249
     fun: 3.1086059741450014
     jac: array([-5.74588776e-05,  4.47034836e-07, -5.36441803e-07, ...,
        2.56597996e-05,  1.97291374e-05,  6.64591789e-06])
 message: 'Optimization terminated successfully.'
    nfev: 2049519
     nit: 125
    njev: 249
  status: 0
 success: True
       x: array([-4.23773106e-01,  4.27470604e-06, -5.42630019e-06, ...,
       -1.36797639e-01, -1.37839926e-01, -8.18447569e-02])
```

e novamente o erro ocorreu no dígito seis, que foi classificado pelo modelo como um nove.

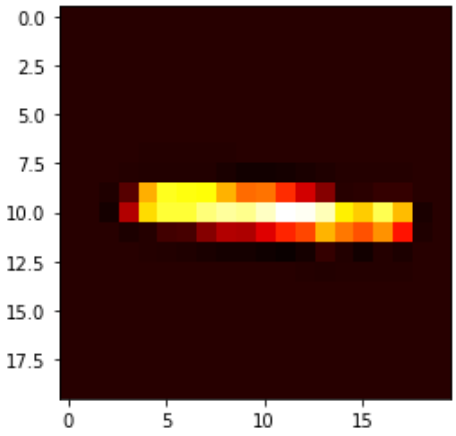
Para um teste com quatro camadas, com 10 neurônios em cada camada escondia, 200 iterações máximas, $\lambda = 1$ e gtol fixado em 10^{-4}, obtivemos apenas uma classificação correta, o que representa uma taxa de eficácia de 10%, e o seguinte resultado retornado pela função *minimize*:

```
Optimization terminated successfully.
         Current function value: 3.250830
         Iterations: 18
         Function evaluations: 169240
         Gradient evaluations: 40
     fun: 3.2508298735348733
     jac: array([-5.96046448e-08, -8.94069672e-08,  1.19209290e-07, ...,
       -2.92062759e-06, -3.24845314e-06, -2.02655792e-06])
 message: 'Optimization terminated successfully.'
    nfev: 169240
     nit: 18
    njev: 40
  status: 0
 success: True
       x: array([ 4.92644317e-03, -9.02890363e-07,  1.18364701e-06, ...,
       -1.50136144e-04, -1.70713674e-04, -1.52406730e-04])
```

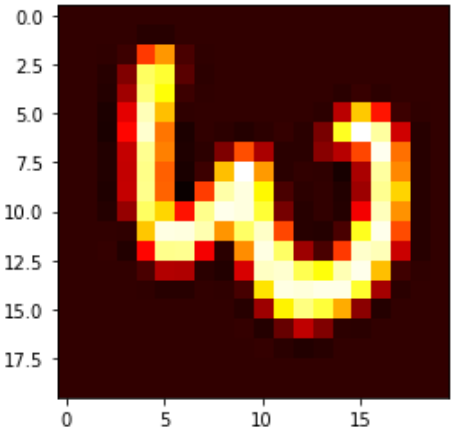
E o único dígito classificado corretamente foi o dois (muito provavelmente a rede sofreu com o fenômeno de vanishing gradient). A seguir, seis dos nove dígitos classificados erroneamente:



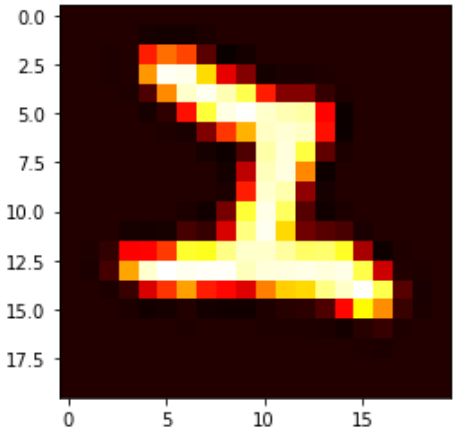
(a) Dígito 0/ Predição 2



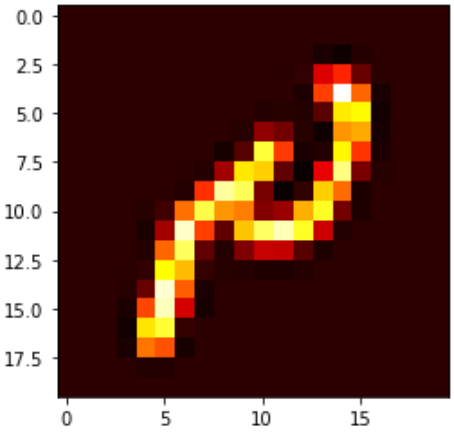
(b) Dígito 1/ Predição 2



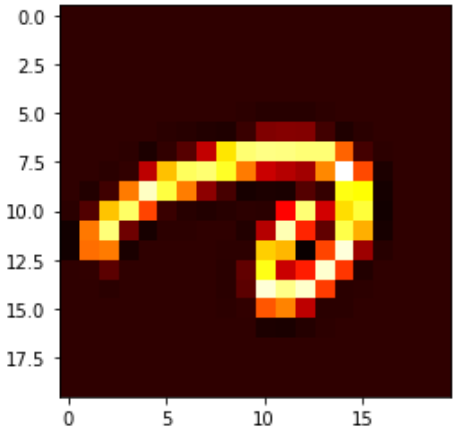
(c) Dígito 3/ Predição 2



(d) Dígito 4/ Predição 2



(e) Dígito 5/ Predição 2



(f) Dígito 6/ Predição 2

Fizemos também um teste com quatro camadas, 20 neurônios em cada camada escondida, 400 iterações máximas, $\lambda = 1$ e gtol fixado em 10^{-4}. Novamente,[obtivemos apenas um acerto e uma taxa de eficácia de 10%, com a única diferença para o caso anterior sendo o dígito classificado corretamente o 1.

Com esses dois resultados anteriores, é seguro afirmarmos que a rede neural não lida bem com um número de camadas $n > 3$ para um número de exemplos tão baixo e infelizmente o teste com um número maior de exemplos se torna inviável pois o custo/tempo computacional se tornaria demasiadamente alto.

Para um teste com três camadas, 30 neurônios na camada escondida, 400 iterações máximas, $\lambda = 0.5$ e gtol fixado em 10^{-4}, obtivemos dez acertos e uma taxa de eficácia de 100%, com o seguinte resultado derivado da função *minimize*:

```
Optimization terminated successfully.
         Current function value: 2.451278
         Iterations: 149
         Function evaluations: 3677618
         Gradient evaluations: 298
     fun: 2.4512782714356094
     jac: array([ 3.02791595e-05,  0.00000000e+00,  0.00000000e+00, ...,
       -2.35438347e-05, -1.14738941e-05, -3.28123569e-05])
 message: 'Optimization terminated successfully.'
    nfev: 3677618
     nit: 149
    njev: 298
  status: 0
 success: True
       x: array([ 1.21557426e+00, -9.18732849e-08,  1.77777959e-07, ...,
       -2.35037339e-01, -3.63770029e-01, -3.03835196e-01])
```

Por fim, em nosso último teste utilizamos três camadas, 15 neurônios na camada escondida, 400 iterações máximas, $\lambda = 0.5$ e gtol fixado em 10^{-4}, novamente obtendo dez acertos, o que equivale a uma taxa de eficácia de 100% e com o seguinte resultado derivado da função *minimize*:

```
Optimization terminated successfully.
