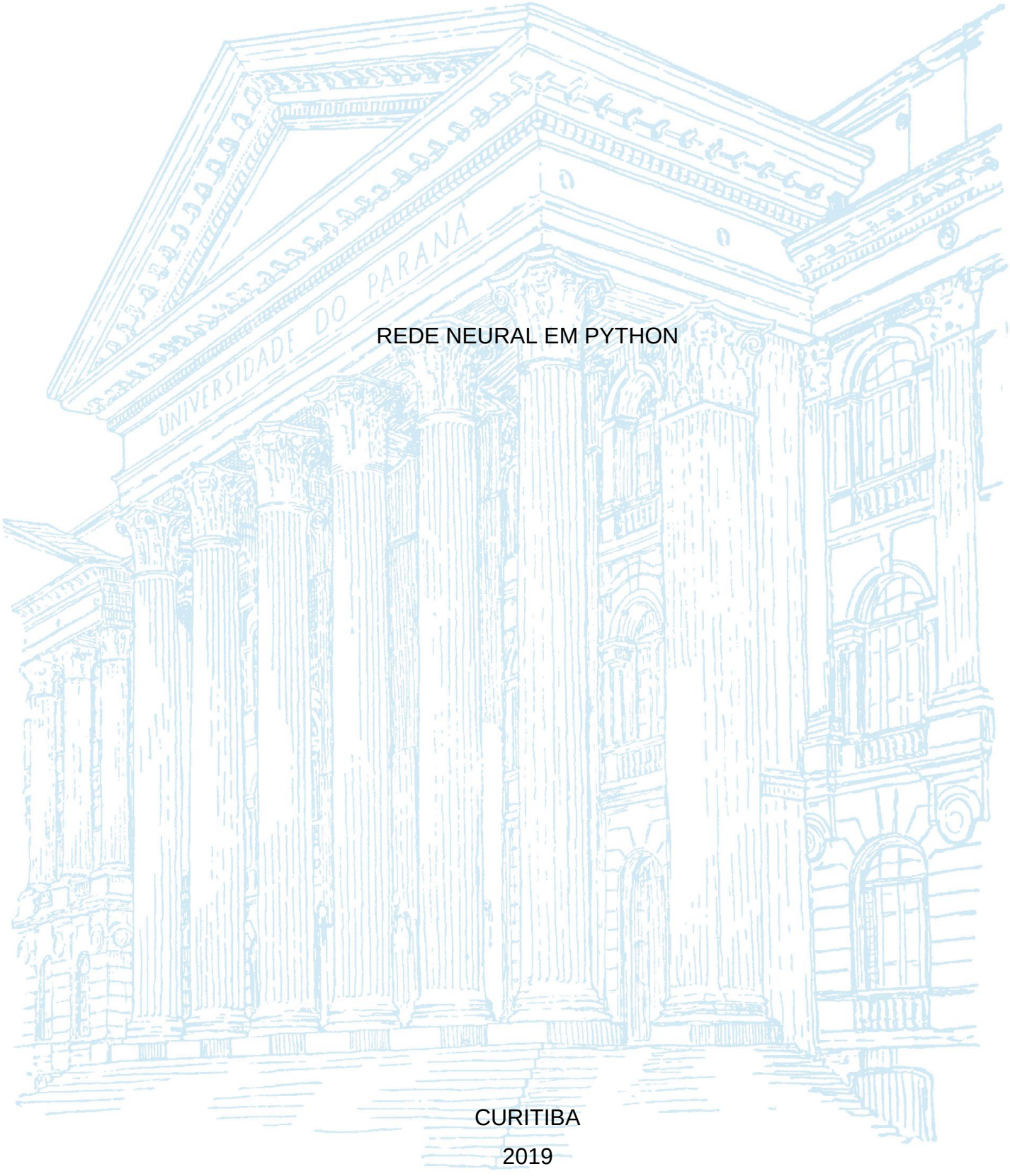
UNIVERSIDADE FEDERAL DO PARANÁ

MARCOS VINICIUS STRIEDER



REDE NEURAL EM PYTHON

CURITIBA

2019

MARCOS VINICIUS STRIEDER

REDE NEURAL EM PYTHON

Trabalho apresentada a disciplina Fundamentos de Programação de Computadores do Curso de Graduação em Matemática da Universidade Federal do Paraná.

