Universidade Federal do Agreste de Pernambuco

Bacharelado em Ciências da Computação

Prof. Tiago Buarque A. de Carvalho

Aprendizagem de Máquina:

Exercícios sobre Redes Neurais

Aluno: Vinícius Santos de Almeida

- 1. a)
    - Após gerar o algorítmo para treinar um neurônio para reconhecer a classe iris-setosa da base íris, obtive o seguinte vetor de pesos:

      `[0.6499999999999995, 2.05, -2.6000000000000005, -1.0999999999999999]`

    - A saída foi:

```
Epoca #1
Total de erros: 2
Epoca #2
Total de erros: 2
Epoca #3
Total de erros: 1
Epoca #4
Total de erros: 0
Histórico de pesos:
[0, 0, 0, 0]
[2.55, 1.75, 0.7, 0.1]
[-0.9500000000000002, 0.1499999999999999, -1.6500000000000001,
-0.6]
[1.5999999999999996, 1.9, -0.9500000000000002, -0.5]
[-1.9000000000000004, 0.2999999999999998, -3.3000000000000003,
-1.2]
Peso definido:
[0.6499999999999995, 2.05, -2.6000000000000005,
-1.0999999999999999]
Número de erros no teste: 0
```

    - Note que, no final da saída, já mostrei o resultado do erro rodando na base de teste, usando os pesos descobertos na fase de treino, mostrado acima, fiz isso usando o método `test()` que criei na classe `IrisNeuron`.

    - Para realização do neurônio criei a função `split_train_test()`, segue abaixo sua implementação:

```
def split_train_test(data: list, train_size: int, by_columns:
list):
    X_train = []
    y_train = []
    X_test = []
    y_test = []
    count_columns = {col: 0 for col in by_columns}
    for row in data:
        for k, v in row.items():
            for col in by_columns:
                if count_columns[col] < train_size \
                        and k == col and v:
                    X_train.append([float(v) for k, v in
row.items() if k not in by_columns])
                    y_train.append({k:int(v) for k, v in
row.items() if k in by_columns})
                    count_columns[col] += 1
                elif count_columns[col] >= train_size \
                    and k == col and v:
                    X_test.append([float(v) for k, v in
row.items() if k not in by_columns])
                    y_test.append({k: int(v) for k, v in
row.items() if k in by_columns})
    y_train = {col: [v[col] for v in y_train] for col in
by_columns}
    y_test= {col: [v[col] for v in y_test] for col in by_columns}
    return X_train, y_train, X_test, y_test
```

- As demais funções necessárias fazem parte da class `IrisNeuron` que criei:

```
from utils import random_zero_one

class IrisNeuron():

    def __init__(self, N: float = 0.5, max_error: int = 0):
        self.weights, self.weights_hist, self.X, self.Y,
self.train_class = [], [], None, None, None
        self.N = N
        self.max_error = max_error
        print()

    def fit(self, X: list[list], y: list) -> None:
        self.X = X
        self.y = y

    def train(self, train_class: str) -> list:
        self.train_class = train_class
        total_errors = 1
        epoch = 0
        while(total_errors > self.max_error):
```

```
            epoch += 1
            print(f'Epoca #{epoch}')
            total_errors = 0
            for i, (x, y) in enumerate(zip(self.X,
self.y[train_class])):
                delta = y - IrisNeuron.f(x, self.weights)
                error = delta * delta
                total_errors += error
                if error > 0:
                    self.weights_hist.append(self.weights)
                    for j, (w_j, x_j) in
enumerate(zip(self.weights, x)):
                        self.weights[j] = w_j + self.N * x_j *
delta
            print(f'Total de erros: {total_errors}')
        return self.weights

    def set_weights(self, w: list) -> None:
        self.weights = w

    def test(self, X: list[list], y: list) -> int:
        total_errors = 0
        for x, y in zip(X, y[self.train_class]):
            delta = y - IrisNeuron.f(x, self.weights)
            error = delta * delta
            total_errors += error
        return total_errors

    def generate_weights(self, n_columns: int, random: bool =
False) -> list:
        weights = [random_zero_one() if random else 0 for i in
range(0, n_columns)]
        self.weights = weights
        return self.weights

    @staticmethod
    def f(x: list, w: list) -> int:
        v = 0
        for x_i, w_i in zip(x, w):
            v += x_i * w_i
        return 1 if v > 0 else 0
```