         Current function value: 1.100677
         Iterations: 138
         Function evaluations: 2297472
         Gradient evaluations: 372
     fun: 1.1006770724415014
     jac: array([ 3.02493572e-06,  0.00000000e+00,  0.00000000e+00, ...,
        5.48362732e-06,  2.47359276e-06, -4.93228436e-06])
 message: 'Optimization terminated successfully.'
    nfev: 2297472
     nit: 138
    njev: 372
  status: 0
 success: True
       x: array([ 1.22282398e-01,  3.73199363e-08, -2.11579931e-08, ...,
       -5.14342752e-01,  9.30118945e-01, -9.74965079e-01])
```

Com isso, concluímos que o gradiente conjugado lida bem com problemas de multiclassificação, como pudemos visualizar em sua aplicação para uma base com dez dados. Seu alto custo computacional, porém, inviabiliza aplicações para uma ampla base de dados e, com isso, para o caso de redes neurais, o método do gradiente descendente se mostrou superior visto que o mesmo é consideravelmente mais rápido e gera uma boa taxa de eficácia, podendo ser tranquilamente aplicado a uma quantidade grande de dados.

2.8 O Algoritmo do Gradiente Conjugado

Para a implementação do gradiente conjugado nos baseamos no algoritmo anteriormente citado desenvolvido para o gradiente de descida. Retiramos, porém, a função responsável pela execução do gradiente de descida em si e adicionamos algumas que nos foram úteis para este modelo. Assim, as novas funções são:

```
def Vectorize(thetas):
    dim=[]
    for h in thetas:
        dim.append(h.shape[0]*h.shape[1])
    vector=thetas[0].flatten()
    for r in thetas[1:]:
        vector=np.hstack([vector,r.flatten()])
    return vector, dim

def Matrix(vector,b,a,m,n,dim,nbr_classes):
    matrix=[]
    matrix.append(vector[:dim[0]].reshape(a[0],n+1))
    aux=cp.deepcopy(dim)
    for i in range(1,b-2):
        aux=cp.deepcopy(dim)
        matrix.append(vector[aux[i-1]:(aux[i-1]+aux[i])].reshape(a[i],a[i-1]+1))
        aux[i]=aux[i-1]+aux[i]
    g=np.sum(dim[:-1])
    matrix.append(vector[g:].reshape(nbr_classes,a[-1]+1))
    return matrix
def assist(vector,X,Y,m,n,b,nbr_classes,k, a, dim):
    return Compute_Cost(vector, X, Y, m, n, b, nbr_classes, k, a, dim)[0]
```

E nosso "hub" de execução ficou organizado da seguinte forma:

```
data_2=pd.read_csv(r'C:\Users\USUARIO\Desktop\Neural Network\imageMNIST.csv')
data_3=pd.read_csv(r'C:\Users\USUARIO\Desktop\Neural Network\labelMNIST.csv')
X,Y,m,n,nbr_classes=Load()
a=1
X_Assist=np.zeros((10,X.shape[1]))
Y_Assist=np.zeros((10,Y.shape[1]))
for t in range(0,9):
    Y_Assist[t,:]=Y[a,:]
    X_Assist[t,:]=X[a,:]
    a=a+500
X_Assist[9,:]=X[4800,:]
Y_Assist[9,:]=Y[4800,:]
X=X_Assist
Y=Y_Assist
m=X.shape[0]
n=X.shape[1]-1
b=int(input("Qual seria o número de camadas? "))
a = list(map(int,input("\nQual seria o número de neurônios em cada camada escondida? ").strip().split()))[:n]
thetas=Theta_Initialization(b,a,X,Y)
vector,dim=Vectorize(thetas)
verifica=cp.deepcopy(vector)
matrix=Matrix(vector,b,a,m,n,dim,nbr_classes)
alpha=int(input("\nQual seria o número de iterações? "))
lambd=float(input("\nQual seria o valor de lambda? "))
t=minimize(assist,vector,args=(X,Y,m,n,b,nbr_classes,lambd,a,dim), method='CG', jac=None, tol=None, callback=None, options=
print(t)
thetas=Matrix(t.x,b,a,m,n,dim,nbr_classes)
pred,real,total=Prediction(Y,X,b,m,thetas)
print("\nO modelo obteve um total de ",total," classificações corretas" )
print("\nRepresentando uma taxa de eficácia de: ","{:.2f}".format((total/m)*100),"%")
q=int(input("\nGostaria de ver os dígitos que foram classificados incorretamente? (1 para sim, 2 para não) "))
if q==1:
    missFeedback(X,Y,pred)
```

Adicionamos a função "$Vectorize$" para automatizar o processo de transformar as matrizes de peso em um único grande vetor; adicionamos também como retorno uma lista denominada "dim", responsável por armazenar a dimensão de cada matriz de peso, possibilitando assim a sua reconstrução futura.

A função "$Matrix$" faz o exato inverso da função "$Vectorize$", ou seja, ela utiliza um grande vetor com todas as composições das matrizes de peso para reconstruir tais matrizes. Por fim, a função "$assist$" apenas retorna o custo gerado pela função "$ComputeCost$".

Em relação ao "hub" de controle, foram necessárias algumas alterações em relação ao anteriormente retratado: primeiramente, selecionamos dez exemplos da matriz X original e seus dez correspondentes em Y, um para cada valor de dígito (0-9). Isso foi necessário pois o gradiente conjugado possui um custo computacional consideravelmente alto, ou seja, é um algoritmo com execução lenta, portanto se torna inviável sua aplicação com os 5000 exemplos da matriz X, visto que o tempo para conclusão se torna excessivamente alto.

Em seguida, para a implementação do gradiente conjugado, utilizamos a função *minimize* do pacote *scipy.optimize* e fomos variando os parâmetros como forma de obter diferentes resultados para uma análise futura. A seguir, um exemplo de como a função *minimize* retorna o resultado, no caso, para uma rede neural com 3 camadas, 15 neurônios na camada escondida e valor de $\lambda = 1$:

```
Optimization terminated successfully.
         Current function value: 3.121572
         Iterations: 91
         Function evaluations: 1124032
         Gradient evaluations: 182
     fun: 3.121571900443427
     jac: array([-2.30073929e-05, -1.19209290e-07,
1.78813934e-07, ...,
       -9.35792923e-06,  2.27391720e-05,  5.03957272e-05])
 message: 'Optimization terminated successfully.'
