Trabalho 5 de Inteligência Artificial – Manha

NOME	MATRICULA
Fabricio Baptista de Castro	0050481821007
Mario Celso Zanin	0050481921023

Código

```
import math
import random

from linear_algebra import dot


def step_function(x):
    return 1 if x >= 0 else 0


def perceptron_output(weights, bias, x):
    """invoca função degrau = step_function"""
    return step_function(dot(weights, x) + bias)


def sigmoid(t):
    return 1 / (1 + math.exp(-t))


def neuron_output(weights, inputs):
    return sigmoid(dot(weights, inputs))


def feed_forward(neural_network, input_vector):
    """manipula uma rede neural(representada como uma lista de lista de pesos)
    e retorna a saída do algoritmo forward-propagating a partir da entrada da
rede"""

    outputs = []
    for layer in neural_network:
        # adiciona bias à entrada
        input_with_bias = input_vector + [1]
        output = [neuron_output(neuron, input_with_bias)    # calcula a saída do
neurônio
                  for neuron in layer]                      # para cada camada
        outputs.append(output)                              # relembrando isto

        # a saída de uma camada de neurônio é a entrada da próxima camada
```

```
        input_vector = output

    return outputs


def backpropagate(network, input_vector, target):
    # feed_forward calcula a saída dos neurônios usando sigmóide
    hidden_outputs, outputs = feed_forward(network, input_vector)
    # 0.5 *alpha* (1 + output) * (1 - output) cálculo de derivada de sigmóide
    output_deltas = [output * (1 - output) * (output - target[i])
                     for i, output in enumerate(outputs)]

    # ajuste dos pesos sinápticos para camadas de saída (network[-1])
    for i, output_neuron in enumerate(network[-1]):
        for j, hidden_output in enumerate(hidden_outputs + [1]):
            output_neuron[j] -= output_deltas[i] * hidden_output

    # retro-programação do erro para camadas intermediárias
    hidden_deltas = [hidden_output * (1 - hidden_output) *
                     dot(output_deltas, [n[i] for n in network[-1]])
                     for i, hidden_output in enumerate(hidden_outputs)]

    # ajuste dos pesos sinápticos para camadas intermediárias (network[0])
    for i, hidden_neuron in enumerate(network[0]):
        for j, input in enumerate(input_vector + [1]):
            hidden_neuron[j] -= hidden_deltas[i] * input


if __name__ == "__main__":

    raw_digits = [
        """ .111.
            1...1
            11111
            1...1
            1...1""",

        """ 11111
            1....
            11111
            1....
            11111""",

        """ 11111
            ..1..
            ..1..
            ..1..
            11111""",

        """ 11111
            1...1
            1...1
            1...1
            11111""",
```

```
        """ 1...1
            1...1
            1...1
            1...1
            11111"""]

    def make_digit(raw_digit):
        return [1 if c == '1' else 0
                for row in raw_digit.split("\n")
                for c in row.strip()]

    inputs = list(map(make_digit, raw_digits))

    targets = [[1 if i == j else 0 for i in range(5)]
               for j in range(5)]

    # pode-se utilizar valores repetidos a partir dos randômicos
    random.seed(0)
    input_size = 25  # dimensões dos vetores relacionados às 10 entradas
    num_hidden = 5  # quantidade de neurônios na camada intermediária
    output_size = 5  # 10 saidas, cada uma relacionada à uma entrada

    # cada neurônio da camada intermediária tem um peso sináptico associado à
entrada
    # e adicionado o peso do bias
    hidden_layer = [[random.random() for __ in range(input_size + 1)]
                    for __ in range(num_hidden)]

    # neurônio de saída tem um peso sináptico associado a cada
    # #neurôio da camada intermediária
    # e adicionado o peso do bias
    output_layer = [[random.random() for __ in range(num_hidden + 1)]
                    for __ in range(output_size)]

    # a rede inicializa com pesos sinápticos randômicos
    network = [hidden_layer, output_layer]

    # 10000 ciclos de treinamento
    for __ in range(10000):  # número de ciclos de treinamento
        for input_vector, target_vector in zip(inputs, targets):
            backpropagate(network, input_vector, target_vector)

    def predict(input):
        return feed_forward(network, input)[-1]

    # TESTE DAS ENTRADAS TREINADAS
    print(f'Entradas Treinadas')
    for i, input in enumerate(inputs):
        outputs = predict(input)
        # resultado dos neurônios de saida com 2 casas decimais
        print(i, [round(p, 2) for p in outputs])

    # TESTE DE DÍGITOS NUMÉRICOS QUE NÃO FORAM TREINADOS
```

```
# TESTE DE DÍGITOS NUMÉRICOS QUE NÃO FORAM TREINADOS
print("""
        .@@@.
        @...@
        @@@@@
        @...@
        @...@
            """)
print([round(x, 2) for x in
       predict([0, 1, 1, 1, 0,      # .@@@.
                1, 0, 0, 0, 1,      # @...@
                1, 1, 1, 1, 1,      # @@@@@
                1, 0, 0, 0, 1,      # @...@
                1, 0, 0, 0, 1])])   # @...@
print("Interprete como variação do dígito A")

print("""
        @@@@@
        @....
        @@@..
        @....
        @@@@@
            """,)
print([round(x, 2) for x in
       predict([1, 1, 1, 1, 1,      # @@@@@
                1, 0, 0, 0, 0,      # @....
                1, 1, 1, 0, 0,      # @@@..
                1, 0, 0, 0, 0,      # @....
                1, 1, 1, 1, 1])])   # @@@@@
print("Interprete como variação do dígito E")

print("""
        .@@@.
        ..@..
        ..@..
        ..@..
        .@@@.
            """,)
print([round(x, 2) for x in
       predict([0, 1, 1, 1, 0,      # .@@@.
                0, 0, 1, 0, 0,      # ..@..
                0, 0, 1, 0, 0,      # ..@..
                0, 0, 1, 0, 0,      # ..@..
                0, 1, 1, 1, 0])])   # .@@@.
print("Interprete como variação do dígito I")

print("""
        .@@@.
        @...@
        @...@
        @...@
        .@@@.
            """)
print([round(x, 2) for x in
```

```
            predict([0, 1, 1, 1, 0,      # .@@@.
                     1, 0, 0, 0, 1,      # @...@
                     1, 0, 0, 0, 1,      # @...@
                     1, 0, 0, 0, 1,      # @...@
                     0, 1, 1, 1, 0])])   # .@@@.
    print("Interprete como possível variação do dígito U")
    print("""
            @...@
            @...@
            @...@
            @...@
            .@@@.
                """)
    print([round(x, 2) for x in
           predict([1, 0, 0, 0, 1,      # @...@
                    1, 0, 0, 0, 1,      # @...@
                    1, 0, 0, 0, 1,      # @...@
                    1, 0, 0, 0, 1,      # @...@
                    0, 1, 1, 1, 0])])   # .@@@.
    print("Interprete como possível variação do dígito U")
```

