Centro Universitário FEI

***Perceptron Multicamadas***

**Aula 7 - Exercícios**

Disciplina:

Tópicos Especiais em Aprendizagem (Prof. Reinaldo A. C. Bianchi)

Aluno:

Claudio Aparecido Borges Junior (RA 120122-7)

São Bernardo do Campo

2o semestre / 2020

**EXERCÍCIO 1:** *Implemente uma Rede Neural Artificial do tipo Multilayer Perceptron, implementando o algoritmo de Back-propagation. Teste o algoritmo usando a base de dados de IRIS. Teste diferentes números de camadas escondidas, e diferentes números de neurônios nas camadas escondidas****.***

A implantação foi realizada utilizando a linguagem Python versão 3.6, com função de entrada ex1\_iris\_3\_classes() presente no Anexo 1. A classe MLP foi criada para encapsular as ações necessárias para treinar e classificar utilizando neurônios perceptron multicamadas através das funções fit e predict. Devido a possibilidade de existir mais do que um processador na camada de saída com nível diferente de 0, foi adicionada uma camada de softmax em que o processor de maior valor é selecionado como ligado e todos os outros como desligados.

Os hiperparâmetros configuráveis são o numero de camadas escondidas e a quantidade de processadores nessas camadas, a tolerância do erro durante a fase de treinamento, o número máximo de iterações, e a taxa de aprendizado. Além disso, essa classe utiliza a função sigmoide como ativação e também utiliza aprendizagem em lote (batch).

A métrica de qualidade foi realizada através da acurácia e medida F1 sendo que foi realizada uma validação cruzada de tamanho 10 em que 9/10 dos exemplos foram utilizados para treinamento e 1/10 para teste, repetindo 10 vezes sem sobreposição. Ao término da validação cruzada a média e os valores máximos e mínimos são apresentados.

A tabela abaixo exibi a medida F1 média, máxima e mínima para algumas configurações de camadas escondidas e também variando a número máximo de iterações:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **1e3 iterações** | | | | | | | |
| **Camadas escondidas** | **(1,)** | **(4,)** | **(8,)** | **(40,)** | **(4, 4,)** | **(20, 20)** | **(4,4,4)** |
| **Mean** | 0,4267 | 0,6733 | 0,8267 | 0,9467 | 0,5667 | 0,9667 | 0,5133 |
| **Max** | 0,8000 | 1,0000 | 1,0000 | 1,0000 | 0,8000 | 1,0000 | 0,9333 |
| **Min** | 0,1333 | 0,1333 | 0,4667 | 0,8667 | 0,8000 | 0,8667 | 0,9333 |
|  |  |  |  |  |  |  |  |
| **1e4 iterações** | | | | | | | |
| **Camadas encondidas** | **(1,)** | **(4,)** | **(8,)** | **(40,)** | **(4, 4,)** | **(20, 20)** | **(4,4,4)** |
| **Mean** | 0,4533 | 0,8733 | 0,9667 | 0,9133 | 0,8467 | 0,9667 | 0,7933 |
| **Max** | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| **Min** | 0,1333 | 0,1333 | 0,8667 | 0,5333 | 0,1333 | 0,8667 | 0,1333 |
|  |  |  |  |  |  |  |  |
| **1e5 iterações** | | | | | | | |
| **Camadas encondidas** | **(1,)** | **(4,)** | **(8,)** | **(40,)** | **(4, 4,)** | **(20, 20)** | **(4,4,4)** |
| **Mean** | 0,8333 | 0,9667 | 0,9667 | 0,9667 | 0,9667 | 0,9667 | 0,9000 |
| **Max** | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| **Min** | 0,3333 | 0,8667 | 0,8667 | 0,8667 | 0,8667 | 0,8667 | 0,2667 |

É possível notar que com 1 camada escondida com 4 processadores possuí a média F1 igual a outras configurações que utilizam mais camadas ou processadores. É interessante notar também que o desempenho de 3 camadas escondidas cada qual com 4 processadores possui um desempenho inferior que 1 camada com 4 processados na configuração apresentada. O número máximo de interações é decisivo na medida de qualidade dos processados.