    nfev: 1124032
     nit: 91
    njev: 182
  status: 0
 success: True
       x: array([-1.39297734e+00, -1.09547752e-06,
1.87860399e-06, ...,
        1.62228953e-01,  1.41199455e-01, -1.90331930e-01])
```

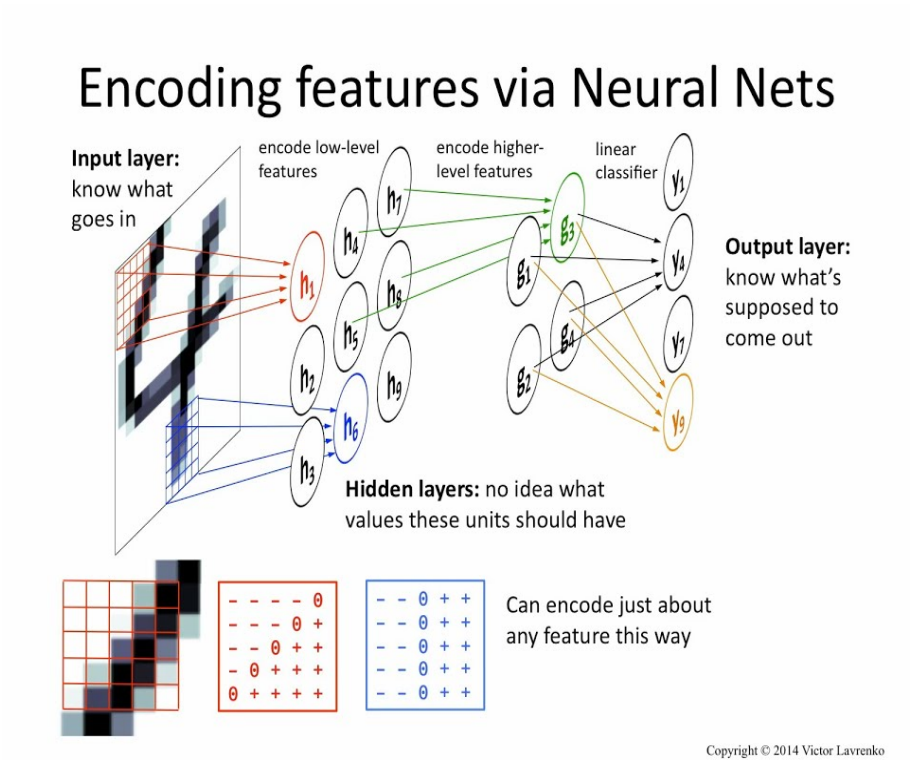
2.9 Visualização de Imagens Para Cada Unidade Escondida

Para possibilitar a visualização das imagens para cada unidade escondida, ou seja, um vislumbre da representação visual de como cada unidade escondida atua na rede, adicionamos ao modelo baseado no método do gradiente descendente uma função denominada "$image_visualization$" responsável por apresentar ao usuário as imagens após a execução da função do gradiente descendente.

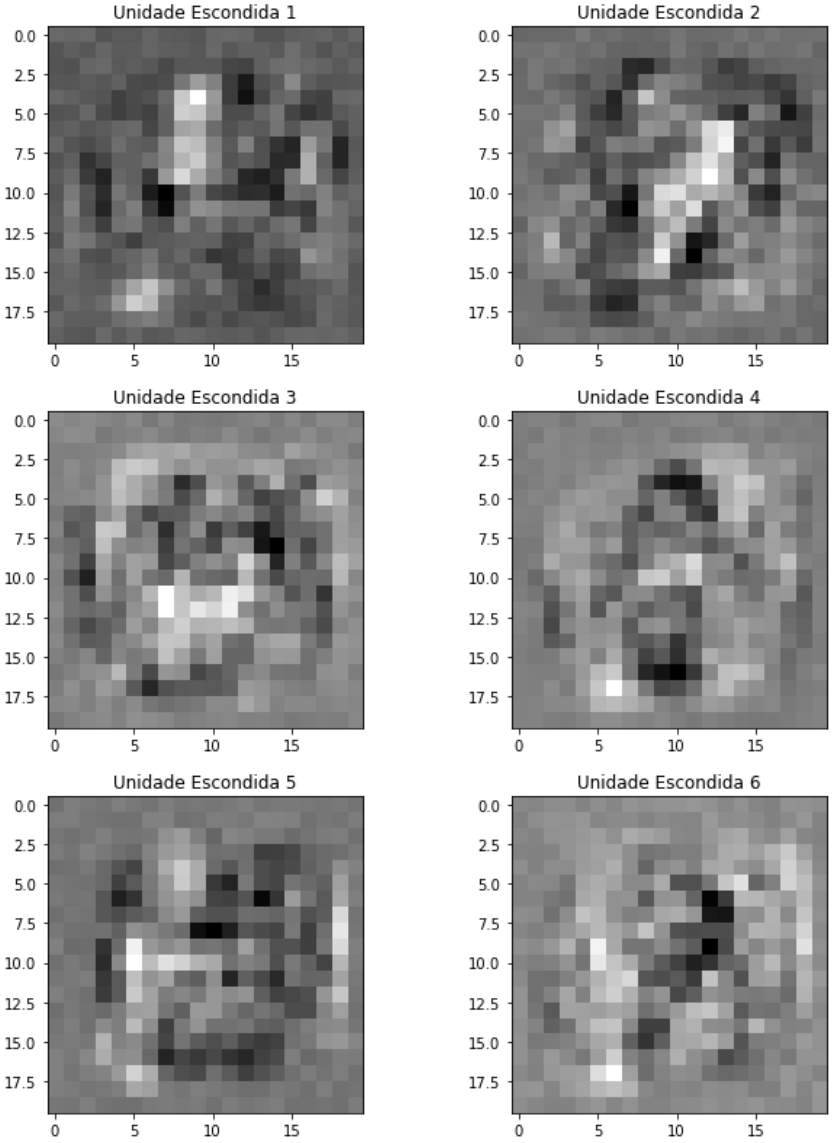
Tal função funciona para qualquer número de camadas e neurônios em cada camada esconida. Segue-se como ficou o resultado final da função:

```
def image_visualization(thetas):
    visu=cp.deepcopy(thetas[0])
    for i in range(visu.shape[0]):
        visu[i,1:]=(visu[i,1:]-np.min(visu[i,1:]))/(np.max(visu[i,1:])-np.min(visu[i,1:]))
        plt.figure()
        plt.imshow(visu[i,1:].reshape(20,20),cmap="gray")
        plt.title("Unidade Escondida {}".format(i+1))
        plt.show()
```

A seguir, um exemplo intuitivo de como as unidades escondidas "armazenam" sua atuação na rede:

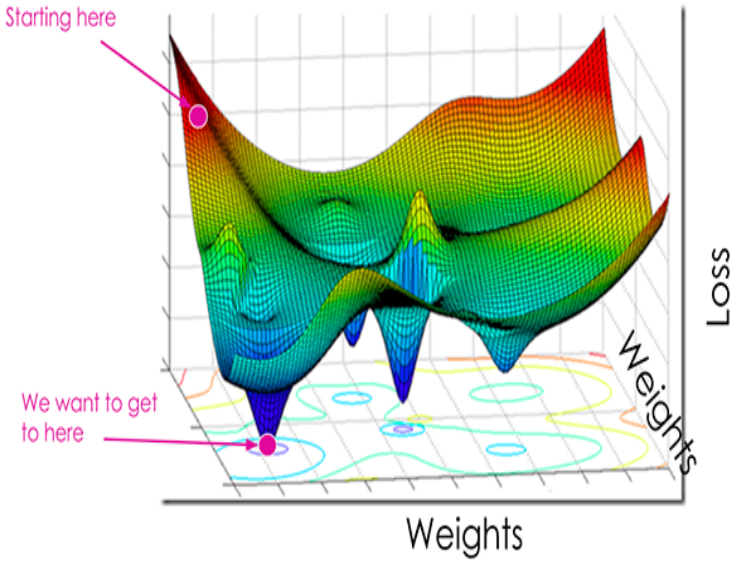


E em seguida, o exemplo de como ficam as imagens para uma rede de três camadas, 6 neurônios na camada escondida, 1200 iterações, taxa de aprendizado igual a 1 e $\lambda = 0.5$:

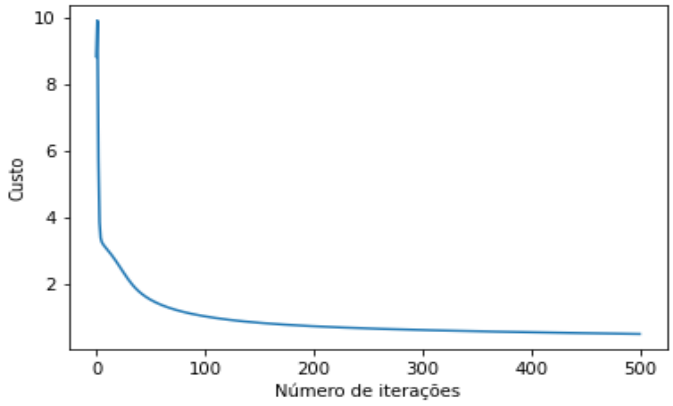


2.10 Anexos da Parte I

Em tal tópico realizaremos testes com variações pertinentes nos parâmetros das redes neurais, buscando uma análise mais aprofundada e, de certa forma, curiosa perante a implementação do modelo.