Console

```
PS D:\workspace\IA> &
C:/Users/bapti/AppData/Local/Programs/Python/Python310/python.exe
d:/workspace/IA/tp5/tp5.py
Entradas Treinadas
0 [0.82, 0.38, 0.0, 0.09, 0.0]
1 [0.0, 0.27, 0.0, 0.15, 0.01]
2 [0.0, 0.09, 0.8, 0.44, 0.01]
3 [0.0, 0.11, 0.07, 0.36, 0.01]
4 [0.0, 0.01, 0.0, 0.01, 0.99]

            .@@@.
            @...@
            @@@@@
            @...@
            @...@

[0.82, 0.38, 0.0, 0.09, 0.0]
Interprete como variação do dígito A

            @@@@@
            @....
            @@@..
            @....
            @@@@@

[0.0, 0.11, 0.19, 0.38, 0.01]
Interprete como variação do dígito E
```

```
            .@@@.
            ..@..
            ..@..
            ..@..
            .@@@.

[0.0, 0.09, 0.79, 0.44, 0.01]
Interprete como variação do dígito I

            .@@@.
            @...@
            @...@
            @...@
            .@@@.

[0.2, 0.34, 0.0, 0.11, 0.0]
Interprete como possível variação do dígito U

            @...@
            @...@
            @...@
            @...@
            .@@@.

[0.0, 0.01, 0.0, 0.01, 0.98]
Interprete como possível variação do dígito U
PS D:\workspace\IA>
```

Conclusão

> Verificamos que ao realizar os testes com os novos caracteres os valores do treinamento foram bem próximos em alguns caracteres com nas letras A e U que a diferença foi mínima.