**CSC3014 Computer Vision**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Lab 7: Artificial Neural Network**

**A. Building a Simple Artificial Neural Network Model using Python**

As shown in Figure 1 is a simple neural network with three layers (one input layer, one hidden layer, and one output layer). The code used to model such a neural network can be found in **Appendix A** [1]. The code can be divided into three sections, (i) constructor for the neural network, (ii) how the training should be conducted, and (iii) how the query should be conducted.

Diagram

Description automatically generated

Figure 1: A simple neural network with three layers.

We will first look at the section used to create a constructor and initilize the neural network (line 3 to 26). This section allows users to create and set the number of nodes that should be used for each layer, as well as the learning rate. We then proceed to create the two weight matrices, wih and who. Based on the example shown in Figure 1, wih represents the weights in between the input layer and the hidden layer, and who represents the weights in between the hidden layer and the output later. In addition to all these, the activation function, which is a sigmoid function, is also defined in the constructor.

Before we look at how the training should be conducted, let us first look at the code used to query (or test) the neural network (line 55 to 70), because they are partially the same. In this case, if the inputs to the neural network are stored in the form of a list, the inputs need to be first converted into an array with n row and one column (e.g. 1 x 3 to 3 x 1). Hence, if the inputs are already prepared in the required format (an array), then the conversion can be omitted. An example of this conversion, together with the steps that will be performed after the conversion is illustrated in Figure 2.

Diagram, schematic

Description automatically generated

Figure 2: An example of converting inputs to the form of an array before matrix multiplication.

As you can see from Figure 2, the inputs are then multiplied with the weights wih defined in the constructor using matrix multiplication (line 61). This will produce the inputs to the hidden nodes. The inputs are then feed into the sigmoid function to get the outputs for the hidden nodes (line 63). Next, the same process is repeated by multiplying these outputs with weights who, and then feed into the sigmoid function again to obtain the outputs for the output nodes (line 65 to 68).

Finally for the training section (line 29 to 53), it can be seen that part of code (line 31 to 42) is similar to the code used to query (or test) the neural network. In other words, based on a training sample, we need to first query the neural network to obtain the outputs from the output nodes (line 35 to 42), and then compare the outputs with the expected outputs (targets) to calculate the errors for the output nodes, as well as the errors for the hidden nodes via back propagation (line 45 to 47). But in addition to all these, we also need to convert the expected outputs (targets) into an array (line 32). At the end, the weights wih and who are adjusted accordingly (line 50 to 53) based on Equation 1, where the descriptions for all the notations are summarized in Table 1.

∆wj,k = α x [ [ek x ok(1 – ok)] ∙ ojT ] Equation (1)

Table 1: Descriptions for notations used in Equation 1.

|  |  |
| --- | --- |
| **Notation** | **Description** |
| α | Learning rate. |
| ek | Error produced by node k in the next layer. |
| ok | Output produced by node k in the next layer. |
| oj | Output produced by node j in the previous layer. |
| · | Matrix multiplication (dot product). |
| T | Transpose (swap the row coordinate with the column coordinate). |

**B. Training and Testing using the MNIST Dataset**

Instead of having to split the digits (or read every single digit from an image file) and flatten the array one-by-one as shown in Figure 3, we will be using the **MNIST** dataset that already did this for us [2]. In this case, all the class labels as well as the intensity values (after flattening) are arranged and stored in two CSV files, one for the training dataset, one for the testing dataset. In other words, we can skip the steps required to prepare the digits for training and testing.

A screenshot of a cell phone

Description automatically generated

Figure 3: Splitting and flattening of an image with 5000 digits into training set and testing set.

In the CSV file for the training dataset, there are 60,000 rows. Each row consists of a class label, which is stored as the first digit of the row, follows by 784 intensity values extracted from a handwritten digit (the spatial resolution of each handwritten digit is 28 x 28). As shown in Figure 4 is the first training sample extracted from the CSV file. However, please take note that all the intensity values are stored in the form of string. Therefore, they must be converted into integer or float before we can use them for calculations.