Orientador: Prof Jackson Antônio do Prado Lima

CURITIBA

2019

RESUMO

Este trabalho consiste no desenvolvimento e treinamento de uma rede neural para reconhecimento de dígitos escritos à mão em imagens, usando para tal os conhecimentos adquiridos na matéria durante o semestre.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Deep Learning (traduzido aproximadamente como aprendizagem profunda) é um conjunto de métodos e técnicas baseados em redes artificiais neurais com o objetivo de treinar computadores para a realização das mais diversas tarefas, como por exemplo reconhecimento de fala e de imagens. Devido a sua grande aplicabilidade a uma ampla gama de problemas, é um tópico de conhecimento bastante valorizado no mercado e em pesquisas e dessa forma é um assunto interessante para aplicação dos conhecimentos de python.

Este trabalho tem o objetivo simples de realizar o treinamento de uma rede neural para reconhecimento de dígitos escritos à mão, levando no processo ao entendimento das bases teóricas de aprendizagem profunda e ao conhecimento de como implementa-las em alguns problemas simples.

2 FUNDAMENTAÇÃO TEÓRICA

A fim de se entender o programa, há três aspectos teóricos que devem ser explicados: a estrutura das redes neurais, a descida de gradiente e a retropropagação.

2.1 REDES NEURAIS ARTIFICIAIS

As redes neurais artificiais são os objetos básicos utilizados na aprendizagem profunda, e a estrutura que se deseja otimizar para a realização da tarefa em mão pelo computador.

A rede neural consiste basicamente de neurônios organizados em diferentes camadas, ligados por conexões entre si chamadas de pesos. Por neurônios se entende um nódulo que possuíra um valor associado a si; os pesos também são números reais e relacionam os valores de uma camada de neurônios com os valores da próxima camada.

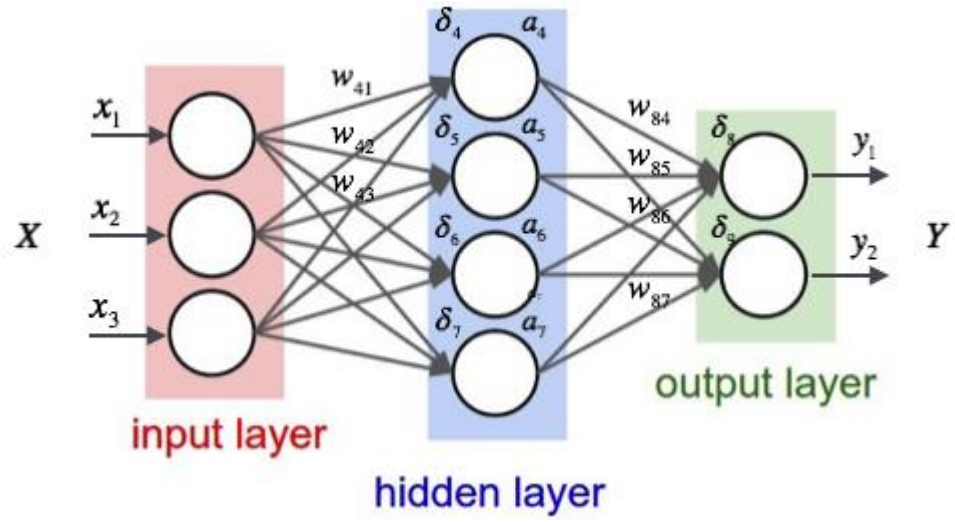


Figura 1 - Esquema de uma rede neural

Os valores dos neurônios da primeira camada consistem dos inputs recebidos pelo programa. Cada neurônio da primeira camada se conecta então com um neurônio da segunda por meio de um peso (os pesos são representados por linhas na imagem). Os valores de inputs e os pesos determinam então os valores dos neurônios da segunda camada da seguinte forma: dado um neurônio da segunda camada, se multiplica o peso da conexão do neurônio com o input do neurônio que o peso conecta; se realiza isso com todos os neurônios da primeira

camada, e se soma todos eles, obtendo-se um valor; então se soma esse valor a uma quantia chamada de viés, dependente somente do neurônio da segunda camada. Por fim se aplica uma função, chamada função sigmoide, e assim se obtém o valor do neurônio.