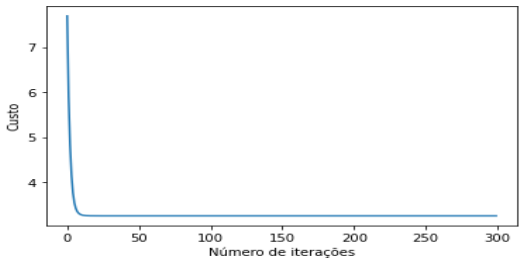


Primeiramente, realizamos um teste com um caso de uma rede de três camadas, com 100 neurônios na camada escondida, uma taxa de aprendizado de 0.8 e um valor de λ igual a 0.001. Com um número de 500 iterações, obtivemos um total de 4669 dígitos classificados corretamente, o que equivale a 93.40% de taxa de eficácia, nos levando a seguinte curva de custos:



Um ponto interessante a se observar nessa curva de custos é que, por conta do alto número de neurônios na camada escondida, temos um pico de custos na primeira iteração, que logo é corrigido pelo próprio modelo.

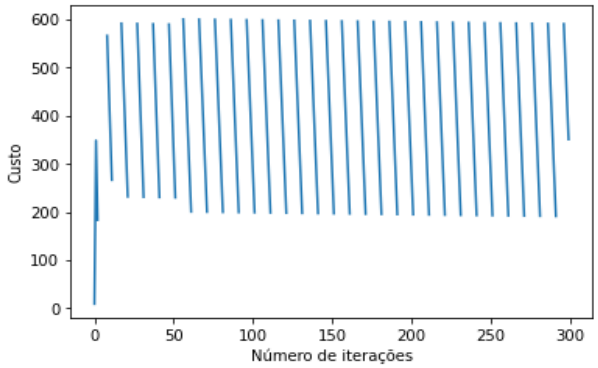
Em seguida, realizamos um teste com uma rede neural de dez camadas, sendo que suas camadas escondidas possuíam 5, 10, 15, 10, 5, 10, 15, e 5 neurônios em cada camada respectivamente além dos parâmetros um $\lambda = 0.5$, taxa de aprendizado igual a 0.5 e 300 iterações. Como era esperado, o número de acertos foi baixíssimo, 491 classificações corretas no total, o que equivale a 9.82% de taxa de eficácia, e sua curva de custos ficou da seguinte forma:



É fácil perceber que em dado momento o modelo estagnou (com um custo de aproximadamente 3.25) e parou de ser treinado, e isso tem como causa o número excessivo de camadas na rede neural para um número relativamente "pequeno" de dados de treinamento, o que contribui para o fenômeno do "Vanishing Gradient", quando os gradientes tornam-se tão pequenos que os pesos da rede passam a não ser atualizados de forma apropriada. O fenômeno é bem vísivel analisando o valor dos pesos da faixa entre a primeira e a segunda camada escondida:

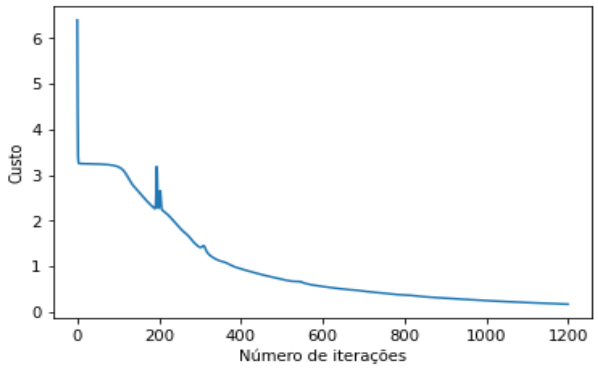
1.3096e-09	4.14549e-05	-1.54603e-05	-4.89957e-07	-2.6921e-05	1.03377e-05	4.70725e-06	1.55916e-05	3.38115e-05	-3.80481e-05	7.97633e-06
3.03084e-08	-2.93609e-05	-3.37794e-05	3.52576e-05	1.80837e-05	2.93047e-05	2.97601e-05	2.21929e-05	-3.66722e-05	5.00043e-06	-4.08207e-05
1.08737e-08	-1.02098e-05	3.60482e-06	-1.27335e-05	-1.35397e-05	1.09801e-06	2.59532e-05	1.58107e-05	2.03268e-05	-3.97783e-05	2.07891e-05
-8.05844e-09	2.25596e-05	4.47538e-05	9.13056e-07	-2.15394e-05	-2.97979e-05	1.15742e-05	6.50144e-06	-1.49874e-05	-3.75171e-05	2.00921e-05
1.51485e-08	-4.72711e-05	-3.5334e-05	-1.66859e-05	3.50378e-05	-3.72559e-05	8.69024e-06	7.38842e-06	-1.77479e-06	-4.48198e-05	-3.1631e-05
-6.41065e-10	3.99131e-05	2.21507e-05	-3.84594e-05	8.18406e-07	1.11754e-05	4.32672e-05	2.98602e-05	2.92674e-05	1.48886e-05	-6.8237e-06
3.8776e-09	-2.44522e-05	-1.83677e-05	-2.34484e-05	2.5125e-06	2.63089e-05	5.43624e-06	2.97193e-05	3.8801e-05	-1.25905e-05	-1.20444e-05
-6.31986e-09	-4.61517e-05	6.31905e-06	-3.09897e-06	-1.13392e-05	-1.62753e-05	4.36489e-05	3.41722e-05	9.47345e-06	-1.32944e-05	3.05445e-05
-4.49537e-09	2.8744e-05	1.02818e-05	-2.78498e-05	1.52369e-05	-2.15681e-05	-4.42381e-05	-4.42334e-05	3.20854e-05	-9.80001e-06	3.17331e-05
-6.29477e-09	-2.31517e-05	1.24642e-05	-2.12975e-05	-2.13354e-05	1.06233e-05	3.92235e-05	-8.76922e-06	-4.18282e-05	2.73385e-05	-1.89699e-06
-1.09059e-08	3.87439e-05	-3.51803e-05	-3.34751e-05	3.21108e-05	-4.51873e-05	2.40345e-05	3.47066e-05	-1.91903e-05	2.36594e-05	-2.26076e-05
-1.04672e-08	3.23149e-05	-2.70942e-05	5.7828e-06	2.94775e-05	1.70853e-06	-1.94934e-05	2.71885e-05	1.4951e-05	4.03513e-05	-4.56961e-05
-1.07348e-08	-2.37705e-05	2.13463e-05	-2.9523e-05	2.73862e-05	-4.13161e-05	-2.4695e-05	4.61589e-05	3.35292e-05	4.47497e-05	3.00743e-05
7.09658e-09	-4.6887e-05	-1.61542e-05	3.65866e-05	-3.4156e-05	-3.14108e-05	2.55851e-05	-4.377e-05	1.91767e-05	-2.19256e-06	-4.66038e-05