**EXERCÍCIO 2:** *Teste em mais 2 problemas da base de dados de ML*

**Conjunto de dados “Breast Cancer Wisconsin”**: <http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29>

O conjunto de dados sobre câncer de mama possui 10 atributos e classificações como câncer maligno ou benigno. Foi utilizado o MLP com 2 camadas escondidas e 100 processadores em cada camada além de uma taxa de aprendizado de 1e-3. Assim como o conjunto de dados da íris, foi utilizado a medida F1 e validação cruzada, sendo o resultado:

|  |  |
| --- | --- |
| **Mean** | 0,8750 |
| **Max** | 0,9649 |
| **Min** | 0,5263 |

Uma outra configuração, com 1 camada escondidas de 100 processadores foi executada, obtendo o resultado:

|  |  |
| --- | --- |
| **Mean** | 0,9030 |
| **Max** | 0,9649 |
| **Min** | 0,7143 |

Também foi avaliado com 1 camada escondida de 200 processadores, obtendo o resultado:

|  |  |
| --- | --- |
| **Mean** | 0,7820 |
| **Max** | 0,9649 |
| **Min** | 0,2446 |

Desse modo, não foi observado uma relação direta entre o número de processadores ou quantidades de processadores com melhoras na qualidade F1.

**Conjunto de dados “Glass Identification”**: <http://archive.ics.uci.edu/ml/datasets/Glass+Identification>

Conjunto de dados utilizado para treinamento de classificação de tipos de vidros em 7 categorias motivados por investigações criminológicas. Foi utilizado o MLP com 1 camada escondidas com 50 processadores em cada camada além de uma taxa de aprendizado de 1e-3. Assim como o conjunto de dados da íris, foi utilizado a medida F1 e validação cruzada, sendo o resultado:

|  |  |
| --- | --- |
| **Mean** | 0,2983 |
| **Max** | 0,4545 |
| **Min** | 0,1429 |

**ANEXO I – CÓDIGO FONTE**

import numpy as np  
  
  
class MLP:  
 *""" Define a MultiLayer Perceptron  
 A set of perceptrons arranged in multiple layers with at least 1 input  
 layer, 1 hidden layer and 1 output layer. MLP utilizes a supervised  
 learning technique called backpropagation for training.  
 """* def \_\_init\_\_(self,  
 hidden\_layer\_sizes=(100,),  
 learning\_rate\_init=0.001,  
 tolerance=1e-3,  
 max\_steps=250):  
 self.hidden\_layer\_sizes = hidden\_layer\_sizes  
 self.learning\_rate = learning\_rate\_init  
 self.tolerance = tolerance  
 self.max\_steps = max\_steps  
 self.weights = None  
 self.biases = None  
  
 # Define the activation function and the derivate function  
 self.activation\_f = lambda x: 1 / (1 + np.exp(-x))  
 self.d\_activation\_f = \  
 lambda x: self.activation\_f(x) \* (1 - self.activation\_f(x))  
  
 def init\_processors(self, x, y):  
 # Initialize the fit processing  
 assert x.ndim == 2, 'x is not 2d'  
 n\_features = x.shape[1]  
 assert y.ndim == 2, 'y is not 2d'  
 n\_classes = y.shape[1]  
 # Build the weights matrix using the number of features (x), hidden  
 # layers, and number of classes (y)  
 layers = (n\_features,) + self.hidden\_layer\_sizes + (n\_classes,)  
 # Create the list of matrixes for weights and biases. Each position is  
 # a ANN layer  
 self.weights = [np.random.randn(layers[i], layers[i + 1])  
 for i in range(len(layers) - 1)]  
 self.biases = [np.random.randn(1, layers[i + 1])  
 for i in range(len(layers) - 1)]  
  
 def \_feed\_forward(self, x):  
 # Process the signal forwardly  
 signal = x  
 # Nets are the processor net value, i.e. before the activation  
 nets = []  
 # Activation is the output of the processor  
 activations = [signal]  
 # Each each iteration is a ANN layer  
 for i, layer\_weights in enumerate(self.weights):  
 nets.append(signal.dot(layer\_weights) + self.biases[i])  
 signal = self.activation\_f(nets[-1])  
 activations.append(signal)  
 return nets, activations  
  
 def \_back\_propagation(self, y, nets, activations):  
 errors = [None] \* len(self.weights)  
 # Process the last layer error  
 errors[-1] = (y - activations[-1]) \* self.d\_activation\_f(nets[-1])  
 # Perform BackPropagation  
 for i in reversed(range(len(errors) - 1)):  
 errors[i] = self.weights[i + 1].dot(errors[i + 1].T).T \  
 \* self.d\_activation\_f(nets[i])  
  
 db = [e.T.dot(np.ones((e.shape[0], 1))) for e in errors]  
 dw = [e.T.dot(s) for e, s in zip(errors, activations)]  
 # Return both the weights and bias delta  
 return dw, db  
  
 def fit(self, x, y):  
 *"""  
 Fit the model to the input x and target y* ***:param*** *x: the input data; an array-like of shape (n\_samples, n\_features)* ***:param*** *y: the target values; ndarray of shape (n\_samples,)* ***:return****: returns a trained MLP model  
 """* self.init\_processors(x, y)  
  