Por exemplo, suponha que a primeira camada consista de dois neurônios com inputs x1 e x2; e dado um neurônio da segunda camada, suponha que os pesos da conexão desse neurônio com x1 e x2 sejam w1 e w2; por fim, seja b o viés associado ao neurônio. Então se calcula x1*w1 + x2*w2 + b e se aplica a função sigmoide para se obter o valor do neurônio (também chamado de valor de ativação).

A função sigmoide usada aqui é f(x) = 1/(1+*e*^(-x)), onde *e* é a constante de Euler. A razão do uso dessa função é que o somatório dos pesos e inputs com o viés pode assumir valores arbitrariamente grandes, e essa função normaliza o output o pondo em uma faixa de 0 a 1. Além disso a função também é continua, o que vai se mostrar importante no cálculo do gradiente.

De forma semelhante a que os inputs da primeira camada determinam os valores da segunda, os valores da segunda determinam os valores da terceira, e assim por diante até a camada final, chamada de camada dos outputs. As camadas intermediárias entre a camada dos inputs e a camada dos outputs são chamadas de camadas ocultas.

A ação da rede neural pode ser representada por meio de matrizes da seguinte forma: dado o primeiro neurônio da segunda camada, se organiza os pesos

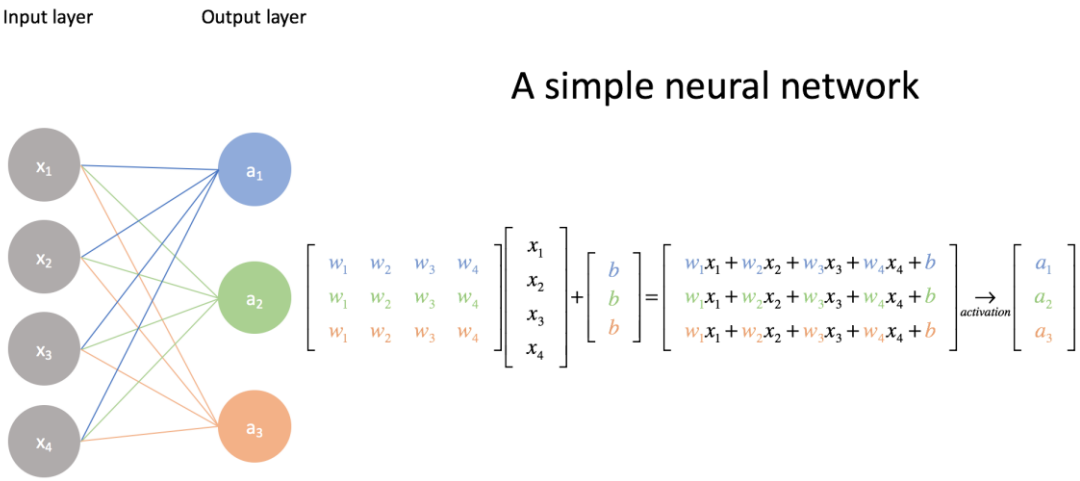


Figura 2 - Matrizes associadas a rede neural

ligando os neurônios da primeira camada com ele na primeira linha da matriz; já os pesos ligando os neurônios da primeira camada com o segundo neurônio da segunda camada são dispostos na segunda linha; e assim por diante. Com isso se obtém a matriz dos pesos de tamanho NxM, onde N é o número de neurônios da segunda camada e M é o número de neurônios da primeira. Já os vieses da segunda camada são dispostos em uma matriz Nx1. Desta forma, pode-se considerar o input como uma matriz Mx1, e a ação da rede neural se resume a multiplicação e soma de matrizes (junto com a aplicação da função sigmoide. Essa visão da rede como um conjunto de matrizes será importante no programa, pois a rede neural lá gerada é simplesmente um conjunto de matrizes de pesos e matrizes de vieses.

