課題1

作成者：Pyii Phyo Maung

December 16, 2022

1 Task

Write a program that takes one MNIST image as input and outputs one of the values from 0 to 9 using a three-layer neural network.

2 The Requirements

- Uses one MNIST image as an input, a three-layer network is used to extract values from 0 to 9.
- Receives an integer from 0 to 9999 from the keyboard as input i and outputs and integer from 0 to 9 to standard output.
- Uses the i-th image among the 10000 images in the MNIST test data as the input image.
- The MNIST image size (28x28), the number of images (10000) and the number of classes (C=10) are known but they can be changeable for future improvement.
- The number of nodes M in the intermediate layer can be freely determined.
- Weights W1, W2, B1 and B2 can be determined by random numbers, given by a normal distribution with a mean of 0 and a variance of 1/N. where N is the number of nodes. The random number seed is fixed so the same result is output each time it is executed.

3 The Python Code

```
import numpy as np
import mnist
import matplotlib.pyplot as plt
from pylab import cm

# Set the input shape, number of images, and number of classes
input_shape = (28, 28, 1)
num_images = 10000
num_classes = 10

# Load the MNIST dataset
X = mnist.download_and_parse_mnist_file("t10k-images-idx3-ubyte.gz")
Y = mnist.download_and_parse_mnist_file("t10k-labels-idx1-ubyte.gz")

# Normalize the images
X = X / 255.0
```

```
# Convert the labels to one-hot encoded vectors
Y = np.eye(num_classes)[Y]

# Flatten the images
X = X.reshape(X.shape[0], 28, 28, 1)

# Get input i from the user
i = int(input("Enter an integer from 0 to 9999: "))

# Get the i-th image from the MNIST test data
x = X[i]

# The number of nodes in the intermediate layer that can be set by the user
M = 100

# Initialize the weights and biases with random numbers from a normal distribution
W1 = np.random.normal(0, 1/M, (M, input_shape[0]*input_shape[1]))
W2 = np.random.normal(0, 1/M, (num_classes, M))
B1 = np.random.normal(0, 1/M, M)
B2 = np.random.normal(0, 1/M, num_classes)

# Flatten the input image
x = x.flatten()

# Perform the forward pass through the neural network
z1 = np.dot(W1, x) + B1
a1 = 1 / (1 + np.exp(-z1))
z2 = np.dot(W2, a1) + B2
alpha = np.max(z2)
a2 = np.exp(z2 - alpha) / np.sum(np.exp(z2 - alpha))

# Get the index of the class with the highest probability
output = np.argmax(a2)

# Print the output to standard output
print(a2)
print(output)

# Display the image and label
idx = i
plt.imshow(X[idx], cmap=cm.gray)
plt.show()
print(Y[idx])
```

4 The Explanation of the Code

This code is a simple implementation of a neural network that can classify handwritten digits from the MNIST dataset.

4.1

```
import numpy as np
import mnist
import matplotlib.pyplot as plt
from pylab import cm
```

The first thing the code does is import several libraries that will be used later in the code. The numpy library is used for numerical operations, mnist is a library for downloading and parsing the MNIST dataset, matplotlib is used for plotting and visualizing data, and pylab is a library that provides additional functions for working with matplotlib.

4.2

```
# Set the input shape, number of images, and number of classes
input_shape = (28, 28, 1)
num_images = 10000
num_classes = 10
```

Next, the code defines some constants that will be used later in the code: input_shape, num_images, and num_classes. input_shape is a tuple representing the shape of the input images in the MNIST dataset (28x28 pixels and 1 color channel). num_images is the number of images in the MNIST dataset that will be used for training or testing, and num_classes is the number of classes (digits from 0 to 9) that the neural network will classify.

4.3

```
# Load the MNIST dataset
X = mnist.download_and_parse_mnist_file("t10k-images-idx3-ubyte.gz")
Y = mnist.download_and_parse_mnist_file("t10k-labels-idx1-ubyte.gz")

# Normalize the images
X = X / 255.0
```

The code then uses the download_and_parse_mnist_file function from the mnist library to download and parse the MNIST test images and labels. The images are stored in a variable called X and the labels are stored in a variable called Y.

The images in X are then normalized by dividing each pixel value by 255.0. This is a common preprocessing step that scales the pixel values to be between 0 and 1 so that overflow is eliminated.

