CENTRO UNIVERSITÁRIO FEI

VICTOR BIAZON

RA: 119.115-4

RELATÓRIO IV – TÓPICOS ESPECIAIS DE APRENDIZAGEM
MULTI LAYER PERCEPTRON

SÃO BERNARDO DO CAMPO

2019

Sumário:

1. Objetivo

Implementar o multy layer perceptron e testá-lo para o dataset Iris de Fisher e outro dataset qualquer da UCI.

2. Teoria

Multy Layer Perceptron

Segundo Goodfellow(2016) o multi layer perceptron permite o aprendizado de distribuições não lineares onde ao se adicionar neurônios e camadas ocultas pode-se aumentar o grau de capacidade de aprendizado de diferentes padrões pelo MLP. O MLP no entanto pode-ser suscetível a over fitting se não forem tomadas as devidar proporções de números de camadas e neurônios, sendo que não existe um estudo analítico para determinar a correta quantidade destes elementos. Hoje, a quantidade de neurônios e camadas escondidas é regida de forma empírica e experimental, não tendo fórmula rígida para definí-la.

O MLP é uma associação de perceptrons, sendo assim tem em sua base a mesma matemática para propagação de saídas a partir de entradas processadas por pesos. No entanto o MLP precisa de um artifício para propagação do erro para as camadas escondidas que é o algoritmo chamado Backward Propagation. Este algoritmo é baseado em propagar o erro da sáida comparando a saída da rede com a saída esperada e diferenciando a função de ativação da rede naquele ponto, e para tal é necessário que o as funções de ativação utilizadas em um MLP sejam diferenciáveis em todos os pontos. Alguns exemplos de funções de ativação são a ReLU, a sigmoide e a softmax.

Para o treinamento do MLP é utilizado primeiro uma fase de forward propagation onde a rede propaga a entrada até a saída e em seguida compara a saída real com a esperada. Com o erro calculado se dá início ao backward propagation que propaga os erros até a entrada calculando a correção necessária nos pesos de cada neurônio. Este aprendizado pode ser feito de modo incremental, ou seja, entrada por entrada, ou o batelada, onde se atualizaos pesos com a média das alterações baseadas nos dados.

3. Implementação

Para implementação foram criadas as seguintes funções e classe:

```
# escala as unidades
X = X/np.amax(X, axis=0)
y = y

class Neural_Network(object):
  def __init__(self):
    #parametros da rede
    self.inputSize = 4
    self.outputSize = 3
    self.hiddenSize = 5
    self.hiddenLayers = 1
    self.Ni = 0.1

    #Inicializa a rede com pesos aleatórios baseados no numero de entradas, camadas oculta, numero de neuronios e saidas
    self.W = [0]*(self.hiddenLayers+1)
    self.W[0] = np.random.randn(self.inputSize, self.hiddenSize)
    for i in range(1, self.hiddenLayers):
        self.W[i] = np.random.randn(self.hiddenSize, self.hiddenSize)
    self.W[self.hiddenLayers] = np.random.randn(self.hiddenSize, self.outputSize)


  def forward(self, X):
    #propaga a entrada para frente na rede.
    self.net = [0]*(len(self.W))
    self.fnet = [0]*(len(self.W))
    self.net[0] = np.dot(X, self.W[0])
    self.fnet[0] = self.sigmoid(self.net[0])
    for k in range(1, len(self.net)):
        self.net[k] = np.dot(self.fnet[k-1], self.W[k])
        self.fnet[k] = self.sigmoid(self.net[k])
```

