Exercise: 6

Meeting on 9th/ 11th February

Aufgabe 1: Properties of the softmax function

The softmax function

$$\text{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

is often used as the final actication function in a multi-class classification problem. Show/explain for the following properties of the softmax function:

(a) $\text{softmax}(\vec{x})_i$ corresponds to a probability distribution. Hence, show that $\sum_i \text{softmax}(\vec{x})_i = 1$.

(b) For i it is for all $j \neq i$: $x_i \gg x_j$, then $\text{softmax}(\vec{x})_i \to 1$ and $\text{softmax}(\vec{x})_j \to 0$. Hence, why is the function called *soft*max?

Solution:

(a) $\sum_i \text{softmax}(\vec{x})_i = \sum_i \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{\sum_i e^{x_i}}{\sum_j e^{x_j}} = 1$

(b) $x_i \gg x_j$ implies $e^{x_i} \gg e^{x_j}$, i.e. $\sum_j e^{x_j} \to e^{x_i}$, hence $\text{softmax}(\vec{x})_i = 1$ and everything else 0. The maximum would be choosing only the largest value with probability/confidence of 1. Softmax allows for a degree of fuzziness. The training error is then not just 1 or 0, but something in between.

Aufgabe 2: Backpropagation in a Deep Network - analytically

Consider a one-dimensional neural network with $\tanh(x)$ as activation function. Forward-Pass and Backpropagation can be represented by:

$$x \xrightarrow{w_1} f_1 \xrightarrow{w_2} f_2 \xrightarrow{w_3} f_3$$

Forward-Pass

$$r_3 \xleftarrow{h_3} r_2 \xleftarrow{h_2} r_1 \xleftarrow{h_1} r_0$$

Backpropagation

The loss function as well as forward and backward variables are computed as

$$\begin{aligned} L(f_3) &= \frac{1}{2}(y - f_3)^2 & r_0 &= h_0 \\ f_3(f_2) &= b_3 + w_3 f_2 & r_1 &= h_1 r_0 \\ f_2(f_1) &= \tanh(b_2 + w_2 f_1) & r_2 &= h_2 r_1 \\ f_1(x) &= \tanh(b_1 + w_1 x) & r_3 &= h_3 r_2 \end{aligned}$$

with

$$h_0 = \frac{\partial L(f_3)}{\partial f_3}$$
$$h_1 = \frac{\partial f_3(f_2)}{\partial f_2}$$
$$h_2 = \frac{\partial f_2(f_1)}{\partial f_1}$$
$$h_3 = \frac{\partial f_1(x)}{\partial x}$$

Compute the derivatives in the variables h_i explicitly. You will need the following derivatives for backpropagation:

$$\begin{aligned}
\frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial w_3} \\
\frac{\partial L}{\partial b_3} &= \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial b_3} \\
\frac{\partial L}{\partial w_2} &= \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial w_2} \\
\frac{\partial L}{\partial b_2} &= \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial b_2} \\
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial w_1} \\
\frac{\partial L}{\partial b_1} &= \frac{\partial L}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial b_1}
\end{aligned}$$

Rewrite these equations using the forward variables f_i and the backward variables r_i.

Solution: The variables h_i are computed as:

- $h_0 = \frac{\partial L(f_3)}{\partial f_3} = \frac{\partial}{\partial f_3}\frac{1}{2}(y - f_3)^2 = -(y - f_3)$
- $h_1 = \frac{\partial f_3(f_2)}{\partial f_2} = \frac{\partial}{\partial f_2}(b_3 + w_3 f_2) = w_3$
- $h_2 = \frac{\partial f_2(f_1)}{\partial f_1} = \frac{\partial}{\partial f_1}\tanh(b_2 + w_2 f_1) = (1 - f_2^2) \cdot w_2$
- $h_3 = \frac{\partial f_1(x)}{\partial x} = \frac{\partial}{\partial x}\tanh(b_1 + w_1 x) = (1 - f_1^2) \cdot w_1$