-1.2195e-08	-1.79006e-05	9.29818e-06	-3.87626e-05	3.24396e-05	2.63571e-05	2.66447e-05	8.4407e-06	7.0006e-06	-3.81936e-05	-1.55044e-05

Fazendo um teste com um valor de taxa de aprendizado extremamente alto, conseguimos perceber nitidamente um fenômeno de oscilação na curva de custos, como ocorre no seguinte exemplo, onde utilizamos uma taxa de aprendizado de 100, quatro camadas, com 30 neurônios em cada camada escondida, um $\lambda = 0.01$ e 300 iterações. Como esperado, a rede sequer iniciou um treinamento e se manteve estática e com a resposta padrão de 500 classificações corretas, o que equivale a 10% de taxa de eficácia. A curva de custos resultante foi a seguinte:



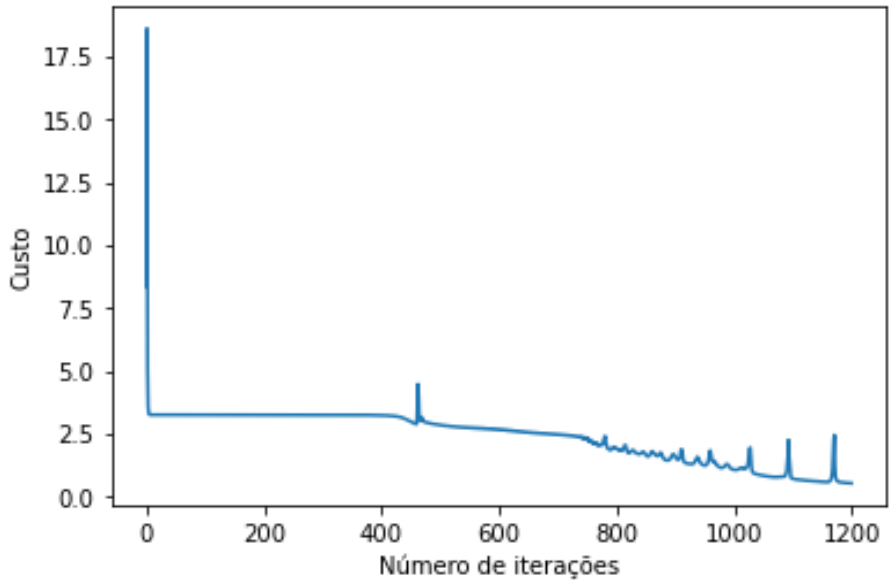
Em alguns pontos o custo foi registrado como "NaN", o que reforça como o modelo "estourou".

No teste seguinte, resolvemos observar como uma rede neural de cinco camadas, com 30 neurônios em cada camada escondia, se comportava com um alto número de iterações. Assim, definimos 1200 delas com uma taxa de aprendizado de 1.2 e um λ de 0.01. Obtivemos uma taxa de eficácia de 98.54%, o que equivale a 4926 classificações corretas (Um valor extremamente alto). A curva de custos resultante foi a seguinte:



Apesar de um repique na iteração 194, o custo retomou a queda e se estabilizou em um patamar baixo.

Para o próximo teste, resolvemos combinar um número alto de camadas com um número alto de neurônios por camada escondida, trabalhando então com uma rede neural de seis camadas com 100 neurônios em cada camada escondida, além de uma taxa de aprendizado de 1.5 e um $\lambda = 0.01$. Como resultado, o modelo atingiu uma taxa de eficácia de 92.38%, o que equivale a um total de 4618 classificações corretas. As curva obtida foi a seguinte:



Por fim, decidimos realizar um teste de apenas uma iteração com a matriz X e a matriz Y completas com o método do gradiente conjugado. Para tal, utilizamos um valor de $\lambda = 0.01$, uma rede de três camadas e 25 neurônios na camada escondida. O resultado final foi de uma taxa de eficácia de 18.10%, o que equivale a um total de 905 classificações.

```
     fun: 3.465854955674759
     jac: array([-2.27160156e-02,  1.78813934e-07,  0.00000000e+00,
...,
        4.78932261e-03,  8.88657570e-03,  9.76192951e-03])
 message: 'Maximum number of iterations has been exceeded.'
    nfev: 30858
     nit: 1
    njev: 3
  status: 1
 success: False
       x: array([ 0.15867483,  0.09740126,  0.00325796, ...,
0.28649258,
       -0.37419391,  0.14589194])
```

3 Parte II: Validação e Seleção de Modelos

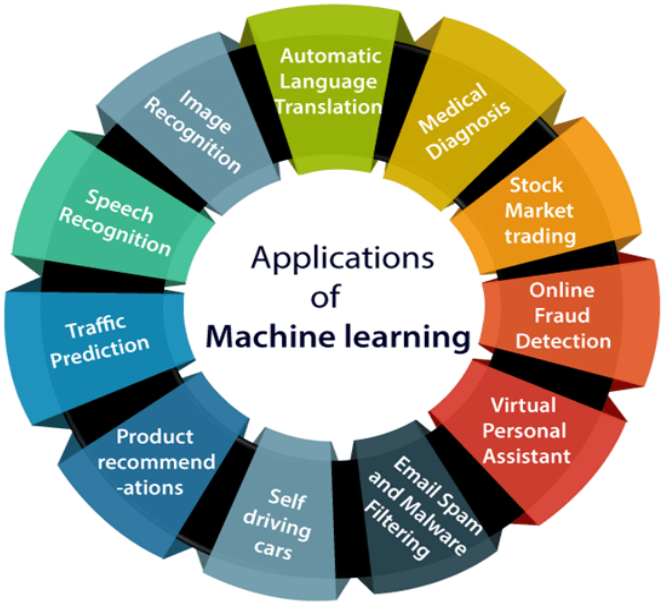
3.1 Introdução

Neste tópico proposto visamos implementar técnicas que nos permitam avaliar a eficiência do modelo de forma adequada, gerando análises que possam indicar pontos positivos e negativos do modelo desenvolvido anteriormente na primeira parte. Importante ressaltar que a implementação base que lidaremos é a que utiliza o método de gradiente descendente para a atualização dos pesos.

3.2 Importância de Avaliar um Modelo

Com o avanço e barateamento do armazenamento de dados, além de novidades em técnicas que agilizam e permitem uma maior aplicação de modelos de Machine Learning, vimos uma ampliação considerável dos campos que podem se utilizar de tais tecnologias. Existem, porém, peculiaridades em cada âmbito que fazem com que seja necessário avaliar os parâmetros (e hiperparâmetros) a cada modelo implementado, visto que as necessidades e principalmente,a precisão necessária para cada exemplo variam consideravelmente.