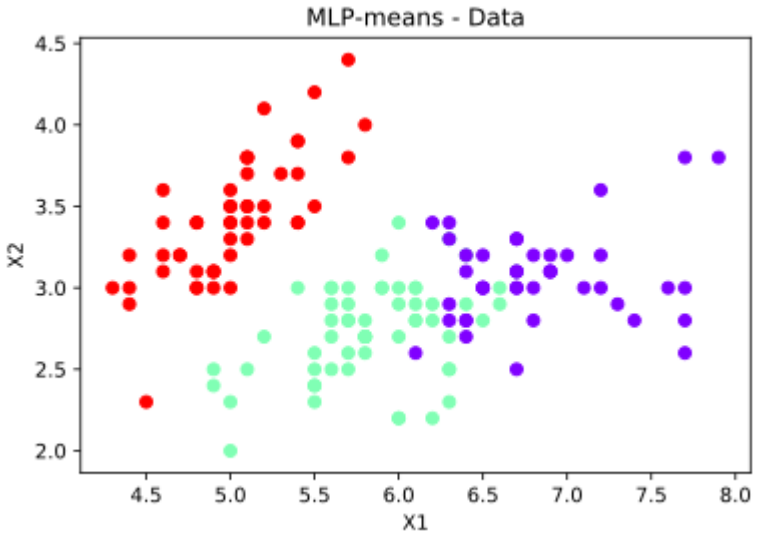
```
    return self.fnet[k]

  def sigmoid(self, s):
    # função de ativação
    return 1/(1+np.exp(-s))

  def sigmoidDer(self, s):
    #derivada da sigmoide
    return s * (1 - s)

  def backward(self, X, y, o):
    # propaga os erros paratrás na rede
    self.o_error = (y - o)*self.sigmoidDer(o) # aplica a diferenciaçao da
sigmoide no erro
    self.error = [0]*(len(self.fnet))
    self.error[len(self.error)-1] = (y - o)*self.sigmoidDer(o)


    for j in range(self.hiddenLayers-1, -1, -1): #calcula a propagação do
erro para cada camada a partir da anterior
        self.error[j] = self.Ni * self.error[j+1].dot(self.W[len(self.W)-
(self.hiddenLayers-j)].T) * self.sigmoidDer(self.fnet[j])


    self.W[0] += X.T.dot(self.error[0]) # ajusta o peso da primeira camada
de acordo com a entrada
    for n in range(1, len(self.W)):
        self.W[n] += self.fnet[n-1].T.dot(self.error[n]) # ajusta os pesos
de cada camada de acordo com o erro propagado


  def train (self, X, y):
    o = self.forward(X)
    self.backward(X, y, o)
```

```
NN = Neural_Network()
for i in range(500000): # treina a rede i vezes
  NN.train(X, y)

print("Input: " + str(X) )
print("Output Real: \n" + str(y) )
print("Output Predita: \n" + str(NN.forward(X)))
print("Erro médio: \n" + str(np.mean(np.square(y - NN.forward(X))))) # mean
sum squared loss
print("\n")

y_pred = NN.forward(X)

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y, y_pred)
```

4. Resultados

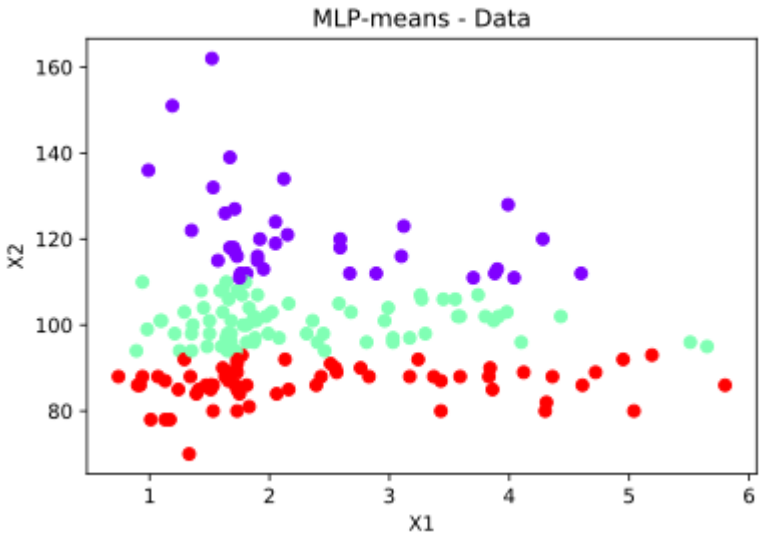
Para se testar o multi layer perceptron foi utilizado o treinamento no dataset IRIS de Fisher, no dataset Wine da UCI e no dataset .....

Para o Iris de Fisher o a classificação das flores resultou na seguinte distribuição:



Pode-se notar uma certa intersecção dos pontos roxos que representam a versicolor com os verdes que são da virginica, porém isto se dá devido a representação em 2D não poder representar corretamente as 4 dimensões consideradas na classificação. Para esta classificação o MLP teve 98,7% de acurácia a partir da matriz de confusão. Com 5 neurônios em uma única camada escondida.

Para o dataset Wine da UCI a classificação resultou na seguinte distribuição:



A divisão teve 97,3% de acurácia para treinamento com nove neurônios na camada escondida, apenas uma camada escondida.

5. Conclusão

Com estes experimentos pôde-se comprovar a eficácia da classificação e generalização do MLP. O MLP se mostrou versátil na identificação de não linearidades a partir do treinamento supervisionado da rede e da aplicação do backward propagation com descida de gradiente. Devido a possibilidade de overfitting deve-se atentar ao número de camadas escondidas e neurônios em cada camada. O treinamento foi realizado com aprendizado por batelada utilizando todas os pesos pela média da variação. É um procedimento que leva mais tempo para realizar, mas que tem mais estabilidade na direção do gradiente.

Referências bibliográficas

[1] Goodfellow, Ian et al , 2016, Deep Learning, MIT Press, disponível em: < http://www.deeplearningbook.org > Acesso em: 10/12/2019 21:08