- Abaixo, segue o main da questão:

```
from utils import load_data, split_train_test

from non_binary_categorizer import NonBinaryConverter

def main():
# Load data from file
data, _ = load_data('./data/iris.csv', ',')
```

```
# Convert categoric attributes to numeric
nb_converter = NonBinaryConverter(data, ['iris'])
data, columns = nb_converter.convert_data(),
nb_converter.generate_columns()

# Select sample from data
classes = ['iris_setosa', 'iris_versicolor', 'iris_virginica']
X_train, y_train, X_test, y_test = split_train_test(data,
train_size=25, by_columns=classes)

# Init iris train neuron
neuron = IrisNeuron()

# Add train X and y to neuron
neuron.fit(X_train, y_train)

# Generate weights list with n_columns, all values will be 0
neuron.generate_weights(n_columns=len(X_train[0]), random=False)

# Train iris_setosa
neuron.train(train_class='iris_setosa')

# Show training iris_setosa
print(f'Histórico de pesos: ')
[print(w) for w in neuron.weights_hist]
print(f'Peso definido: ')
print(neuron.weights)

# Test iris_setosa
errors = neuron.test(X_test, y_test)
print(f'Número de erros no teste: {errors}')
```

* A função `load_data()` já foi apresentada nas semanas anteriores.

** A classe `NonBinaryConverter` é reaproveitada da semana passada, e é usada no `main()` para converter a coluna "classe", que é categórica, em numérica, gerando três colunas com valores 0 e 1, que possibilitam o uso na rede neural.

- 1. b)
    - Foram feitas algumas adaptações no código da questão anterior para limitar o algoritmo à época 100 invés do erro em 0.
    - A saída rodou as 100 épocas definidas, e não encontrou 0 antes que terminasse, segue abaixo a saída resumida:

```
Epoca #1
Total de erros: 1
Epoca #2
Total de erros: 3
```

```
Epoca #3
Total de erros: 3
...
Epoca #98
Total de erros: 3
Epoca #99
Total de erros: 3
Epoca #100
Total de erros: 2
Histórico de pesos:
[0, 0, 0, 0]
[3.15, 1.65, 3.0, 1.25]
[0.6000000000000001, -0.10000000000000009, 2.3, 1.15]
...
[-29.500000000000018, -15.700000000000001, 30.000000000000064,
26.50000000000003]
[-26.60000000000002, -14.350000000000001, 32.55000000000006,
27.450000000000028]
[-29.55000000000002, -15.950000000000001, 30.150000000000063,
26.55000000000003]
Peso definido:
[-26.65000000000002, -14.600000000000001, 32.70000000000006,
27.50000000000003]
Número de erros no teste: 5
```

- Note que, no final do treinamento, ao realizar o teste, houve um total de 5 erros.
- Implementação
  - Classe `IrisNeuron` atualizada:

```
class IrisNeuron():

    def __init__(self, N: float = 0.5, max_error: int = 0,
limit_epoch: int = None):
        self.weights, self.weights_hist, self.X, self.Y,
self.train_class = [], [], None, None, None
        self.N = N
        self.max_error = max_error
        self.limit_epoch = limit_epoch
        print()

    def fit(self, X: list[list], y: list) -> None:
        self.X = X
        self.y = y

    def train(self, train_class: str) -> list:
        self.train_class = train_class
        total_errors = 1
        epoch = 0
        while total_errors > self.max_error \
            or epoch < self.limit_epoch:
```

```
            epoch += 1
            print(f'Epoca #{epoch}')
            total_errors = 0
            for i, (x, y) in enumerate(zip(self.X,
self.y[train_class])):
                delta = y - IrisNeuron.f(x, self.weights)
                error = delta * delta
                total_errors += error
                if error > 0:
                    self.weights_hist.append([w for w in
self.weights])
                    for j, (w_j, x_j) in
enumerate(zip(self.weights, x)):
                        self.weights[j] = w_j + self.N * x_j
* delta
            print(f'Total de erros: {total_errors}')
        return self.weights

    def test(self, X: list[list], y: list) -> int:
        total_errors = 0
        for x, y in zip(X, y[self.train_class]):
            delta = y - IrisNeuron.f(x, self.weights)
            error = delta * delta
            total_errors += error
        return total_errors

    def set_weights(self, w: list) -> None:
        self.weights = w

    def generate_weights(self, n_columns: int, random: bool
= False) -> list:
        weights = [random_zero_one() if random else 0 for i
in range(0, n_columns)]
        self.weights = weights
        return self.weights

    @staticmethod
    def f(x: list, w: list) -> int:
        v = 0
        for x_i, w_i in zip(x, w):
            v += x_i * w_i
        return 1 if v > 0 else 0
```