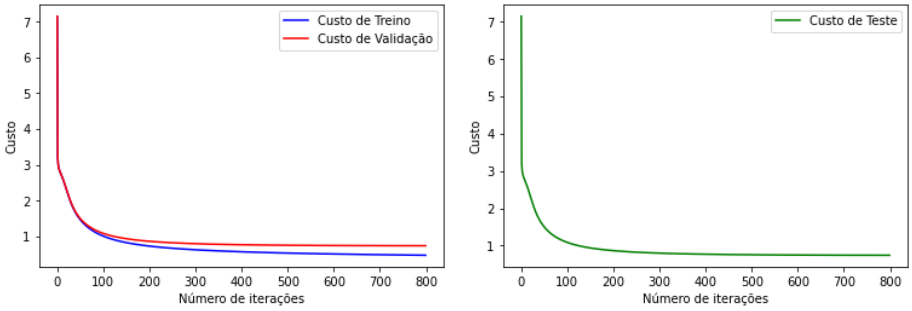
Com base nisso, existem algumas possíveis abordagens para maximizar a utilidade de um modelo em um dado contexto: podemos citar o aumento no número de dados disponíveis para o treino e construção do modelo, e técnicas de avaliação (como separar os dados em grupos de teste, treino e validação) que permitem que nós vislumbremos se o modelo oferece uma boa generalização (basicamente se ele não sofre com overfitting ou underfitting), entre outros.



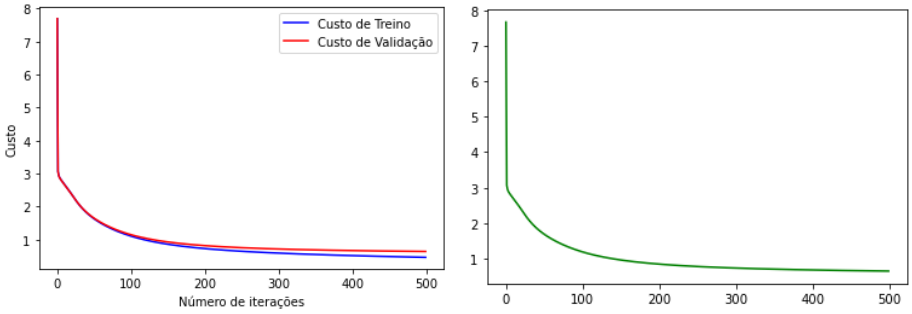
3.3 A Divisão em Treino, Teste e Validação

Para a resolução de tal questão, implementamos um modelo que permite ao usuário a escolha de como será feita a divisão nos três grupos: treino, teste e validação. O algoritmo seleciona aleatoriamente os dados do dataframe, mantendo a proporcionalidade, e calcula o custo por iteração para cada um desses grupos.

Assim, para um teste que realizamos com uma divisão de 60% dos dados para treino, 20% para validação e 20% para teste, com um número de camadas igual a três, 25 neurônios na camada escondida, taxa de aprendizado igual a um, $\lambda = 0.2$ e 800 iterações, obtivemos os seguintes gráficos:



Para o grupo de treino tivemos um custo final de 0.33, no grupo de teste tivemos um custo final de 0.62 e, no grupo de validação, um custo de 0.63. Como podemos observar, os custos foram consideravelmente parecidos, o que indica que o modelo lida bem com o problema proposto.
Executamos também um teste com a mesma divisão anterior, taxa de aprendizado igual a 0.8, $\lambda = 0.1$ e o número de iterações igual a 500, e assim obtivemos o seguinte resultado:



O custo final do grupo de teste foi de 0.63, do grupo de validação foi 0.64 e do grupo de treino foi 0.46. As curvas indicadas demonstram que esse é um ótimo ajuste.

3.4 O Algoritmo Para a Divisão dos Grupos

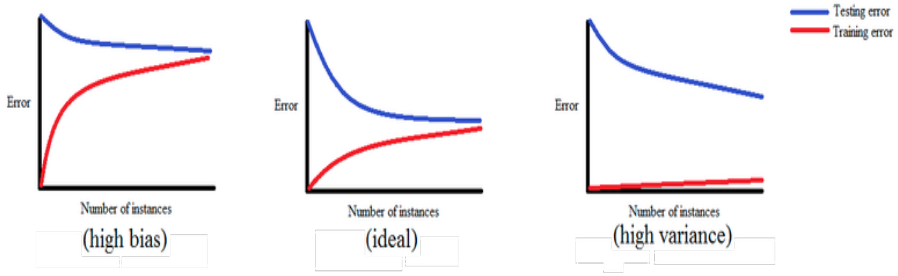
Utilizamos o modelo anteriormente implementado com o método de gradiente descendente como base para o desenvolvimento de nossa nova implementação. Assim, apenas removemos as funções anteriores que não seriam úteis e adicionamos uma função denominada "*Division*", responsável pela divisão dos três grupos (treino, teste e validação) de forma aleatória e estratificada. Assim, segue a nova função implementada:

```
def Division(div,data_2,data_3,m,n,nbr_classes):
    data_joined = data_2.join(data_3)
    data_random = data_joined.sample(frac=1)
    m_train=int(m*div[0])
    m_test=int(m*div[1])
    m_valid=int(m*div[2])
    df_train = data_random.iloc[:m_train,:]
    X_train, y_train = df_train.iloc[:,:-1].values, df_train.iloc[:,-1].values
    X_train = np.append(np.ones([m_train,1]),X_train,axis=1)
    df_test = data_random.iloc[m_train:(m_train+m_test),:]
    X_test, y_test = df_test.iloc[:,:-1].values, df_test.iloc[:,-1].values
    X_test = np.append(np.ones([m_test,1]),X_test,axis=1)
    df_valid=data_random.iloc[(m_train+m_test+2):,:]
    X_valid, y_valid= df_valid.iloc[:,:-1].values, df_valid.iloc[:,-1].values
    X_valid=np.append(np.ones([m_valid,1]),X_valid,axis=1)
    aux_1= []
    for i in range(m_train):
        aux_1.append(np.array([1 if y_train[i] == j else 0 for j in range(nbr_classes)]))
    Y_train = np.array(aux_1).reshape(-1, nbr_classes)
    aux_2= []
    for k in range(m_test):
        aux_2.append(np.array([1 if y_test[k] == h else 0 for h in range(nbr_classes)]))
    Y_test = np.array(aux_2).reshape(-1, nbr_classes)
    aux_3= []
    for t in range(m_valid):
        aux_3.append(np.array([1 if y_valid[t] == l else 0 for l in range(nbr_classes)]))
    Y_valid=np.array(aux_3).reshape(-1,nbr_classes)
    return X_train, X_test, X_valid, Y_train, Y_test, Y_valid,m_train,m_test,m_valid
```

A função se utiliza das porcentagens oferecidas pelo usuário para fazer a montagem correta das matrizes X e Y para cada grupo. Tais matrizes já são retornadas prontas para serem usadas na rede neural.

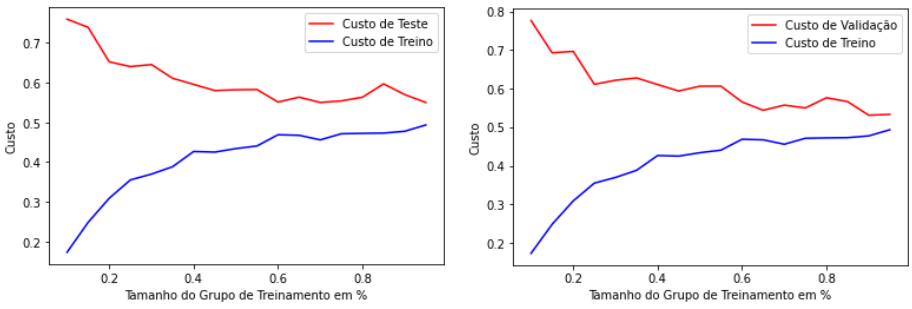
3.5 Curvas de Aprendizado

As curvas de aprendizado são ferramentas importantes na identificação e constatação acerca da qualidade do ajuste de um modelo para um dado problema. Através das mesmas conseguimos constatar se ocorrem fenômenos prejudiciais para a generalização da implementação, como o overfitting e o underfitting, e se existe um viés ou variância alta nos dados, entre outros.