2.2 DESCIDA DE GRADIENTE

Uma vez iniciada uma rede neural, se deseja realizar o seu treinamento. Para isso é definido uma função contínua, chamada de função de custo, que mede o desempenho da rede. Por exemplo, neste trabalho se deseja que a rede diga qual digito de 0 a 9 está escrito em uma imagem; para tal ela deve ter de output uma lista com 9 dígitos, sendo que a posição do maior digito é o ‘palpite’ da rede (por exemplo um output (0.1, 0.3, 0.1, 0.1, 0.9, 0,8, 0.1, 0.1, 0.1) seria um palpite de número 5); nesse exemplo a rede oferece um palpite, mas mesmo que o palpite de número 5 esteja correto, há alguns problemas: a rede apresentou um alto valor na posição 6, indicando uma certa incerteza (provavelmente mudando alguns pixels da imagem sem mudar o digito escrito seria possível fazer a rede ter como output o número 6 e portanto o palpite estaria correto); então o palpite, embora correto, ainda não é o ideal. A fim de medir o ‘quão correto’ esta o palpite se compara então o output da rede com o output ideal (no caso o output ideal seria (0, 0, 0, 0, 1, 0, 0, 0, 0)) calculando a soma dos quadrados da diferença de cada entrada – esta é um exemplo de uma função de custo (e a função de custo usada nesse trabalho).

Assim se possui uma rede neural e uma função de custo medindo o desempenho da rede, sendo a função de custo uma função da rede. Para obter então uma rede melhor, se deseja minimizar a função de custo. Do cálculo se sabe que o gradiente da função indica a direção que maximiza a função localmente. Como se quer minimizar a função, é necessário dar um passo na direção negativa

ao gradiente. Em resumo, sabendo o gradiente se conhece como minimizar a função.

Em teoria o gradiente poderia ser calculado analiticamente; no entanto, como se tem uma função com potencialmente centenas de variáveis (aqui a variável da função de custo é a rede neural, que nada mais é do que um conjunto de pesos e vieses – portanto a função de custo pode ser vista como uma função de cada um dos pesos e cada um dos vieses), se torna intratável calcular o mínimo de primeiros princípios. É usado então um procedimento denominado retropropagação para calculo do gradiente.

Em resumo: se tem uma função de custo que se deseja minimizar, onde suas variáveis são os pesos e vieses da rede; para minimiza-la se calcula o gradiente e se atualiza a rede na direção oposta ao gradiente; realizando esse processo milhares de vezes, a rede neural consegue minimizar a função de custo e se torna assim boa na predição de imagens.

3 METODOLOGIA

Terminada a fundamentação teórica será agora aplicado os princípios para a construção de uma rede neural que consiga reconhecer números de 0 a 9 escritos a mão.

As bibliotecas utilizadas são pickle, numpy, matplotlib e random. Dessas a mais utilizada é a numpy, uma biblioteca bastante usada em trabalhos científicos e que foca na manipulação de matrizes e arrays multidimensionais.

Os dados utilizados para o treinamento da rede vêm do MNIST database (*Modified National Institute of Standarts and Technology database*), um conjunto de dados amplamente utilizado para treino e teste em Deep Learning. Esse database contém milhares de imagens de resolução 28x28 com algarismos escritos á mão.

Esses dados estão postos em formato já adequado para uso por python no arquivo ‘data.txt’, e é utilizada a biblioteca pickle para importa-los. Os dados estão organizados da seguinte maneira: há uma lista training_data, consistindo de 50000 entradas, onde cada entrada contém uma array com 784 valores representando a imagem, bem como outra array de 10 valores representando o output ideal da rede (por exemplo se a imagem consiste de um 5, o array representando o output ideal seria um array consistindo todo de zeros com exceção de um 1 na quinta posição). Há também uma lista testing_data, com 10000 entradas, onde cada entrada consiste da imagem (um array com 784 entradas) e do número na imagem (só o número, não o vetor output desejado).

```
import pickle
import numpy as np
from matplotlib import pyplot as plt
import random

training_data, testing_data = pickle.load(open('data.txt','rb'))
'''importa os dados que serão utilizados pelo programa; training_data
consiste de uma lista com 50000 entradas, onde cada entrada consiste de dois
elementos: uma lista com 748 numeros (a imagem) e um vetor representando
o output desejado (por exemplo, se a imagem consiste de um 5 então o output
desejado é (0 0 0 0 0 1 0 0 0 0) ); ja testing_data contém 10000 elementos,
cada elemento consiste de uma imagem(um array com 748 números) e o número ao
qual corresponde a imagem '''
```

