Universidade De São Paulo - USP
Instituto de Ciências Matemáticas e de Computação - ICMC

Trabalho 1

Teste de performance: Multilayer-Perceptron

Bruno Bacelar Abe 9292858

Professora:
Profa. Dra. Roseli Romero

São Carlos, Setembro de 2018

Introdução

A rede neural *Multilayer Perceptron* (MLP) é uma rede que é constituída por mais de uma camada interna, que é chamada de *hidden layer* neste trabalho. O processo de aprendizado da rede é dividido em duas etapas principais, a de alimentação dos nós e propagação dos erros.

A etapa de propagação, também chamada de *feedforward* consiste em atualizar os valores de previsão de acordo com os pesos encontrados. O processo inicia-se na camada inicial (*input layer*) e alastra-se até a última camada (*output layer*). Uma vez que a alimentação encontra-se na última camada, o erro é retro propagado por toda a rede, atualizando-se os pesos nesse processo. A retro propagação dos erros, com à atualização dos pesos, é o que chamamos de *backpropagation* .

Neste trabalho, analisaremos o comportamento de uma MLP com duas bases de dados, uma para classificação de dados e outra para predição de resultados.

Para a classificação, utilizou-se os dados providenciados pela especificação do trabalho, os dados podem ser encontrados neste link. Basicamente, iremos classificar os dados em 3 categorias de vinhos (1, 2 e 3) de acordo com 13 parâmetros.

Para a predição, também utilizou-se a base de dados especificada no trabalho, sendo possível encontrá-la neste link . A ideia é aproximar o valor da latitude e longitude de origem da música a partir dos dados disponibilizados. São 70 colunas disponíveis na base de dados, sendo as duas últimas a latitude e longitude.

Implementação

Para implementação da rede MLP foi utilizado a linguagem de programação Python 3.6.5, utilizando o editor de textos *Sublime* no Windows 10 de 64 bits. Todos os testes foram realizados nesse mesmo computador.
Em relação ao exercício entregue, houve a modificação da implementação para que o código ficasse mais otimizado, porém, a ideia continua a mesma. Há um objeto chamado *MLP*, que possui todos os métodos necessários para o aprendizado da rede (*feedforward*, *backpropagation* e função de ativação). Os parâmetros desse objeto são o número de nós na camada inicial, escondida e final.

```
"""Recebe como parâmetros a quantidade de nós no input, camada do meio e output como parâmetros"""
class MLP():
    def __init__(self, n_nodeInput, n_nodeHidden, n_nodeOutput):
```

(Imagem 1)

Basicamente, executa-se a alimentação e propagação do erro até que o erro quadrado médio seja menor do que um *threshold* previamente definido.

```
while(it < 1000):
#while(gradErrorWhile > threshold):
    for p in x:
        expected = np.matrix(p[1])
        inputs = np.matrix(p[0])

        #fast foward
        output = self.ff(inputs)
        #Erro quadrado
        gradError = (np.power((expected - output), 2))
        gradErrorWhile = np.max(np.asarray(gradError[:]))
        error = (expected - output)

        #Deltas
        delta_output_layer = np.asmatrix(self.dsigmoide(output))
        delta_hidden_layer = self.dsigmoide(np.asarray(self.hiddenlayer_activations))
        d_out = (-1)*np.asarray(error)*np.asarray(delta_output_layer)
        error_hiddenLayer = np.dot(d_out, np.transpose(self.wout))
        d_hiddenLayer = np.asarray(error_hiddenLayer)*np.asarray(delta_hidden_layer)

        #Atualização dos pesos
        self.wout = self.wout - momentum *np.dot(np.transpose(self.hiddenlayer_activations), d_out)*learnigRate
        self.wh = self.wh - momentum*np.dot(np.transpose(inputs), d_hiddenLayer)*learnigRate
```

(Imagem 2)

Agora, todas as multiplicações de matrizes são realizadas pelas funções da biblioteca *numpy*, diferentemente do que era feito na implementação anterior. Além de deixar o código mais enxuto, a legibilidade é maior.
Variáveis importantes do objeto:

1. wout: pesos da camada de saída;
2. wh: pesos da camada intermediária;
3. bh: bias para a camada intermediária;
4. bo: bias para a camada de saída;
5. hiddenlayer_activations: valores após a função de ativação.

É importante salientar que houve uma normalização de todos os dados antes que a rede tivesse acesso aos mesmos. Como pré-processamento, obteve-se o maior elemento de todas as informações e dividiu-se todos os elementos pelo maior elemento, assim, temos todos os dados em um intervalo entre 0 e 1. Em relação às saídas, o mesmo foi feito. Na imagem abaixo, pode-se ver o pré-processamento para os dados usados na tarefa de classificação.