Com vistas disso, implementamos um algoritmo que automatiza o processo de seleção das porcentagens: o mesmo varia a porcentagem do grupo de treino de 10% até 95% aumentando 5% a cada iteração. Importante ressaltar que o gráfico é afetado por parâmetros como o λ, taxa de aprendizado e até mesmo o número de iterações escolhida, e existe também uma incerteza gerada pelo fato de, a cada iteração, o dataframe original ser "misturado", ou seja, os dados para as matrizes X e Y de cada grupo mudam a cada iteração.

Com isso, para um caso que usa três camadas escondidas, um valor de λ igual a 0.01, uma taxa de aprendizado de 0.8 e um número de iterações igual a 500, obtivemos os seguintes gráficos:



Considerando as incertezas já comentadas, podemos concluir que um modelo com tais parâmetros possui um ajuste bom para o problema e a quantidade de dados proposta, gerando uma boa generalização.

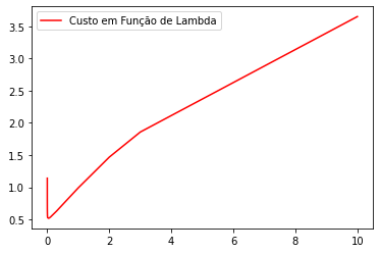
3.6 Escolha Automatizada do λ

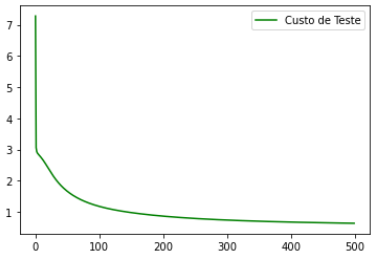
O λ é um dos parâmetros essenciais quando analisamos redes neurais regularizadas, tendo a capacidade de ajustar o modelo e propiciar uma generalização mais ampla. Se escolhido de forma errônea, porém, pode ocasionar problemas e distorções na implementação.

Dessa forma, desenvolvemos um modelo que escolhe automaticamente o valor de λ que será utilizado na rede neural através de uma análise de custos referentes ao grupo de validação. Para tal, criamos uma lista *lambdas* contendo alguns valores de λ sugeridos pelo professor, mas também adicionamos alguns valores "extras", fazendo-a tomar a seguinte forma:

$$lambdas = [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 0.005, 0.05, 0.5, 2, 0.0001]$$

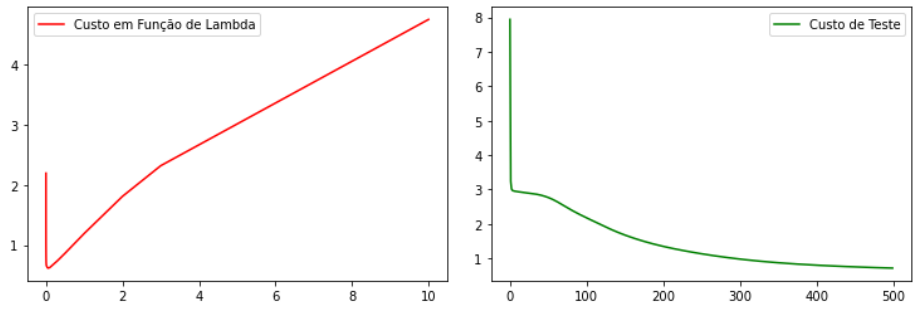
O programa testa de forma automática todos os λ propostos (sendo inclusive possível expandir tal lista se for da vontade do usuário), e salva o que propicia o menor custo no grupo de validação para ser utilizado no gradiente descendente principal. Com isso, para um caso de 3 camadas, sendo 25 neurônios na camada escondida, uma taxa de aprendizado de 0.8 e 500 iterações, com os grupos de treino, teste e validação divididos respectivamente em 60%, 20% e 20%, obtivemos um valor de λ ótimo equivalente a 0.0001, e as seguintes curvas para o custo do grupo de validação em função de λ e do custo do grupo de teste em função do número de iterações:





Como podemos observar, os gráficos confirmam a veracidade dos resultados, e demonstram que realmente o algoritmo seleciona a melhor escolha possível de λ, adequada aos outros parâmetros da rede. O custo final do grupo de teste foi 0.48

Também realizamos um teste com quatro camadas, 30 e 15 neurônios nas camadas escondidas, 400 iterações, taxa de aprendizado de 0.8 e divisão de grupos igual ao caso anterior, obtendo as seguintes curvas:



O valor de λ escolhido para este caso foi 0.05, e o custo final do grupo de teste foi 0.71.

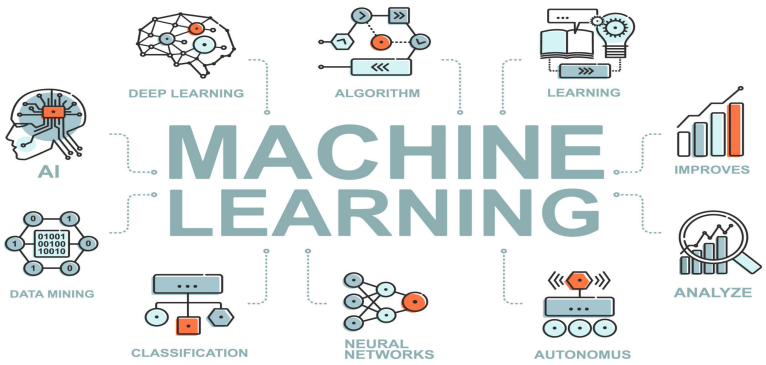
3.7 Algoritmo Para a Automatização da Escolha de λ

Novamente, nos baseamos no modelo anteriormente desenvolvido que utiliza o método do gradiente descendente e, assim, apenas adicionamos uma função nova denominada "$TesteLambda$", responsável por automatizar o processo de escolha do λ. Assim, a nova função implementada é a seguinte:

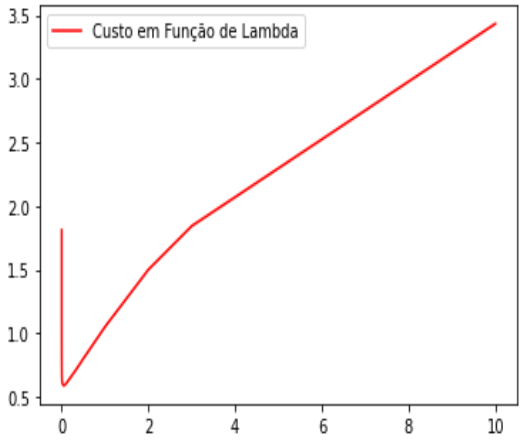
```
def TesteLambda(X_train,Y_train,m_train,thetas,apren,alpha,n,nbr_classes,X_validation,Y_validation,m_validation,X_test,Y_tes
  lambdas=[0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 0.005, 0.05, 0.5, 2, 0.0001]
  lambd=100
  aux=1000
  lambdas.sort()
  custos=[]
  for k in lambdas:
      J_validation=GradientDescent(X_train,Y_train,m_train,thetas,apren,alpha,k,n,nbr_classes,X_validation,Y_validation,m_va
      custos.append(J_validation[-1])
      if J_validation[-1] < aux:
          aux=cp.deepcopy(J_validation[-1])
          lambd=k
  plt.plot(lambdas,custos,"r",label="Custo em Função de Lambda")
  plt.legend()
  plt.show()
  return lambd
```

3.8 Anexos da Parte II

Em tal tópico visamos analisar e testar a rede neural de diferentes formas, obtendo assim resultados novos que podem ser interessante para a compreensão geral do projeto proposto.