Figura 3 – Importação dos dados

A próxima função definida é a função imagem(x); o input x dessa função é o array da imagem; o output consiste de uma visualização da imagem, usando para tal a biblioteca matplotlib.

```
def imagem(x):
    #vai mostrar a imagem contida nos dados
    x = np.reshape(x,(28,28))
    plt.imshow(x, interpolation='nearest')
    plt.show()
```

Figura 4 - Código da função Imagem(x)

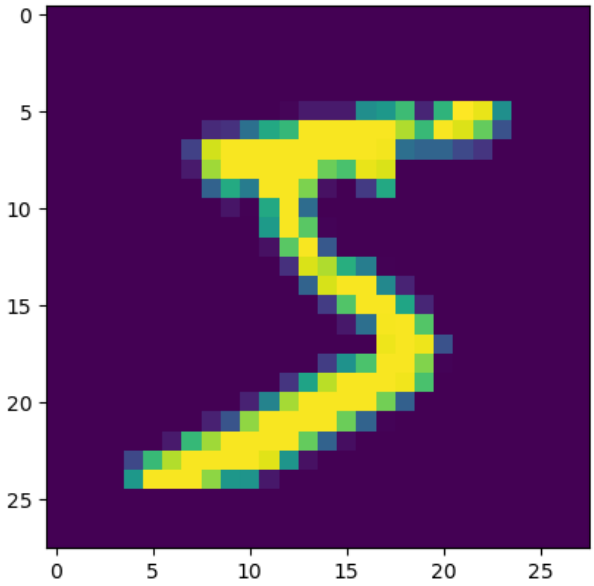


Figura 5 - Output da função Imagem

As próximas funções são sigmoide(x) e derivada_sigmoide(x), que simplesmente aplicam a função sigmoide e a sua derivada ao input; esse input pode ser uma array ou matriz graças ao uso da biblioteca numpy (nesse caso a função é aplicada a cada entrada da matriz).

```
def random_neural(sizes):
    """cria uma rede neural baseada na lista sizes; por ex
    se sizes = [1,2,3], vai criar uma rede com 3 camadas e um neuronio na
    primeira camada, 2 na segunda e 3 na terceira """
    n = len(sizes)#número de camadas
    vieses = []
    for i in range(1,n):
        temp = np.random.randn(sizes[i],1)
        vieses.append(temp)
    pesos = []
    for i in range(len(sizes)-1):
        b = np.random.randn(sizes[i+1],sizes[i])
        pesos.append(b)
    net = [vieses, pesos]
    return net
```

Figura 6 – Função random_neural()

A próxima função é random_neural(sizes), que cria uma rede neural aleatoriamente; o input sizes é uma lista contendo os tamanhos de cada camada da rede (por exemplo uma rede igual à da Figura 1 teria de input [3, 4, 2], e uma igual à

da figura 2 teria input [4,3]). Essa função gera uma lista com 2 entradas, uma entrada sendo uma lista contendo as matrizes dos pesos e outra entrada contendo as matrizes dos vieses. Para gerar os dados de forma aleatória se usou a função numpy.random.randn(m,n) que gera uma array m por n de números aleatórios. A rede neural que queremos deve ler 784 inputs e devolve 10 outputs, então deve possui 784 neurônios na primeira camada e 10 na última. Um exemplo de uma rede assim seria random_neural([784, 30, 10]) (de fato esse é o tamanho da rede que será usada quando correrrmos o programa).

```
def resultado(net, x):
    #x é o input que a rede neural net recebe e a função retorna o resultado
    for i in range(len(net[0])):
        x = sigmoide(np.dot(net[1][i],x)+net[0][i])
        ###np.dot(x,y) realiza a multiplicação de duas matrizes
    return x


def avaliacao (net, test_data):
    #olha quantas imagens a rede acertou
    i = 0
    for k in range(len(test_data)):
        r = resultado(net, test_data[k][0])
        r = np.argmax(r) #argmax traz a posicao com maior valor da lista
        if r == test_data[k][1]:
            i = i+1
    return i
```

Figura 7 – Funções resultado e avaliacao

As próximas funções são resultado(net, x) e avaliacao(net, test_data). A função resultado recebe de input uma rede neural e o input que essa rede neural deve receber, e retorna o array que resulta da rede. A função avaliacao recebe de input a rede e a testa em todos os casos dos dados de teste (no nosso caso esses casos estão contidos na lista testing_data importada no começo, que contém 10000 imagens).