Background pattern

Description automatically generated

Figure 4: First training sample extracted from the CSV file that contains the MNIST training dataset.

As shown in **Appendix B** is the code that (i) demonstrates how we can create a neural network based on the model we discussed in the previous section, (ii) prepare the training and testing datasets, and (iii) evaluate the neural network [1]. Similarly, the code can be divided into three sections, (i) create the neural network for the MNIST dataset, (ii) train the neural network using the training dataset, and (iii) evaluate the neural network using the testing dataset.

First, let us look at the section that creates the neural network (line 5 to 13). In this case, we are creating a neural network with 784 input nodes, 200 hidden nodes, and 10 output nodes, The learning rate is set to 0.01. The number of input nodes and output nodes are fixed, because each pixel consists of 784 intensity values and there are 10 different classes (from digit 0 to 9). Hence, the number of hidden nodes is the only parameter we can adjust. In general, the number should be smaller than the number of input nodes. To begin with, we set this to 200.

Before we start to train the neural network, we need to first read the data for the 60,000 training samples from the CSV file (line 16 – 18). The data in each sample (or row) is stored as a long string. In other words, the series of digits shown in Figure 4 is actually stored as a string. Hence, we need to first split them into separate digit (but still in the form of string) based on the commas (,), and put them into a list (line 28). Then, the list is converted into an array by using the asfarray() function (line 32). This function will also convert each digit from string to float, before putting it into the array. At the same time, we also perform the necessary scaling to scale the intensity values to the range of 0.01 to 1.00.

Regarding the expected outputs required for each training sample, we first create an array based on the number of classes (which is 10 in this case), and set all the values to 0.01 (line 35). Then, only the class label at index m is set to 0.99, where m is the class label obtained from the first digit of a training sample (line 35 to 39). For example, if the first digit (or the class label) is 5, then targets[5] will be set to 0.99. Once everything is ready, the train() function is called to start the training (line 42). You can see that the number of epochs is set to 5 (line 22). This means that the 60,000 training samples will be repeatedly used to train the neural network for 5 rounds.

There is not much difference between the code used to evaluate and train the neural network. For evaluation, we start with reading the testing samples from the CSV file (line 46 to 48), split the digits in each sample based on commas (line 57), put all the digits into an array and convert them to float before scaling the intensity values to the range of 0.01 to 1.00 (line 64), and then call the query() function to obtain the outputs (line 67). But on top of all these, we also need to prepare the correct label for each sample, so that we can compare the outputs with the correct labels to calculate the overall accuracy (line 83 to 84) based on Equation 2.

Equation (2)

To achieve the above, we first prepare an empty list (line 52). Then for each comparison, a ‘1’ will be added to the list if we have a correct match. Otherwise, a ‘0’ will be added to the list for an incorrect match (line 75 – 80). Since each sample will produce 10 outputs (there are 10 output nodes), we can identify at which index the output is the highest (the class predicted by the neural network) by using the argmax() function (line 72).

|  |
| --- |
| **Exercise**  Put the two CSV files that contain the MNIST training dataset and testing dataset respectively into the same folder where you placed the two Python codes shown in Appendix A and B. Next, train and evaluate the neural network by using the MNIST training dataset and testing dataset. You can try to adjust the number of hidden nodes, as well as the number of epochs and the learning rate to see how they could affect the overall accuracy and the time needed for training. |

**C. Building a Simple Artificial Neural Network Model using the OpenCV Machine Learning Package**

Besides building our own neural network, we can also choose to use the methods provided by the OpenCV org.opencv.ml package. As shown in Appendix C is how we can make use of the methods provided to create a similar neural network. Similarly, we have to first create a neural network and define the structure of it (line 4 – 8). Then we can proceed to define the activation function, learning method, and learning rate that should be adopted (line 10 – 14).