- `main()` atualizado:

```
def main():
    # Load data from file
    data, _ = load_data('./data/iris.csv', ',')

    # Convert categoric attributes to numeric
    nb_converter = NonBinaryConverter(data, ['iris'])
    data, columns = nb_converter.convert_data(),
```

```
nb_converter.generate_columns()

    # Select sample from data
    classes = ['iris_setosa', 'iris_versicolor',
'iris_virginica']
    X_train, y_train, X_test, y_test =
split_train_test(data, train_size=25, by_columns=classes)

    # Init iris train neuron
    neuron = IrisNeuron(limit_epoch=100)

    # Add train X and y to neuron
    neuron.fit(X_train, y_train)

    # Generate weights list with n_columns, all values will
be 0
    neuron.generate_weights(n_columns=len(X_train[0]),
random=False)

    # Train iris_virginica
    neuron.train(train_class='iris_virginica')

    # Show training iris_virginica
    print(f'Histórico de pesos: ')
    [print(w) for w in neuron.weights_hist]
    print(f'Peso definido: ')
    print(neuron.weights)

    # Test iris_virginica
    errors = neuron.test(X_test, y_test)
    print(f'Número de erros no teste: {errors}')
```

* As demais classes e funções usadas já foram apresentadas.

- 1. c)
    - Para realizar as operações, foi necessário apenas alterar o `main()`, e reusar a classe `IrisNeuron` da questão **1. b.**.
    - Segue abaixo os resultados organizados da saída:
        - Classe: iris_setosa
            - Taxa de aprendizagem: 0.1
                - Peso definido: `[-0.02000000000000024, 0.4999999999999999, -0.4900000000000001, 0.21000000000000008]`
                - Número de erros no teste: 0
            - Taxa de aprendizagem: 1.0
                - Peso definido: `[1.799999999999999, 5.1, -6.6, -2.6999999999999993]`
                - Número de erros no teste: 0 Taxa de aprendizagem: 10.0
                - Peso definido: `[11.599999999999994, 36.1, -51.7, -21.8]`
                - Número de erros no teste: 0
        - Média de erros: 0.0

        - Desvio: 0.0
    - Classe: iris_virginica
        - Taxa de aprendizagem: 0.1
            - Peso definido: `[-5.389999999999992, -3.94, 6.91999999999997, 6.200000000000039]`
            - Número de erros no teste: 5
        - Taxa de aprendizagem: 1.0
            - Peso definido: `[-54.70000000000007, -38.800000000000004, 71.40000000000006, 57.800000000000075]`
            - Número de erros no teste: 6
        - Taxa de aprendizagem: 10.0
            - Peso definido: `[-552.3, -425.2, 741.5, 586.3]`
            - Número de erros no teste: 5
        - Média de erros: 5.333333333333333
        - Desvio: 0.4714045207910317
- A saída completa está anexa em `questao_1_c.log`.
- Segue abaixo o `main()` adaptado para essa questão:

```
def main():
    # Load data from file
    data, _ = load_data('./data/iris.csv', ',')

    # Convert categoric attributes to numeric
    nb_converter = NonBinaryConverter(data, ['iris'])
    data, columns = nb_converter.convert_data(),
nb_converter.generate_columns()

    # Select sample from data
    classes = ['iris_setosa', 'iris_versicolor',
'iris_virginica']
    X_train, y_train, X_test, y_test = split_train_test(data,
train_size=25, by_columns=classes)

    learning_rates = [0.1, 1.0, 10.0]
    train_test_classes = ['iris_setosa', 'iris_virginica']

    for train_test_class in train_test_classes:
        print(f'Classe: {train_test_class}, ', end='')
        errors = np.array([])
        for rate in learning_rates:
            print(f'Taxa de aprendizagem: {rate}')
            # Init iris train neuron
            neuron = IrisNeuron(limit_epoch=30, N=rate)

            # Add train X and y to neuron
            neuron.fit(X_train, y_train)

            # Generate weights list with n_columns, all values
will be 0
            neuron.generate_weights(n_columns=len(X_train[0]),
```

```
random=True)

            # Train
            neuron.train(train_class=train_test_class)

            # Show training
            print(f'Histórico de pesos: ')
            [print(w) for w in neuron.weights_hist]
            print(f'Peso definido: ')
            print(neuron.weights)

            # Test
            error = neuron.test(X_test, y_test)
            print(f'Número de erros no teste: {error}')
            errors = np.append(errors, error)

        print(f'Média de erros: {np.average(errors)}')
        print(f'Desvio: {errors.std()}')
```