A próxima função é a função retropropagação. O parâmetro net consiste da rede neural. Já os parâmetros x e y consistem respectivamente da imagem (no formato de uma array 784x1) e o output y desejado (um array 10x1). A função então age e retorna dois arrays nabla_b e nabla_w, que consistem do gradiente calculado nesse exemplo específico. Dentro da função ocorre um cálculo complicado de derivadas parciais dos pesos e vieses; para proposito desse trabalho, a função é melhor vista como um black box que retorna o gradiente calculado em um único exemplo.

```
def retropropagação(net, x, y):
    ''' Net é a rede sendo usada; x é a imagem usada de input na rede, e
    y é o output desejado'''
    nabla_b = [np.zeros(b.shape) for b in net[0]]
    nabla_w = [np.zeros(p.shape) for p in net[1]]
    activation = x
    activations = [x]
    zs = []
    for b, w in zip(net[0], net[1]):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoide(z)
            activations.append(activation)
    delta = (activations[-1] - y) * derivada_sigmoide(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in range(2, len(net[0])+1):
            z = zs[-l]
            sp = derivada_sigmoide(z)
            delta = np.dot(net[1][-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

Figura 8 - Função Retropopagação

```
def update(net, mini_batch, eta):
    '''usa o mini_batch para calcular o gradiente aproximado, e então da um
    'passo' eta na direção do negativo do gradiente para otimizar a rede'''
    x = [np.zeros(b.shape) for b in net[0]]
    y = [np.zeros(p.shape) for p in net[1]]
    #cria matrizes no formato das matrizes de pesos e vieses mas com zeros de entradas
    for i in mini_batch:
        delta_x, delta_y = retropropagação(net, i[0], i[1])
        for l in range(len(x)):
            x[l] = x[l] + delta_x[l]
            y[l] = y[l] + delta_y[l]
    for i in range(len(net[0])):
        net[0][i] = net[0][i] - (eta/len(mini_batch))*x[i]
        net[1][i] = net[1][i] - (eta/len(mini_batch))*y[i]
```

Figura 9 – Função Update

A função update recebe de inputs os parâmetros net, mini_batch e eta. O parâmetro net é simplesmente a rede neural usada; o parâmetro mini_batch consiste de uma lista com apenas alguns elementos da lista testing_data; por exemplo, testing_data contém 50000 elementos, então uma lista com os primeiros 10 elementos seria um mini_batch; a função update usa a função retropopagação para calcular os gradientes em cada elemento do mini_batch, e soma esses gradientes, obtendo assim o gradiente de forma aproximada do gradiente real; o gradiente real seria aquele calculado usando os 50000 exemplos de treinos, mas se calcula apenas o gradiente aproximado usando 10 exemplos a fim de evitar o alto uso

computacional da alternativa. Por fim, eta é um número real indicando o tamanho do ‘passo’ a ser dado na direção do gradiente negativo, passo esse que idealmente vai minimizar a função de custo; assim a função vai atualizar a rede neural ao soma-la com o gradiente calculado no mini_batch vezes o passo eta.

