KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```python
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```python
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```python
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```python
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg
img = mpimg.imread('BRO_WS_KI/drei.png')
imgplot = plt.imshow(img)
plt.show()
x_my = img
x_my = np.squeeze(x_my[:,:,0])
x_my = np.squeeze(x_my)
x_my = x_my.reshape(-1,1)

y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```python
import matplotlib.image as mpimg
img = mpimg.imread('BRO_WS_KI/drei.png')
imgplot = plt.imshow(img)
plt.show()
x_my = img
x_my = np.squeeze(x_my[:,:,0])
x_my = np.squeeze(x_my)
x_my = x_my.reshape(-1,1)

y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen

data = pd.read_csv('BRO_WS_KI/train.csv')


# Ein paar Zeilen aus den Daten zeigen

data.head()

data = np.array(data)

# Dimension der Daten zeigen

m, n = data.shape

print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)


data_dev = data[0:validation_samples_number].T

Y_dev = data_dev[0]

X_dev = data_dev[1:n]

X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train, 
Y_train, 0.10, 500)


plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```python
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```python
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```python
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```python
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen

data = pd.read_csv('BRO_WS_KI/train.csv')


# Ein paar Zeilen aus den Daten zeigen

data.head()

data = np.array(data)

# Dimension der Daten zeigen

m, n = data.shape

print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)


data_dev = data[0:validation_samples_number].T

Y_dev = data_dev[0]

X_dev = data_dev[1:n]

X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```python
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```python
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```python
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```python
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)


plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1,
b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```python
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```python
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```python
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```python
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)


plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg
img = mpimg.imread('BRO_WS_KI/drei.png')
imgplot = plt.imshow(img)
plt.show()
x_my = img
x_my = np.squeeze(x_my[:,:,0])
x_my = np.squeeze(x_my)
x_my = x_my.reshape(-1,1)

y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```python
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```python
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```python
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```python
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1,
b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen

data = pd.read_csv('BRO_WS_KI/train.csv')


# Ein paar Zeilen aus den Daten zeigen

data.head()

data = np.array(data)

# Dimension der Daten zeigen

m, n = data.shape

print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)


data_dev = data[0:validation_samples_number].T

Y_dev = data_dev[0]

X_dev = data_dev[1:n]

X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)


plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```python
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```python
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```python
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```python
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)


plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```python
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen
data = pd.read_csv('BRO_WS_KI/train.csv')

# Ein paar Zeilen aus den Daten zeigen
data.head()
data = np.array(data)
# Dimension der Daten zeigen
m, n = data.shape
print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)

data_dev = data[0:validation_samples_number].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg
img = mpimg.imread('BRO_WS_KI/drei.png')
imgplot = plt.imshow(img)
plt.show()
x_my = img
x_my = np.squeeze(x_my[:,:,0])
x_my = np.squeeze(x_my)
x_my = x_my.reshape(-1,1)

y_my = make_predictions(x_my,W1,b1,W2,b2)
```

KI-Workshop

Benutzername: **xxx** Passwort: **xxx**

Bibliotheken importieren.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

Daten erfassen:

```
# Daten einlesen

data = pd.read_csv('BRO_WS_KI/train.csv')


# Ein paar Zeilen aus den Daten zeigen

data.head()

data = np.array(data)

# Dimension der Daten zeigen

m, n = data.shape

print(f"Zeilen: {m} und Spalten: {n}")
```

Daten verarbeiten:

```
np.random.shuffle(data)
```

```
validation_samples_number = int(0.2 * m)


data_dev = data[0:validation_samples_number].T

Y_dev = data_dev[0]

X_dev = data_dev[1:n]

X_dev = X_dev / 255.
```

```
data_train = data[validation_samples_number:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
```

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
```

Modell aufbauen:

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
```

```
# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)

def Softmax(z):
    exp_z = np.exp(z - np.max(z)) # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)
    return x * (1 - x)
```

```python
def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = Softmax(Z2)
```

```python
def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)

    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_derivative(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

```python
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```python
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

```python
def compute_cost(A2, Y):
    one_hot_Y = one_hot(Y)
    logprobs = -np.log(A2) * one_hot_Y
    cost = np.mean(logprobs)
    return cost
```

```python
def plot_cost_graph(cost_history):
    iterations = len(cost_history)
    plt.figure(figsize=(22, 8))
    plt.plot(range(iterations), cost_history)
    plt.title('Cost vs. Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.grid(True)
    plt.show();
```

```python
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    cost_history = []  # List to store cost values at each iteration

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))

        cost = compute_cost(A2,Y)
        cost_history.append(cost)
```

```python
W1, b1, W2, b2, cost_history= gradient_descent(X_train,
Y_train, 0.10, 500)

plot_cost_graph(cost_history)
```

Modell evaluieren:

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)

    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Eigene Handschrift testen:

```
import matplotlib.image as mpimg

img = mpimg.imread('BRO_WS_KI/drei.png')

imgplot = plt.imshow(img)

plt.show()

x_my = img
x_my = np.squeeze(x_my[:,:,0])

x_my = np.squeeze(x_my)

x_my = x_my.reshape(-1,1)


y_my = make_predictions(x_my,W1,b1,W2,b2)
```