 it = 0  
 loss = self.tolerance + 1  
 # Keep training  
 while (it < self.max\_steps) and (loss > self.tolerance):  
 nets, activations = self.\_feed\_forward(x)  
 dw, db = self.\_back\_propagation(y, nets, activations)  
 # Adjust the weights  
 self.biases = [b + self.learning\_rate \* d.T  
 for b, d in zip(self.biases, db)]  
 self.weights = [w + self.learning\_rate \* d.T  
 for w, d in zip(self.weights, dw)]  
 loss = np.linalg.norm(activations[-1] - y)  
 #print("loss = {}".format(loss))  
 it += 1  
  
 if loss > self.tolerance:  
 # if it hasn't converged, print a message  
 #print('Train timeout')  
 pass  
  
 def predict(self, x, softmax=True):  
 *"""  
 Predict using the MLP* ***:param*** *x: the input data; an array-like of shape (n\_samples, n\_features)* ***:param*** *softmax: force only one processor per sample* ***:return****: ndarray of shape (n\_samples, n\_outputs)  
 """* \_, activations = self.\_feed\_forward(x)  
 prediction = activations[-1]  
 if not softmax:  
 return prediction  
 prediction\_max = np.zeros(prediction.shape)  
 for i, p in enumerate(np.argmax(prediction, axis=1)):  
 prediction\_max[i][p] = 1  
 return prediction\_max  
  
  
def score(ys, y\_preds):  
 # Return the precision, recall, accuracy and F1 measures  
 tp, tn, fp, fn = 0, 0, 0, 0  
 for y, y\_pred in zip(ys, y\_preds):  
 for e\_y, e\_y\_pred in zip(y, y\_pred):  
 tp += e\_y == 1 and e\_y == e\_y\_pred  
 tn += e\_y == 0 and e\_y == e\_y\_pred  
 fp += e\_y == 0 and e\_y != e\_y\_pred  
 fn += e\_y == 1 and e\_y != e\_y\_pred  
 precision = tp / (tp + fp)  
 recall = tp / (tp + fn)  
 accuracy = (tp + tn) / (tp + fp + tn + fn)  
 f1 = 2 \* precision \* recall / (precision + recall)  
 return precision, recall, accuracy, f1  
  
  
def cross\_validation(x, y, k=10):  
 # Split the dataset into multiple foldings of train and test  
 ranges = np.linspace(0, len(x), num=(k + 1), dtype=int)  
 x\_groups = [x[ranges[i]:ranges[i + 1]] for i in range(len(ranges) - 1)]  
 y\_groups = [y[ranges[i]:ranges[i + 1]] for i in range(len(ranges) - 1)]  
 for i in range(k):  
 x\_train = np.concatenate(x\_groups[:i] + x\_groups[i + 1:])  
 y\_train = np.concatenate(y\_groups[:i] + y\_groups[i + 1:])  
 x\_test = x\_groups[i]  
 y\_test = y\_groups[i]  
 yield x\_train, y\_train, x\_test, y\_test,  
  
  
def exi\_iris\_3\_classes():  
 # Read the iris dataset  
 data = np.genfromtxt('datasets/iris.data', delimiter=',',  
 dtype=None,  
 encoding=None)  
  
 # Suffle the data to decrease the training bias  
 np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the  
 # output samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if i < len(row) - 1] for row in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == len(row) - 1] for row in  
 data])  
  
 # Use values instead of class names  
 class\_map = {  
 'Iris-virginica': np.asarray([1, 0, 0]),  
 'Iris-setosa': np.asarray([0, 1, 0]),  
 'Iris-versicolor': np.asarray([0, 0, 1]),  
 }  
 y\_adj = np.asarray([class\_map[y\_sample[0]] for y\_sample in y])  
  
 hidden\_layers\_setups = [(1,), (4,), (8,), (40,), (4, 4), (20, 20,),  
 (4, 4, 4,), ]  
  
 for max\_steps in [1e3, 1e4, 1e5]:  
 print("---------------------")  
 print("Max Steps: ", max\_steps)  
 for setup in hidden\_layers\_setups:  
 print("Hiddent Layer setups: ", setup)  
 scores = []  
 for x\_train, y\_train, x\_test, y\_test in cross\_validation(x, y\_adj):  
 mlp = MLP(hidden\_layer\_sizes=setup, max\_steps=max\_steps)  
 mlp.fit(x\_train, y\_train)  
 y\_preds = mlp.predict(x\_test)  
  