The derivatives of the loss functions are:

$$\frac{\partial L}{\partial w_3} = h_0 f_2 = r_0 f_2$$
$$\frac{\partial L}{\partial b_3} = h_0 = r_0$$
$$\frac{\partial L}{\partial w_2} = h_0 h_1 (1 - f_2^2) f_1 = r_1 (1 - f_2^2) f_1$$
$$\frac{\partial L}{\partial b_2} = h_0 h_1 (1 - f_2^2) = r_1 (1 - f_2^2)$$
$$\frac{\partial L}{\partial w_1} = h_0 h_1 h_2 (1 - f_1^2) x = r_2 (1 - f_1^2) x$$
$$\frac{\partial L}{\partial b_1} = h_0 h_1 h_2 (1 - f_1^2) = r_2 (1 - f_1^2)$$

Aufgabe 3: Convolution and Pooling operations in a CNN

Consider the following three matrices:

- $A = \begin{pmatrix} 4 & -4 & 1 & 3 \\ -5 & 1 & -1 & 0 \\ 6 & 4 & 7 & 4 \\ 18 & 3 & -5 & 11 \end{pmatrix}$
- $B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{pmatrix}$

As well as the filter matrix W:

$$W = \begin{pmatrix} -3 & 3 \\ -1 & 2 \end{pmatrix}$$

Using W perform the convolution operations on A and B, once without padding (= "valid") and once with padding while keeping the original resolution (= "same").

Solution:

- Without padding (Padding = "valid"):

$$A * W = \begin{pmatrix} 4 & -4 & 1 & 3 \\ -5 & 1 & -1 & 0 \\ 6 & 4 & 7 & 4 \\ 18 & 3 & -5 & 11 \end{pmatrix} * \begin{pmatrix} -3 & 3 \\ -1 & 2 \end{pmatrix} = \begin{pmatrix} -17 & 12 & 7 \\ 20 & 4 & 4 \\ -18 & -4 & 18 \end{pmatrix}$$

$$B * W = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{pmatrix} * \begin{pmatrix} -3 & 3 \\ -1 & 2 \end{pmatrix} = \begin{pmatrix} 10 & 11 & 12 \\ 9 & 8 & 7 \\ -1 & -2 & -3 \end{pmatrix}$$

- With padding = "same":

$$A * W = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & -4 & 1 & 3 \\ 0 & -5 & 1 & -1 & 0 \\ 0 & 6 & 4 & 7 & 4 \\ 0 & 18 & 3 & -5 & 11 \end{pmatrix} * \begin{pmatrix} -3 & 3 \\ -1 & 2 \end{pmatrix} = \begin{pmatrix} 8 & -12 & 6 & 5 \\ 2 & -17 & 12 & 7 \\ -3 & 20 & 4 & 4 \\ 54 & -18 & -4 & 18 \end{pmatrix}$$

$$B * W = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 & 8 \\ 0 & 8 & 7 & 6 & 5 \\ 0 & 4 & 3 & 2 & 1 \end{pmatrix} * \begin{pmatrix} -3 & 3 \\ -1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 13 & 10 & 11 & 12 \\ 31 & 9 & 8 & 7 \\ 32 & -1 & -2 & -3 \end{pmatrix}$$

Aufgabe 4: Programming a simple Neural Network

The lecture showed how you can program and train a simple neural network without any major frameworks. In this exercise you should try this yourself.

(a) Write a class "neuralNetwork" which initializes all necessary parameters, like input and output nodes as well as weights, and possesses the methods "train" and "query". Train and evalute the network on the MNIST[1] dataset.

(b) Try using your own or downloaded images as inputs.

Solution:

(a) **Easy Neural Network in Python**:

```python
import numpy as np
from scipy.special import expit
import csv


class neuralNetwork:
    # Klasse, um ein fully connected neuronales Netz mit einer versteckten Schicht zu erstellen
    def __init__(self,
                 inputnodes: int,
                 hiddennodes: int,
                 outputnodes: int,
                 learningrate: float = 0.01):
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # Elemente der Gewichtsmatrizen sind W_ih und W_ho
        # Gewicht w_ij verbindet den Knoten i der vorherigen Schicht mit dem Knoten j in der naechsten
        # Zufaellige Initialisierung der Gewichte ueber eine Normalverteilung (np.random.normal)
```