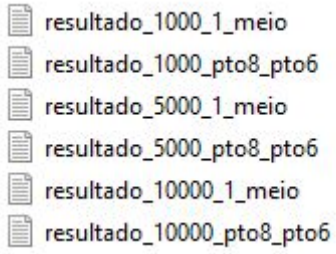
```
for i in range (len(file)):
    x = []

    x.append(file[i][0]/3)

    #Normaliza os dados
    y = np.array((file[i][1:])/bigger)
    l = y.tolist()
    w = [l, x]
```

(Imagem 3)

Quando o trabalho é executado, há a execução de um menu inicial, no qual é possível escolher uma ação.

1. Testar a rede nos dados de vinho (Wine);
2. Testar a rede nos dados de músicas (Tracks);
3. Testar a rede com a porta XOR.

Após a escolha da ação, é possível escolher o número de interações, momentum, *learning rate* e configuração das bases de treino e teste. Uma vez que tudo é escolhido, começa o treinamento. Todos os resultados são guardados nos arquivos de texto com nome inicial “resultado_alguma coisa.txt”, toda vez que uma nova execução é feita, os arquivos são sobrescritos.



(Imagem 4)

Testes

Classificação

Para a classificação, houve a variação das bases de dados em relação a original (visando aprender melhor). Há 3 variações da base de dados, a qual foi dividida em “treinamento” e “teste”.

1. Divisão 1: Houve a divisão da base original (que possuía 178 dados) em: 120 dados para treino e 58 para testes. Essa divisão ocorreu de maneira aleatória, evitando escolher apenas um tipo de dado;
2. Divisão 2: Houve a divisão da base original em duas metades iguais. Ou seja, uma porcentagem de 50% para testes e 50% para treino;
3. Divisão 3: Usa-se os 134 primeiros dados para treino, os dados restantes são usados para testes.

A configuração da rede foi a seguinte: 13 nós na camada inicial, 10 nós na camada intermediária e 1 nó na camada final. Para a classificação, usaremos interações (como sugere a especificação do trabalho).

1. Utilizando a seguinte configuração:

Momentum	1
Ciclos	1000
Velocidade aprendizado	0.5

Resultados	Acurácia
Teste 1	0.75
Teste 2	0.70
Teste 3	0.65

2. Utilizando a seguinte configuração:

Momentum	1
Ciclos	5000
Velocidade aprendizado	0.5

Resultados	Acurácia

Teste 1	0.77
Teste 2	0.80
Teste 3	0.58

3. Utilizando a seguinte configuração:

Momentum	1
Ciclos	10000
Velocidade aprendizado	0.5

Resultados	Acurácia
Teste 1	0.67
Teste 2	0.65
Teste 3	0.66

4. Utilizando a seguinte configuração:

Momentum	0.8
Ciclos	1000
Velocidade aprendizado	0.6

Resultados	Acurácia 1
Teste 1	0.72
Teste 2	0.64
Teste 3	0.73

5. Utilizando a seguinte configuração:

Momentum	0.8
Ciclos	5000
Velocidade aprendizado	0.6

Resultados	Acurácia
Teste 1	0.75

Teste 2	0.73
Teste 3	0.61

6.

Utilizando a seguinte configuração:

Momentum	0.8
Ciclos	10000
Velocidade aprendizado	0.6

Resultados	Acurácia
Teste 1	0.73
Teste 2	0.64
Teste 3	0.67

Conclusões classificação

Após os experimentos, foi possível observar que muitas interações prejudicam o aprendizado (basicamente em todos os casos, quando havia 10000 iterações os resultados eram menos equilibrados do que as de 5000). A mudança da taxa de aprendizagem e o *momentum* também impactaram negativamente no aprendizado.
Quanto à escolha das bases, pode-se notar que equilibrar bem as bases de treino e teste é importante para a generalização do aprendizado da rede. As duas primeiras bases obtiveram resultados parecidos, enquanto que a terceira obteve resultados muito ruins.

Aproximação

Assim como ocorreu na tarefa de classificação, houve a divisão da base de dados em 3, além da variação do número de interações, *momentum* e taxa de aprendizado. É importante salientar que, no site em que a base era disponibilizada, havia dois arquivos com dados parecidos, nesse trabalho utilizamos apenas uma das bases (default_features_1059_tracks).
As bases foram divididas da seguinte maneira:

1. Houve a divisão da base original em uma base de testes e outra de treinamento. A base de treinamento possui 860 dados, enquanto que a de testes ficou com o restante da base (escolhidos aleatoriamente);
2. Houve a divisão entre 700 dados para treinamento e o restante para testes, também aleatoriamente escolhidos;

3. Utiliza-se toda a base para treinamento e testes.

A configuração da rede foi: 68 nós na camada inicial, 45 na intermediária e 2 na camada de saída. Para a aproximação, houve a variação do número de interações, *momentum* e taxa de aprendizado.

1.

Momentum	1
Ciclos	1000
Velocidade aprendizado	0.5

Resultados	Erro1	Erro 2
Teste 1	1.5x10^-5	0.04
Teste 2	10^-3	4.8x10^-6
Teste 3	2x10^-16	9x10^-5

2.