 # Adjust prediction to have just one processor activated per time  
 # using a softmax  
 y\_pred\_adj = np.zeros(y\_preds.shape)  
 for i, y\_pred in enumerate(np.argmax(y\_preds, axis=1)):  
 y\_pred\_adj[i][y\_pred] = 1  
  
 # print("Weights:", mlp.weights)  
 # print("Biases:", mlp.biases)  
 scores.append([\*score(y\_test, y\_pred\_adj)])  
 # print(  
 # "Score: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 # .format(\*score(y\_test, y\_pred\_adj)))  
 scores = np.array(scores)  
 means = scores.mean(axis=0)  
 maxs = scores.max(axis=0)  
 mins = scores.min(axis=0)  
 print("Score")  
 print(  
 " mean: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*means))  
 print(  
 " max: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*maxs))  
 print(  
 " min: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*mins))  
  
  
def breast\_cancer():  
 data = np.genfromtxt('datasets/wdbc.data', delimiter=',', dtype=None,  
 encoding=None)  
  
 # Suffle the data to decrease the bias  
 # np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the output  
 # samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if i >= 2] for row in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == 1] for row in  
 data])  
  
 # Normalize X to prevent sigmoid overflow  
 x\_adj = x / x.max() - x.min()  
 # Use values instead of class names  
 class\_map = {  
 'B': np.asarray([1, 0]),  
 'M': np.asarray([0, 1]),  
 }  
 y\_adj = np.asarray([class\_map[y\_sample[0]] for y\_sample in y])  
  
 scores = []  
 for x\_train, y\_train, x\_test, y\_test in cross\_validation(x\_adj, y\_adj):  
 mlp = MLP(hidden\_layer\_sizes=(200,), max\_steps=1e4,  
 learning\_rate\_init=1e-3)  
 mlp.fit(x\_train, y\_train)  
 y\_preds = mlp.predict(x\_test)  
  
 # print("Weights:", mlp.weights)  
 # print("Biases:", mlp.biases)  
 scores.append([\*score(y\_test, y\_preds)])  
 print(  
 "Score: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*score(y\_test, y\_preds)))  
 scores = np.array(scores)  
 means = scores.mean(axis=0)  
 maxs = scores.max(axis=0)  
 mins = scores.min(axis=0)  
 print("Score")  
 print(  
 " mean: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*means))  
 print(  
 " max: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*maxs))  
 print(  
 " min: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*mins))  
  
  
def glass():  
 data = np.genfromtxt('datasets/glass.data', delimiter=',', dtype=None,  
 encoding=None)  
  
 # Suffle the data to decrease the bias  
 np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the output  
 # samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if 0 < i < len(row) - 1] for row  
 in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == len(row) - 1] for row in  
 data])  
  
 # Normalize X to prevent sigmoid overflow  
 x\_adj = x / x.max() - x.min()  
 class\_map = {  
 1: np.asarray([1, 0, 0, 0, 0, 0, 0]),  
 2: np.asarray([0, 1, 0, 0, 0, 0, 0]),  
 3: np.asarray([0, 0, 1, 0, 0, 0, 0]),  
 4: np.asarray([0, 0, 0, 1, 0, 0, 0]),  
 5: np.asarray([0, 0, 0, 0, 1, 0, 0]),  
 6: np.asarray([0, 0, 0, 0, 0, 1, 0]),  
 7: np.asarray([0, 0, 0, 0, 0, 0, 1]),  
 }  
 y\_adj = np.asarray([class\_map[y\_sample[0]] for y\_sample in y])  
  
 scores = []  
 for x\_train, y\_train, x\_test, y\_test in cross\_validation(x\_adj, y\_adj):  
 mlp = MLP(hidden\_layer\_sizes=(50, ), max\_steps=1e4,  
 learning\_rate\_init=1e-3)  
 mlp.fit(x\_train, y\_train)  
 y\_preds = mlp.predict(x\_test)  
  
 # print("Weights:", mlp.weights)  
 # print("Biases:", mlp.biases)  
 scores.append([\*score(y\_test, y\_preds)])  
 print(  
 "Score: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*score(y\_test, y\_preds)))  
 scores = np.array(scores)  
 means = scores.mean(axis=0)  
 maxs = scores.max(axis=0)  
 mins = scores.min(axis=0)  
 print("Score")  
 print(  
 " mean: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*means))  
 print(  
 " max: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*maxs))  
 print(  
 " min: precision={:.4f}, recall={:.4f}, accuracy={:.4f}, F1={:.4f}"  
 .format(\*mins))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 exi\_iris\_3\_classes()  
 breast\_cancer()  
 glass()