[1] https://pjreddie.com/projects/mnist-in-csv/

```python
        self.wih = np.random.normal(0.0, pow(self.hnodes, -0.5), (self.hnodes, self.inodes))
        self.who = np.random.normal(0.0, pow(self.onodes, -0.5), (self.onodes, self.hnodes))

        self.lr = learningrate

        # Aktivierungsfunktion ist sigmoid (auch expit genannt) fuer alle Schichten
        self.activation_function = lambda x: expit(x)

    # Fuehrt ein Gradient Descent Update mit gegebenem Batch aus
    def train(self, inputs_list, targets_list):
        # Eingabe (Bild + Ground Truth) zu 2D Array konvertieren
        # Bilder sind bereits flattened, also ein Bild = eine Zeile
        # und ein one-hot Label-Vektor = eine Zeile, damit zwei Matrizen
        inputs = np.array(inputs_list, ndmin=2).T
        targets = np.array(targets_list, ndmin=2).T

        # Berechnung der versteckten Schicht
        hidden_inputs = np.dot(self.wih, inputs)
        # Anwendung der Aktivierungsfunktion
        hidden_outputs = self.activation_function(hidden_inputs)

        # Berechnung der Ouput-Schicht
        final_inputs = np.dot(self.who, hidden_outputs)
        # Anwendung der Aktivierungsfunktion
        final_outputs = self.activation_function(final_inputs)

        # Berechnung der Fehler der Knoten
        # Fehler der Ausgabeschicht ist einfache Differenz, berechnet aus der Lossfunktion (Berechnung im Uebungsblatt)
        output_errors = targets - final_outputs
        # Fehler der versteckten Schicht propagieren von oben
        hidden_errors = np.dot(self.who.T, output_errors)

        # Gewichtsupdate
        # Exakte Berechnung: elementweise Ableitung der Lossfunktion
        # Mit Backpropagation nur noch abhaengig von den umgebenden Schichten (genaue Formel in der Loesung)

        # Gewichte zwischen versteckt und Output
        self.who += self.lr * np.dot((output_errors * final_outputs * (1.0 - final_outputs)),
                                     np.transpose(hidden_outputs))

        # Gewichte zwischen versteckt und Input
        self.wih += self.lr * np.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), np.transpose(inputs))

    # Berechnet einen Forward-Pass des Netzes
    def query(self, inputs_list):
        # Eingabe zu 2D Array konvertieren
        inputs = np.array(inputs_list, ndmin=2).T

        # Berechnung der versteckten Schicht
```

```python
        hidden_inputs = np.dot(self.wih, inputs)
        # Anwendung der Aktivierungsfunktion
        hidden_outputs = self.activation_function(hidden_inputs)

        # Berechnung der Ouput-Schicht
        final_inputs = np.dot(self.who, hidden_outputs)
        # Finale Ausgabe berechnen
        final_outputs = self.activation_function(final_inputs)

        return final_outputs


# Anzahl der Knoten pro Schicht
input_nodes = 784  # aufgerollte MNIST-Aufloesung von 28 x 28
hidden_nodes = 200  # Anzahl versteckter Knoten waehlbar
output_nodes = 10  # 10 moegliche Klassen als Ausgabe

# Lenrrate
learning_rate = 0.1

# Netzwerk initialisieren
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes, learning_rate)


# MNIST-Daten als numpy array laden
# Aufbau der Daten: Jede Reihe enthaelt das Label als ersten Eintrag und aufgerolltes Bild als Rest
with open("mnist_train.csv", 'r') as train_file:
    reader = csv.reader(train_file)
    train_data = np.asarray(list(reader), dtype=int)
with open("mnist_test.csv", 'r') as test_file:
    reader = csv.reader(test_file)
    test_data = np.asarray(list(reader), dtype=int)


# Training

# Anzahl Epochen, also wie oft wird durch den Datensatz iteriert
epochs = 5

for e in range(epochs):
    # Datensatz mischen
    np.random.shuffle(train_data)
    # Durch die Reihen iterieren
    for row in train_data:
        label = row[0]
        inputs = row[1:]
        # Inputs auf 0 bis 1 skalieren
        inputs = (inputs / 255.0 * 0.999) + 0.001
        # Label One-Hot encoden, also alles auf 0 bis auf Index = Label, den auf 1 (mit label smoothing)
        targets = np.zeros(output_nodes) + 0.001
        targets[label] = 0.999
        n.train(inputs, targets)