After we have done with the creation and setup of the neural network. We must again prepare the digits from the MNIST dataset for training and testing. The preparation (line 16 – 48) is similar to the steps described in the previous section. When using the neural network we built ourselves, we used a for loop to feed every single training sample to the neural network one-by-one. But when comes to the neural network constructed using methods provided by the OpenCV org.opencv.ml package, we can put all the training samples into an array and then feed the entire array to the neural network. Therefore, some additional steps have to be carried out to put the training samples, as well as the class labels, into arrays (line 21 – 23, 33 – 35, 44 – 45).

Once we have done preparing the training samples, we can send them to the neural network for training (line 48). It might take a while to train the neural network. After the training is completed, we can proceed to test the neural network. To test the neural network, we need to prepare the testing samples according to the steps we took to prepare the training samples (line 50 – 75). The testing and training samples must be arranged in the same pattern in order for the neural network to work. Lastly, we can accuracy the accuracy of the neural network that we have trained by calculating how many testing samples have been correctly predicted (line 77 – 103).

|  |
| --- |
| **Exercise**  Put the two CSV files that contain the MNIST training dataset and testing dataset respectively into the same folder where you placed the Python code shown in Appendix C. Next, train and evaluate the neural network by using the MNIST training dataset and testing dataset. Try to compare the accuracy that you obtained here with the accuracy that you obtained from the previous exercise. |

**References**

1. Tariq Rashid, Make Your Own Neural Network, 2016.
2. Joseph Chet Redmon, MNIST in CSV, [Link](https://pjreddie.com/projects/mnist-in-csv/).
3. Michael Beyeler, Machine Learning for OpenCV, 2017.

**Appendix A: Building a Simple Artificial Neural Network Model using Python (neuralNetwork.py) [1]**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | import numpy as np  class neuralNetwork:  # Create and initialise the neural network (constructor).  def \_\_init\_\_(self, input\_nodes, hidden\_nodes, output\_nodes, learning\_rate):  # Set number of nodes in the input, hidden, and output layer.  self.inodes = input\_nodes  self.hnodes = hidden\_nodes  self.onodes = output\_nodes    # Create the weight matrices (wih and who).  # wih is the weights for the links between nodes in the input layer and the hidden layer.  # who is the weights for the links between nodes in the hidden layer and the output layer.  self.wih = np.random.rand(self.hnodes, self.inodes)-0.5  self.who = np.random.rand(self.onodes, self.hnodes)-0.5  # Set the learning rate.  self.lr = learning\_rate    # Define the activation function (sigmoid function)  def sigmoid(x):  sig = 1 / (1 + np.exp(-x))  return sig    # Set the activiation function (sigmoid function)  self.activation\_function = lambda x: sigmoid(x)  # Train the neural network.  def train(self, inputs\_list, targets\_list):  # Convert a list into an array (e.g. 1 x 3 then apply transpose to make it 3 x 1).  inputs = np.array(inputs\_list, ndmin=2).T  targets = np.array(targets\_list, ndmin=2).T    # Calculate signals into hidden layer.  hidden\_inputs = np.dot(self.wih, inputs)  # Calculate the output signals from the hidden layer.  hidden\_outputs = self.activation\_function(hidden\_inputs)    # Calculate signals into final output layer.  final\_inputs = np.dot(self.who, hidden\_outputs)  # Calculate the output signals from final output layer.  final\_outputs = self.activation\_function(final\_inputs)    # Calcualte output layer error (target - actual).  output\_errors = targets - final\_outputs  # Calculate hidden layer error based on output layer error (split by weights and recombined at hidden nodes).  hidden\_errors = np.dot(self.who.T, output\_errors)    # Update the weights for the links between the hidden layer and the output layer.  self.who += self.lr \* np.dot((output\_errors \* final\_outputs \* (1.0 - final\_outputs)), np.transpose(hidden\_outputs))    # update the weights for the links between the input layer and the hidden layer.  self.wih += self.lr \* np.dot((hidden\_errors \* hidden\_outputs \* (1.0 - hidden\_outputs)), np.transpose(inputs))    # Query (test) the neural network.  def query(self, inputs\_list):  # Convert a list into an array (e.g. 1 x 3 then apply transpose to make it 3 x 1).  inputs = np.array(inputs\_list, ndmin=2).T    # Calculate signals into hidden layer.  hidden\_inputs = np.dot(self.wih, inputs)  # Calculate the output signals from the hidden layer.  hidden\_outputs = self.activation\_function(hidden\_inputs)    # Calculate signals into final output layer.  final\_inputs = np.dot(self.who, hidden\_outputs)  # Calculate the output signals from final output layer.  final\_outputs = self.activation\_function(final\_inputs)    return final\_outputs |