Momentum	1
Ciclos	5000
Velocidade aprendizado	0.5

Resultados	Erro1	Erro 2
Teste 1	5.5 x 10^-6	2.7 x 10^-2
Teste 2	4.6 x 10^-6	2.1 x 10^-6
Teste 3	4.6 x 10^-6	3.3 x 10^-3

3.

Momentum	1
Ciclos	10000
Velocidade aprendizado	0.5

Resultados	Erro1	Erro 2
Teste 1	7.1 x 10^-6	2.5x10^-2
Teste 2	1.4 x 10^-10	1.9 x 10^-7
Teste 3	4 x 10^-10	2.8x10^-2

4.

Momentum	0.8
Ciclos	1000
Velocidade aprendizado	0.6

Resultados	Erro1	Erro 2
Teste 1	1.1 x10^-4	4.1x10^-3
Teste 2	7.2x10^-4	4.5x10^-3
Teste 3	3.5x10^-11	3.15x10^-4

5.

Momentum	0.8
Ciclos	5000
Velocidade aprendizado	0.6

Resultados	Erro1	Erro 2
Teste 1	1.2x10^-5	2n8 x 10^-2
Teste 2	8.2x10^-5	1.2x10^-5
Teste 3	5x10^-19	6.7x10^-8

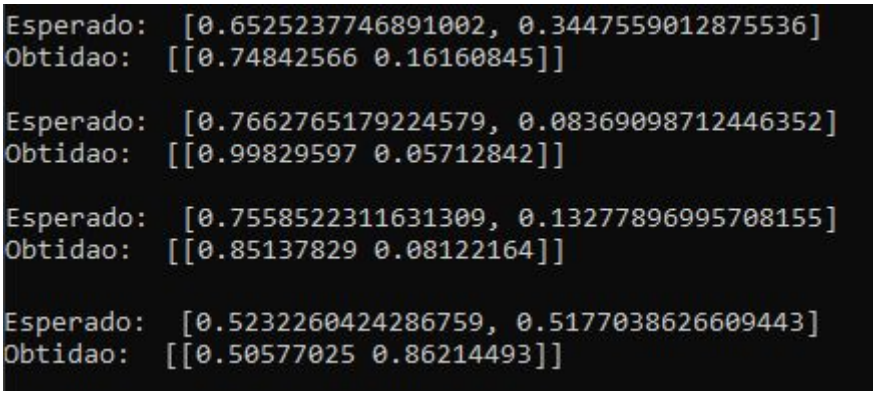
6.

Momentum	0.8
Ciclos	10000
Velocidade aprendizado	0.6

Resultados	Erro1	Erro 2
Teste 1	1.3 x 10^-5	2.6 x 10^-2
Teste 2	1.3 x 10^-5	1.7 x 10^-7
Teste 3	1.15x10^-19	2 x 10^-6

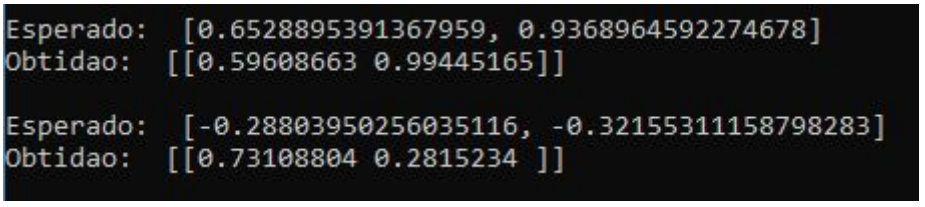
Conclusões aproximação

Após a execução dos testes, notou-se que a rede neural possui uma facilidade maior em encontrar os valores da primeira variável (Latitude), possuindo um erro muito menor se comparado a segunda. Ou seja, a convergência da latitude é maior do que na longitude.
Como pode ser visto na imagem abaixo, o resultado obtido (ainda normalizado) da rede é mais acurado na primeira variável do que na segunda.



(Imagem 4)

Um comportamento anormal da rede foi constatado, a mesma não consegue tratar números negativos. Ao contrário do que acontece com os outros valores, quando há um valor negativo, a efetividade da rede cai consideravelmente.



(Imagem 5)

Conclusão

Após a implementação e testes do algoritmo de MLP, foi possível perceber como uma boa escolha para a base de treinamento e testes influência nos resultados, assim como um estudo mais refinado sobre a quantidade de interações, *momentum* e *learning rate* é importante para um melhor desempenho, tanto para resultados quanto de tempo, do algoritmo.

Foi possível notar também que, assim como dito em aula, treinar a rede excessivamente pode trazer malefícios (como pode ser visto nos resultados onde houve muitas interações).

Em suma, foi possível observar alguns dos conceitos vistos em sala de aula sendo utilizados na prática, além de ter encarado alguns desafios na implementação do algoritmo (total mudança do algoritmo entregue, que fazia todas as operações entre matrizes “na mão”, não possuía BIAS e nem *momentum*).

Todo o código está disponível no *github*, pelo endereço: https://github.com/abe2602/T1-RedesNeurais . Quaisquer dúvida sobre o relatório, implementação ou código, por favor, entrar em contato pelo *email* institucional: bruno.abe@usp.br .