```

```

# Testen

# Tracker, der die richtig und falsch erkannten Faelle speichert
scorecard = []

# durch den Test-Datensatz iterieren
for row in test_data:
    # gleiche Aufbereitung wie beim Training
    correct_label = row[0]
    inputs = row[1:]
    inputs = (inputs / 255.0 * 0.999) + 0.001
    # Forward-pass aufrufen
    outputs = n.query(inputs)
    # Label ueber maximale Wahrscheinlichkeit bestimmen
    label = np.argmax(outputs)
    # Korrekte oder falsche Erkennung in die Evaluation eintragen
    if (label == correct_label):
        # 1 bei korrekter Erkennung
        scorecard.append(1)
    else:
        # 0 bei falscher Erkennung
        scorecard.append(0)

# Evaluation

# Performance des Netzes wird ueber die Accuracy bewertet, also korrekt erkannte
    Bilder durch Anzahl aller Bilder
accuracy = np.sum(scorecard) / len(scorecard)
print("Accuracy = ", accuracy)
```

The code from the lecture has been used for the most part. Below some explanations on backpropagation.

Optimizing a network is always performed by minimizing the loss function L, which can theoretically be chosen arbitrarily. In this example we use the l_2 loss, i.e. given prediction $\vec{y}$ and targets $\vec{t}$:

$$L = \frac{1}{2}\sum_i^N \left(\vec{y} - \vec{t}\right)^2$$

The weight update Δw_{jk} for a matrix element of the weight matrix W is computed via the derivative of the loss function (with the chain rule) and can be simplified using the backpropagation algorithm to:

$$\Delta w_{jk} = \alpha \cdot \delta(a_k) \cdot \frac{\partial}{\partial \hat{a}_k}\text{activation}(\hat{a}_k) \cdot a_j$$

where $\delta(a_k)$ is the error of node a_k in the next layer, $\hat{a}_k$ is the value of a_k before activation and a_j the value of node a_j in the previous layer. Since we use the sigmoid function everywhere and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ and $\sigma(\hat{a_k}) = a_k$, the expression simplifies to

$$\Delta w_{jk} = \alpha \cdot \delta(a_k) \cdot a_k(1 - a_k) \cdot a_j.$$

We extend this formula to matrix multiplication in the code so we don't have to compute each element on its own (lines 56 to 61 in the code).

The error of the hidden layer is computed by summing all errors of each directly connected nodes in the next layer, weighted with the corresponding element of the weight matrix connecting the two nodes (line 50). The error of the output layer is in this case the difference between target and output (line 48), which is given by the derivative of the loss function.

(b) **Code fragment for reading in images**:

```
import imageio
import glob
import numpy as np

# Liste fuer unsere Daten
our_images = []
our_labels = []

for image_file_name in glob.glob('custom/meine_?.png'):
    print ("loading ... ", image_file_name)
    # Label wird hier ueber den Dateinamen bestimmt (auch andere Varianten moeglich )
    label = int(image_file_name[-5])
    # als Graustufenbild reinladen
    img_array = imageio.imread(image_file_name, as_gray=True)
    # Von 28x28 zu 784 aufrollen
    img_data  = 255.0 - img_array.reshape(784)
    # Passende Skalierung vornehmen
    img_data = (img_data / 255.0 * 0.999) + 0.001
    # Zur Liste mit allen Bildern und Labels hinzufuegen
    our_images.append(img_data)
    our_labels.append(label)

output = n.query(our_images)
results = [np.argmax(row) for row in output]

# print oder return (results, our_labels)
```

It is best to run it all in one script and save the network weights in a fitting format, to load them later without having to retrain the network. Jupyter notebooks are well suited for this task.