**Appendix B: Training and Testing using the MNIST Dataset (queryMNIST.py) [1]**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85 | from neuralNetwork import neuralNetwork  import numpy as np  # Set the number of input, hidden and output nodes  input\_nodes = 784  hidden\_nodes = 200  output\_nodes = 10  # Set the learning rate.  learning\_rate = 0.01  # Create a neural network based on the settings given above.  n = neuralNetwork(input\_nodes,hidden\_nodes,output\_nodes, learning\_rate)  # Load the MNIST training data stored in the CSV file into a list.  training\_data\_file = open("mnist\_train.csv", 'r')  training\_data\_list = training\_data\_file.readlines()  training\_data\_file.close()  # Train the neural network.  # Epochs is the number of times the training data set is used for training.  epochs = 5  for e in range(epochs):  # Go through all samples in the training data set.  for sample in training\_data\_list:  # Split the intensity values based on commas (,) and store the values into a list.  all\_values = sample.split(',')    # Convert each intensity value to an integer (from string) and store in an array.  # Scale the intensity values to the range of 0.01 to 1.00.  inputs = (np.asfarray(all\_values[1:]) / 255.0 \* 0.99) + 0.01    # Create the expected output values and set all to 0.01.  targets = np.zeros(output\_nodes) + 0.01    # The first value in a sample (all\_values[0]) is the correct label.  # Set the excepted output value of the correct class to 0.99.  targets[int(all\_values[0])] = 0.99    # Start training.  n.train(inputs, targets)  # Load the MNIST test data stored in the CSV file into a list.  test\_data\_file = open("mnist\_test.csv", 'r')  test\_data\_list = test\_data\_file.readlines()  test\_data\_file.close()  # Test the neural network.  # Use to store matching result (1 for correct match, else 0), initially empty.  accuracy\_list = []  # Go through all the samples in the test data set  for sample in test\_data\_list:  # Split the intensity values based on commas (,) and store the values into a list.  all\_values = sample.split(',')    # The first value in a sample (all\_values[0]) is the correct label.  correct\_label = int(all\_values[0])    # Convert each intensity value to an integer (from string) and store in an array.  # Scale the intensity values to the range of 0.01 to 1.00.  inputs = (np.asfarray(all\_values[1:]) / 255.0 \* 0.99) + 0.01    # Query the network network.  outputs = n.query(inputs)    # Get the index of highest output.  # For example, if outputs[5] has the higest output, then the index returned is 5.  # The index corresponds to the predicted label.  label = np.argmax(outputs)    # Append the matching result to the list.  if (label == correct\_label):  # Add 1 to indicata a correct match.  accuracy\_list.append(1)  else:  # Add 1 to indicata an incorrect match.  accuracy\_list.append(0)  # Calculate the overall acurracy.  accuracy\_array = np.asarray(accuracy\_list)  print ("Accuracy = ", accuracy\_array.sum() / accuracy\_array.size) |