```
def SGD(net, training_data, epocas, tamanho, eta, test_data):
    ''' SGD = stochastic gradient descent ou descida de gradiente estocástica.
    É a função que efetivamente vai treinar a rede. O training_data é os dados
    carregados no começo do programa, consistem de 50000 listas (x,y) onde x é a
    imagem e y é o output desejado; '''
    n = len(training_data)#vai ser 50000 no nosso caso
    for i in range(epocas):
        random.shuffle(training_data)#embaralha aleatoriamente os dados
        mini_batches = [training_data[k:k+tamanho] for k in range(0,n,tamanho)]
        for l in mini_batches:
            update(net, l, eta)
        print ('Epoca {0}: Acertos: {1} de {2}'.format(i, avaliacao(net, test_data), len(test_data)))
```

Figura 10 - Função SGD (Stochastic Gradient Descent)

Finalmente, a função SGD é que vai realizar o treinamento da rede. O parâmetro net é a rede neural a ser otimizada; training_data é o arquivo contendo as 50000 imagens de treino; test_data vai ser o arquivo com 10000 imagens inicial; basicamente, primeiro a função random.shuffle embaralha os dados de treinamento; os dados de treinamento são então divididos em mini_batches, onde a variável tamanho é o (!) tamanho escolhido de mini_batch. Por exemplo, se tamanho for 10, então as 50000 entradas do training_data serão divididas em 5000 pacotes, cada uma com 10 imagens; a rede neural vai então se otimizar com a função update em cada um desses mini_batches. Uma vez que a função passou por todos esses pacotes, se diz que foi completado uma época; o parâmetro épocas é então um inteiro dizendo quantas vezes esse processo deve ser reproduzido. Ao fim de cada época, se avalia o resultado da rede com os dados de teste usando a função avaliacao (net, test_data). Por fim, se printa os resultados da rede ao fim de cada época, possibilitando assim monitorar a evolução da rede.

4 RESULTADOS OBTIDOS

Ao final, se rodou o programa com três organizações diferentes; a arquitetura da rede neural usada é até certo ponto arbitrária, apesar da primeira camada sempre ter 784 neurônios e a última possuir 10 neurônios.

De forma similar, o número de épocas, os tamanhos dos mini_batches e o tamanho do passo ‘eta’ são arbitrários; há no entanto técnicas mais avançadas para otimizar o tamanho destes, mas estão fora do escopo desse trabalho.

O melhor resultado mostrou uma porcentagem de acerto de 95.05% na primeira configuração, embora a diferença entre a primeira e a terceira não tenha sido significativa.

```
>>> net = random_neural([784, 30, 10])
>>> SGD(net, training_data, 30, 10, 3.0, testing_data)
Epoca 0: Acertos: 8960 de 10000
Epoca 1: Acertos: 9177 de 10000
Epoca 2: Acertos: 9292 de 10000
Epoca 3: Acertos: 9363 de 10000
Epoca 4: Acertos: 9355 de 10000
Epoca 5: Acertos: 9406 de 10000
Epoca 6: Acertos: 9415 de 10000
Epoca 7: Acertos: 9423 de 10000
Epoca 8: Acertos: 9448 de 10000
Epoca 9: Acertos: 9458 de 10000
Epoca 10: Acertos: 9437 de 10000
Epoca 11: Acertos: 9466 de 10000
Epoca 12: Acertos: 9436 de 10000
Epoca 13: Acertos: 9454 de 10000
Epoca 14: Acertos: 9453 de 10000
Epoca 15: Acertos: 9473 de 10000
Epoca 16: Acertos: 9478 de 10000
Epoca 17: Acertos: 9457 de 10000
Epoca 18: Acertos: 9498 de 10000
Epoca 19: Acertos: 9471 de 10000
Epoca 20: Acertos: 9484 de 10000
Epoca 21: Acertos: 9494 de 10000
Epoca 22: Acertos: 9502 de 10000
Epoca 23: Acertos: 9462 de 10000
Epoca 24: Acertos: 9476 de 10000
Epoca 25: Acertos: 9489 de 10000
Epoca 26: Acertos: 9495 de 10000
Epoca 27: Acertos: 9503 de 10000
Epoca 28: Acertos: 9510 de 10000
Epoca 29: Acertos: 9505 de 10000
```

Figura 11 – Desempenho 1