Aufgabe 5: Q-Learning

Consider the *Markov Decision Problem* with three states $Z \in \{1, 2, 3\}$ and their rewards B with $B_1 = -1$, $B_2 = -2$ and $B_3 = 0$. State $Z = 3$ is a terminal state. In the states $Z = 1$ and $Z = 2$ there are two possible actions a and b. The indeterministic transition model is shown in Fig. 1.

In this scenario the agent does not know the transition model nor the rewards. Instead he know only about the number of states and the available actions.

(a) Discuss, which action would be the most sensible one (as an external observer

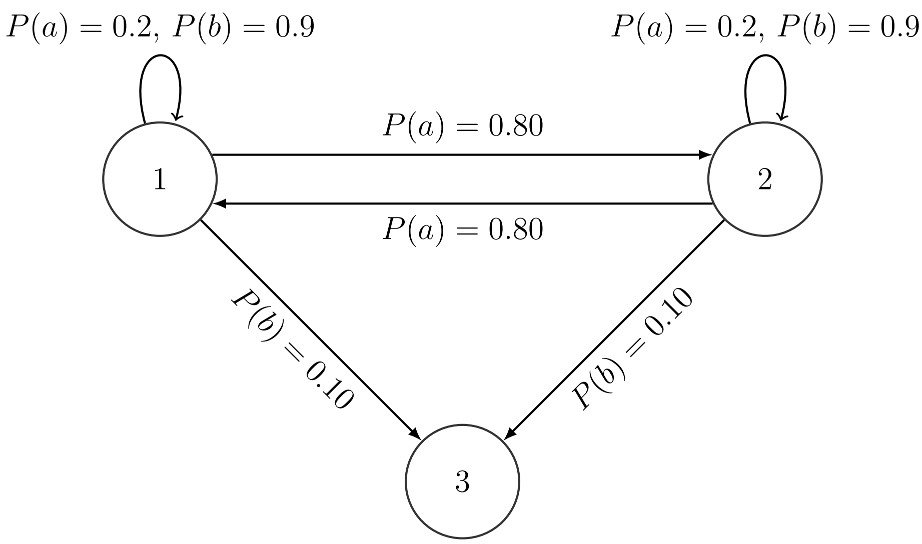


Figure 1: The indeterministic transition model

with access to all information) to end with the highest possible reward (or the lowest penalty) in the terminal state.

Solution: The agent should try to reach $Z = 3$ as quickly as possible since staying in one of the other two states only gives negativ rewards. In state $Z = 1$ they try action b to reach $Z = 3$. In state $Z = 2$ they try action a to reach $Z = 1$ and from there reach $Z = 3$ with action b since there is less penalty for staying in 1.

(b) Explain the components of the formula $Q(s, a)$ for the Q-learning algorithm.

Solution:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t) \cdot \Big(R_{t+1} + \gamma \cdot \max_a Q_t(s_{t+1},a) - Q_t(s_t, a_t)\Big)$$

- $Q_{t+1}(s_t, a_t)$: New Q-Value
- $Q_t(s_t, a_t)$: Old Q-Value
- $\alpha_t(s_t, a_t)$: leraning rate
- R_{t+1}: reward
- γ: Discount factor
- $\max_a Q_t(s_{t+1},a)$: Estimation of the future optimal value
- $R_{t+1} + \gamma \cdot \max_a Q_t(s_{t+1},a)$: learned value

Also see: English wikipedia page for Q-Learning!

(c) What is the meaning if the discount factor γ?

Solution: γ weights the rewards by tiem, i.e. the smaller γ is the faster the agent tries to get rewards since the later rewards are smaller due to weighting.