4.4

```
# Convert the labels to one-hot encoded vectors
Y = np.eye(num_classes)[Y]
```

The labels in Y are then converted to one-hot encoded vectors using the np.eye function. One-hot encoding is a way to represent categorical variables as numerical data. It is commonly used in machine learning models, especially in classification tasks where the target variable has more than two categories. One-hot encoding converts each category value into a new categorical feature, with a binary value of 0 or 1. For example, if the target variable has three categories A, B, and C, one-hot encoding would create three new binary features, one for each category, with a value of 1 for the corresponding category and 0 for all other categories.

For example:

```
# Original categorical data
categories = ['A', 'B', 'C']
data = ['A', 'B', 'C', 'B', 'A']

# One-hot encoded data
one_hot_data = [[1, 0, 0],
                [0, 1, 0],
                [0, 0, 1],
                [0, 1, 0],
                [1, 0, 0]]
```

In this case, each label (a digit between 0 and 9) is represented as a vector with 10 elements, where the element corresponding to the correct class is 1 and all other elements are 0. For example, the label "3" would be represented as the vector [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

4.5

```
# Flatten the images
X = X.reshape(X.shape[0], 28, 28, 1)

# Get input i from the user
i = int(input("Enter an integer from 0 to 9999: "))

# Get the i-th image from the MNIST test data
x = X[i]

# The number of nodes in the intermediate layer that can be set by the user
M = 100
```

The images in X are then reshaped from a 2D array to a 4D array, with the shape (number of images, 28, 28, 1). This is done so that the images can be passed through the neural network. The shape of the 2D array after flattening the images is (num_images, 784).

The code then prompts the user to input an integer between 0 and 9999, which will be used to select one of the images in the MNIST dataset. This selected image is stored in the variable x.

The variable M then stores the number of nodes in the intermediate layer of the neural network.

4.6

```
# Initialize the weights and biases with random numbers from a normal distribution
W1 = np.random.normal(0, 1/M, (M, input_shape[0]*input_shape[1]))
W2 = np.random.normal(0, 1/M, (num_classes, M))
B1 = np.random.normal(0, 1/M, M)
B2 = np.random.normal(0, 1/M, num_classes)

# Flatten the input image
x = x.flatten()
```

Next, the code initializes the weights and biases of the neural network with random numbers from a normal distribution. The weights and biases are stored in the variables W1, W2, B1, and B2. The weights W1 and W2 are used to multiply the input image and the output of the first layer, respectively. The size of the weights matrix is determined by the number of nodes in the current layer and the number of nodes in the previous layer.

The selected image x is then flattened from a 2D array to a 1D array. This is done so that the image can be passed through the neural network.

4.7

```
 # Perform the forward pass through the neural network
z1 = np.dot(W1, x) + B1
a1 = 1 / (1 + np.exp(-z1))
z2 = np.dot(W2, a1) + B2
alpha = np.max(z2)
a2 = np.exp(z2 - alpha) / np.sum(np.exp(z2 - alpha))

# Get the index of the class with the highest probability
output = np.argmax(a2)
```

The code then performs the forward pass through the neural network. The forward pass consists of two steps: the first step is to calculate the output of the first layer, and the second step is to calculate the output of the second (output) layer.

In the first step, the output of the first layer is calculated as the dot product of the weights W1 and the input image x, plus the biases B1. This result is passed through the sigmoid function, which is used as the activation function of the first layer. The sigmoid function maps any input value to a value between 0 and 1, which is useful for modeling probabilities.

In the second step, the output of the second (output) layer is calculated as the dot product of the weights W2 and the output of the first layer A1, plus the biases B2. This result is then passed through the softmax function, which is used as the activation function of the second layer. The softmax function maps the output values to a probability distribution, where the sum of all the output values is 1. This is useful for classification tasks, as the output value with the highest probability can be selected as the predicted class.

After the forward pass is completed, the code gets the index of the class with the highest probability using the np.argmax function. This index is stored in the variable output.

4.8

```
# Print the output to standard output
print(a2)
print(output)

# Display the image and label
idx = i
plt.imshow(X[idx], cmap=cm.gray)
plt.show()
print(Y[idx])
```

The code then prints the output probabilities and the predicted class to the standard output.

Finally, the code uses matplotlib to display the selected image and its label, which is obtained from the Y array.