**Appendix C: Building a Simple Artificial Neural Network Model using OpenCV Machine Learning Package**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104 | import cv2  import numpy as np  #Create the artificial neural network.  ann = cv2.ml.ANN\_MLP\_create()  #Set the number of nodes in the input layer, hidden layer and output layer.  ann.setLayerSizes(np.array([784,200,10]))  #Set the activation function (sigmoid function is selected in this case).  ann.setActivationFunction(cv2.ml.ANN\_MLP\_SIGMOID\_SYM, 2.5, 1.0)  #Set the training method and learning rate.  ann.setTrainMethod(cv2.ml.ANN\_MLP\_BACKPROP, 0.01)  # Load the MNIST training data stored in the CSV file into a list.  training\_data\_file = open("mnist\_train.csv", 'r')  training\_data\_list = training\_data\_file.readlines()  training\_data\_file.close()  # Prepare an array to store all the training data.  training\_data\_array = np.zeros((60000,784),np.float32)  training\_data\_class\_label\_array = np.zeros((60000,10),np.float32)  for x in range(len(training\_data\_list)):  # Split the intensity values based on commas (,) and store the values into a list.  all\_values = training\_data\_list[x].split(',')    # Convert each intensity value to an integer (from string) and store in an array.  # Scale the intensity values to the range of 0.01 to 1.00.  inputs = (np.asfarray(all\_values[1:]) / 255.0 \* 0.99) + 0.01    # Reshape and put all the intensity values into the array.  inputs = np.reshape(inputs,(1,784))  training\_data\_array[x,:] = inputs    # Create the expected output values and set all to 0.01.  targets = np.zeros(10) + 0.01    # The first value in a sample (all\_values[0]) is the correct label.  # Set the excepted output value of the correct class to 0.99.  targets[int(all\_values[0])] = 0.99    # Reshape and put the expected value of each class into the array.  training\_data\_class\_label\_array[x,:] = targets  # Train the artificial neural network using the training data.  ann.train(training\_data\_array,cv2.ml.ROW\_SAMPLE,training\_data\_class\_label\_array)  # Load the MNIST testing data stored in the CSV file into a list.  test\_data\_file = open("mnist\_test.csv", 'r')  test\_data\_list = test\_data\_file.readlines()  test\_data\_file.close()  # Prepare an array to store all the testing data.  test\_data\_array = np.zeros((10000,784),np.float32)  test\_data\_answer\_array = np.zeros((10000,1),np.float32)  for x in range(len(test\_data\_list)):  # Split the intensity values based on commas (,) and store the values into a list.  all\_values = test\_data\_list[x].split(',')    # Convert each intensity value to an integer (from string) and store in an array.  # Scale the intensity values to the range of 0.01 to 1.00.  inputs = (np.asfarray(all\_values[1:]) / 255.0 \* 0.99) + 0.01    # Reshape and put all the intensity values into the array.  inputs = np.reshape(inputs,(1,784))  test\_data\_array[x,:] = inputs    # The first value represents the correct answer (i.e. 0 - 9).  # It is not an intensity value.  # Store the first values (the correct answers) into the array.  # The answers will  test\_data\_answer\_array[x,:] = int(all\_values[0])    # Use the trained artificial neural network for prediction.  \_, all\_prediction\_results = ann.predict(test\_data\_array)  # To evalutate the neural network.  # Use to store matching result (1 for correct match, else 0), initially empty.  accuracy\_list = []  for x in range(len(all\_prediction\_results)):  # Get the index of highest output.  # For example, if outputs[5] has the higest output, then the index returned is 5.  # The index corresponds to the predicted label.  prediction = np.argmax(all\_prediction\_results[x,:])    # Get the correct answer.  answer = test\_data\_answer\_array[x,:]    # Append the matching result to the list.  if (prediction == answer):  # Add 1 to indicata a correct match.  accuracy\_list.append(1)  else:  # Add 1 to indicata an incorrect match.  accuracy\_list.append(0)  # Calculate the overall acurracy.  accuracy\_array = np.asarray(accuracy\_list)  print ("Accuracy = ", accuracy\_array.sum() / accuracy\_array.size) |