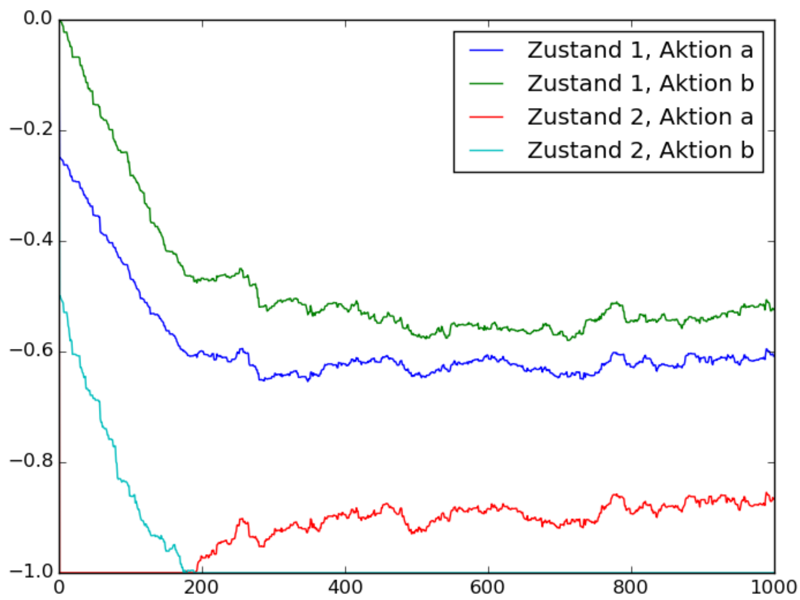
(d) Provide an outline of the sequence of the Q-learning algorithm.

Solution:

1. Initialize $Q(s,a)$-Matrix
2. For every episode do: Choose a random starting point and repeat until a terminal state is reached
   - Randomly choose a possible action in the current state s_t
   - With that explore s_{t+1} (provide the agent with an unkown environment)
   - Compute $Q_{t+1}(s_t, a_t)$ and with that update the Q-Matrix
   - Inkrement t (go from s_t to s_{t+1})

(e) Program the Q-learning algorithm for $\alpha = 0.001$ and show the evolution of the (normalized) Q-matrix across 1000 iterations.

Solution: Here is an examplary parameter evolution. The max. value is normalized to -1:



You can see that the agent first considers action b to be better in state 2 but changes their opinion over time.

Example code in Python:

```python
import numpy as np
import matplotlib.pylab as plt

# world transition matrix (unknown to agent) T(s, a, s')
wtm = np.zeros((3, 2, 3))
wtm[0, 0, 0] = 0.2
wtm[0, 0, 1] = 0.8
wtm[0, 1, 0] = 0.9
wtm[0, 1, 2] = 0.1

wtm[1, 0, 0] = 0.8
wtm[1, 0, 1] = 0.2
wtm[1, 1, 1] = 0.9
wtm[1, 1, 2] = 0.1

reward = [-1, -2, 0]

n_episodes = 1000


def select_random_action():
    # random 0 or 1
    return np.random.randint(0, 2)


def get_new_state(action, cur_state):
    r = np.random.rand()

    t = 0
    for i in range(3):
        t += wtm[cur_state, action, i]
        if r < t:
            return i

    raise Exception('Unexpected error', 'Maybe unallowed state and action')


def get_random_state():
    # random 0, 1, or 2
    return np.random.randint(0, 3)


def is_goal_state(state):
    return state == 2


def compute_Q_value(Q, R, state, action, new_state, alpha=0.001, gamma=0.1):
    return Q[state, action] + alpha * (R[state] + gamma * np.max(Q[new_state, :]) - Q[state, action])


# ALGORITHM
# initialize Q-Matrix
Q = np.zeros((3, 2))
Qmemory = np.zeros((n_episodes, 3, 2))
```

```python

for episode in range(n_episodes):
    print('Episode %d' % episode)
    state = get_random_state()

    while not is_goal_state(state):
        action = select_random_action()
        new_state = get_new_state(action, state)

        Q[state, action] = compute_Q_value(Q, reward, state, action, new_state)

        state = new_state

    # normalize Q (optional)
    m = np.max(np.abs(Q))
    if m > 0:
        Q = Q / m

    Qmemory[episode] = Q

    # print('Result Q=%s' % Q)

plt.plot(Qmemory[:, 0, 0], label='Zustand 1, Aktion a')
plt.plot(Qmemory[:, 0, 1], label='Zustand 1, Aktion b')
plt.plot(Qmemory[:, 1, 0], label='Zustand 2, Aktion a')
plt.plot(Qmemory[:, 1, 1], label='Zustand 2, Aktion b')
plt.legend()
plt.show()
```