```
>>> net = random_neural([784, 10])
>>> SGD(net, training_data, 30, 10, 3.0, testing_data)
Epoca 0: Acertos: 4899 de 10000
Epoca 1: Acertos: 4959 de 10000
Epoca 2: Acertos: 5765 de 10000
Epoca 3: Acertos: 6633 de 10000
Epoca 4: Acertos: 6694 de 10000
Epoca 5: Acertos: 6726 de 10000
Epoca 6: Acertos: 6720 de 10000
Epoca 7: Acertos: 7334 de 10000
Epoca 8: Acertos: 8236 de 10000
Epoca 9: Acertos: 8307 de 10000
Epoca 10: Acertos: 8317 de 10000
Epoca 11: Acertos: 8341 de 10000
Epoca 12: Acertos: 8323 de 10000
Epoca 13: Acertos: 8322 de 10000
Epoca 14: Acertos: 8341 de 10000
Epoca 15: Acertos: 8326 de 10000
Epoca 16: Acertos: 8345 de 10000
Epoca 17: Acertos: 8353 de 10000
Epoca 18: Acertos: 8342 de 10000
Epoca 19: Acertos: 8332 de 10000
Epoca 20: Acertos: 8326 de 10000
Epoca 21: Acertos: 8344 de 10000
Epoca 22: Acertos: 8359 de 10000
Epoca 23: Acertos: 8339 de 10000
Epoca 24: Acertos: 8370 de 10000
Epoca 25: Acertos: 8348 de 10000
Epoca 26: Acertos: 8344 de 10000
Epoca 27: Acertos: 8366 de 10000
Epoca 28: Acertos: 8348 de 10000
Epoca 29: Acertos: 8355 de 10000
```

Figura 12 – Desempenho 2

```
>>> net = random_neural([784, 20, 20, 10])
>>> SGD(net, training_data, 30, 20, 5.0, testing_data)
Epoca 0: Acertos: 8870 de 10000
Epoca 1: Acertos: 9132 de 10000
Epoca 2: Acertos: 9207 de 10000
Epoca 3: Acertos: 9259 de 10000
Epoca 4: Acertos: 9288 de 10000
Epoca 5: Acertos: 9264 de 10000
Epoca 6: Acertos: 9341 de 10000
Epoca 7: Acertos: 9347 de 10000
Epoca 8: Acertos: 9347 de 10000
Epoca 9: Acertos: 9295 de 10000
Epoca 10: Acertos: 9350 de 10000
Epoca 11: Acertos: 9410 de 10000
Epoca 12: Acertos: 9406 de 10000
Epoca 13: Acertos: 9394 de 10000
Epoca 14: Acertos: 9382 de 10000
Epoca 15: Acertos: 9414 de 10000
Epoca 16: Acertos: 9413 de 10000
Epoca 17: Acertos: 9450 de 10000
Epoca 18: Acertos: 9441 de 10000
Epoca 19: Acertos: 9434 de 10000
Epoca 20: Acertos: 9469 de 10000
Epoca 21: Acertos: 9435 de 10000
Epoca 22: Acertos: 9419 de 10000
Epoca 23: Acertos: 9440 de 10000
Epoca 24: Acertos: 9446 de 10000
Epoca 25: Acertos: 9445 de 10000
Epoca 26: Acertos: 9416 de 10000
Epoca 27: Acertos: 9430 de 10000
Epoca 28: Acertos: 9445 de 10000
Epoca 29: Acertos: 9423 de 10000
```

Figura 13 - Desempenho 3

5 CONSIDERAÇÕES FINAIS

O trabalho foi uma boa maneira de exercitar os conhecimentos obtidos no semestre e permitiu ver um pouco das grandes possibilidades de projetos que a programação traz. As funções não exigiram nada muito diferente de alguns exercícios feitos no Ava e é assim gratificante obter resultados mais avançados usando um conhecimento básico.

5.1 RECOMENDAÇÕES PARA TRABALHOS FUTUROS

Em um trabalho de tema semelhante poderia ser usado um código similar e treiná-lo em outros databases (não são difíceis de encontrar, o próprio mnist possui outros), é possível explorar diferentes estruturas de rede neural e comparar os desempenhos relativos entre si, ou simplesmente tentar implementar técnicas mais avançadas de aprendizagem.

REFERÊNCIAS

Michel A. Nielsen. ‘Neural Networks and Deep Learning’, Determination Press, 2015

MNIST database. Disponível em <http://yann.lecun.com/exdb/mnist/> Acessado 25/11/2018

Biblioteca numpy. Disponível em <https://numpy.org/> Acessado 25/11/2018