- 2. a)
    - Fiz o algoritmo usando a biblioteca Keras como foi sugerido, após algum tempo tentando os parâmetros e tentando entender, consegui uma taxa de acurácia superior a 74%, assim como foi solicitado.
    - Antes de começar, foi necessário um preprocessamento: converter as três classes possíveis da base wine para três colunas binárias, para isso, usei a classe `NonBinaryConverter` usada em questões anteriores, não foi feito uso de biblioteca nesse passo.
        - Aqui também testei converter os valores binários da base de 0 e 1 para 0.1 e 0.9, porém não obtive melhores resultados dessa forma, por isso descartei.
    - Em seguida, usei o `train_test_split()` para separa a base em 50% de treino e 50% de teste.
    - No final, ao realizar rodar o algoritmo com 100 épocas, com entrada dos 13 valores, mais 2 camadas densas, a primeira de 26 neurônios e a camada de saída de 3 neurônios, usando a função de ativação `softmax`, foi possível atingir uma acurácia de **87%** ao avaliar na base teste.
    - Saída:

```
0.8764045238494873
```

    - Implementação:

```
from sklearn.model_selection import train_test_split

from keras.models import Sequential
from keras.layers.core import Dense

from non_binary_categorizer import NonBinaryConverter
from utils import load_data
```

```
data, columns = load_data('./data/wine.csv', ',')
nb_converter = NonBinaryConverter(data, ['class'])
data, columns = nb_converter.convert_data(),
nb_converter.generate_columns()

X = [[float(v) for k, v in row.items() if 'class' not in k] for
row in data]
y = [[int(v) for k, v in row.items() if 'class' in k] for row in
data]

X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.5)

model = Sequential()
model.add(Dense(26, input_dim=13))
model.add(Dense(3, activation="softmax"))
model.compile(loss="categorical_crossentropy", metrics=
['accuracy'])
model.fit(X_train, y_train, epochs=100, batch_size=1)

_, accuracy = model.evaluate(X_test, y_test)
print(accuracy)
```

- 2. b)
  - Após ajustar o código para fazer as 30 repetições de holdout 50/50 foi possível notar que a média é um pouco menor que o resultado obtido na questão acima.
  - Saídas:

```
Desvio padrão: 0.1243315983157791
Média: 0.836329597234726
```

  - Implementação:

```
import numpy as np
from sklearn.model_selection import train_test_split

from keras.models import Sequential
from keras.layers.core import Dense

from non_binary_categorizer import NonBinaryConverter
from utils import load_data

data, columns = load_data('./data/wine.csv', ',')
nb_converter = NonBinaryConverter(data, ['class'])
data, columns = nb_converter.convert_data(),
nb_converter.generate_columns()
```

```
X = [[float(v) for k, v in row.items() if 'class' not in k] for
row in data]
y = [[int(v) for k, v in row.items() if 'class' in k] for row in
data]

acuracias = np.array([])
for i in range(30):
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.5)

model = Sequential()
model.add(Dense(26, input_dim=13))
model.add(Dense(3, activation="softmax"))
model.compile(loss="categorical_crossentropy", metrics=
['accuracy'])
model.fit(X_train, y_train, epochs=100, batch_size=1)

_, accuracy = model.evaluate(X_test, y_test)
print(accuracy)
acuracias = np.append(acuracias, accuracy)

print(f'Desvio padrão: {acuracias.std()}')
print(f'Média: {np.average(acuracias)}')
```

- 3.
    - Essa questão não consegui terminar a tempo, cheguei a tentar simular a mesma situação do exercício, porém como a acurácia estava dando muito abaixo do mostrado no exercício, acabei não conseguindo concluir a análise a tempo, pois também não tive mais tempo para testar mais possibilidades.
    - Segue o código que estava usando (agradeço se puder apontar o problema):

```
import numpy as np
from sklearn.model_selection import train_test_split

from keras.optimizer_v2.adam import Adam
from keras import Input
from keras.models import Sequential
from keras.layers.core import Dense
from keras.layers.core import Activation

from non_binary_categorizer import NonBinaryConverter
from utils import load_data

data, columns = load_data('./data/spiral.csv', ',')

nb_converter = NonBinaryConverter(data, ['class'])
data, columns = nb_converter.convert_data(),
nb_converter.generate_columns()

X = [[float(v) for k, v in row.items() if 'class' not in k] for
```

```
row in data]
y = [[int(v) for k, v in row.items() if 'class' in k] for row in
data]

acuracias = np.array([])
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.7)

model = Sequential()
model.add(Input(shape=(2,)))
model.add(Dense(4))
model.add(Dense(3))
model.add(Activation(activation="softmax"))
opt = Adam(learning_rate=0.3)
model.compile(loss="categorical_crossentropy", metrics=
['accuracy'], optimizer=opt)
model.fit(X_train, y_train, epochs=500, batch_size=1)

_, accuracy = model.evaluate(X_test, y_test)
print(accuracy)
```