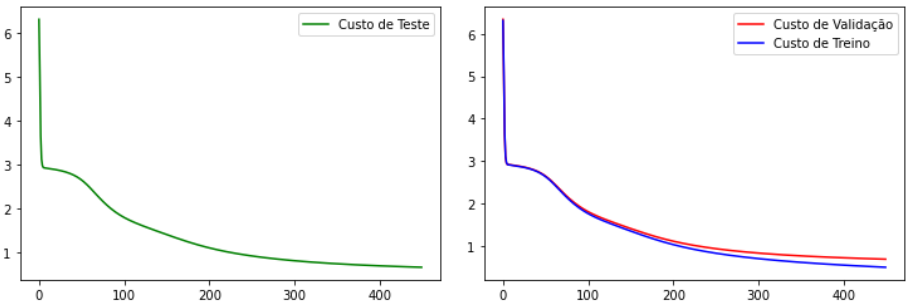


Primeiramente, vamos iniciar um teste para verificarmos como fica a curva do grupo de validação e do grupo de treino utilizando o melhor λ possível, visto que sua escolha agora é automatizada. Com isso, vamos utilizar uma taxa de aprendizado de 0.8 e 450 iterações em uma rede neural de quatro camadas com 40 e 60 neurônios em cada camada escondida. Importante ressaltar que fizemos uma divisão de 50% para o grupo de treino e 25% para o teste e para a validação, obtendo assim as seguintes curvas:



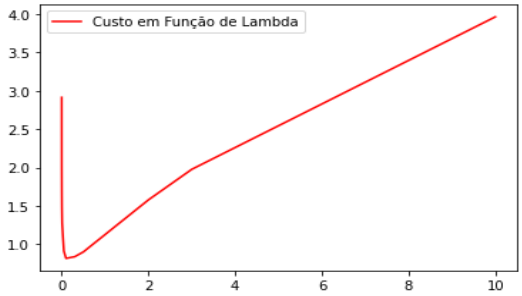
Tal curva diz respeito ao custo do grupo de validação em função de λ, e teve

como resultado para o λ ótimo 0.05. Com isso, tivemos as seguintes curvas de custo para os grupos de treino, validação e teste:

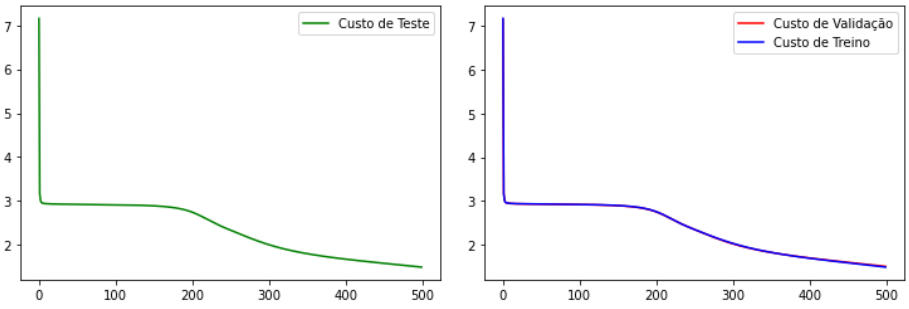


Conseguimos observar claramente que o modelo se ajusta bem ao problema proposto.

Em seguida, decidimos testar um modelo com o mesmo tipo de separação do anterior, mas com um número maior de camadas, cinco, com 25,30 e 35 neurônios em cada uma. Assim, com uma taxa de aprendizado de 0.7 e um número de iterações igual a 500, obtivemos os seguintes resultados:



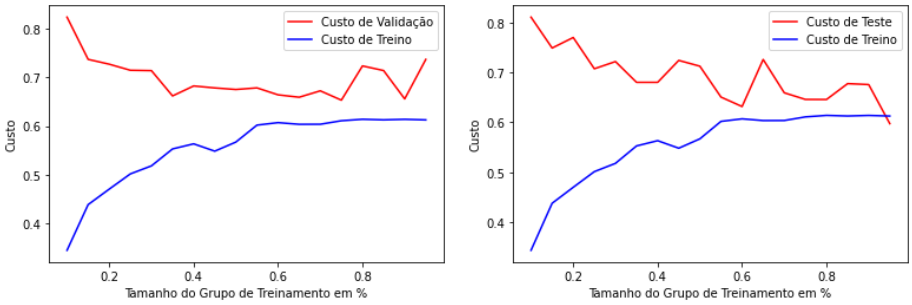
O λ ótimo teve como resultado 0.1, gerando as seguintes curvas:



O gap entre a curva de validação e a curva de treino é mínimo, ao ponto de

se tornar quase imperceptível, indicando um ótimo ajuste do modelo.

Por fim, temos um teste para a geração da curva de aprendizado para uma rede neural de três camadas, com 25 neurônios na camada escondida, um λ de 0.01, taxa de aprendizado de 0.8 e 300 iterações, obtendo assim os seguintes resultados:



É vísivel que com essas curvas de aprendizado geradas o modelo não se ajusta muito bem ao problema, e uma alteração em seus hiperparâmetros se faz necessária.

4 Conclusão

A partir de todo projeto desenvolvido, tanto no âmbito da montagem da rede neural em si quando na seleção e avaliação de modelos, pudemos constatar o quão útil e interessante é a implementação de algoritmos de Machine Learning; a quantidade de problemas possíveis de serem abordados a partir de tal conhecimento é gigantesca, e avaliar, analisar e construir criticamente modelos como os que fizemos permite que possamos expandir nosso aprendizado, colocando realmente a "mão na massa" e conhecendo aplicações reais que permeiam o mundo moderno.

Em relação aos modelos abordados, foi interessante observar que, apesar de convergir em um número menor de passos, o método do gradiente conjugado é extensamente mais custoso computacionalmente em relação ao método do gradiente descendente. Em um primeiro momento isso é um tanto quanto contra-intuitivo, mas quando paramos para analisar a forma de resolução, passa a fazer pleno sentido.

5 Bibliografia:

[1] M. A. Gomes Ruggiero, V. L. da Rocha Lopes. Cálculo Numérico - Aspectos Teóricos e Computacionais, 2ª edição, Editora Pearson, 1997.

[2] https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

[3] https://www.ibm.com/cloud/learn/neural-networks

[4] https://towardsdatascience.com/gradient-descent-in-python-a0d07285742f

[5] https://www.skynettoday.com/overviews/neural-net-history

[6] https://pt.coursera.org/learn/machine-learning

[7] https://realpython.com/python-ai-neural-network/

[8] https://datascience-enthusiast.com/DL/Improving_DeepNeural_Networks_Gradient_Checking.html

[9] http://deeplearning.stanford.edu/tutorial/supervised/DebuggingGradientChecking

[10] https://towardsdatascience.com/a-short-introduction-to-model-selection-bb1bb9c73376

[11] Slides e Jupyter Notebooks disponibilizados pelo professor da disciplina (MS960), João Batista Florindo.