CSCI 264

MLP, KNN and CNN

Project 2

Janaki Panneerselvam

Spring 2023

Introduction:

In the domain of Machine Learning, an important task is the classification of data into pre-defined categories. There are various algorithms and techniques that have been developed and used to accomplish this task, with each having its own strengths and weaknesses. In this project, we aim to explore and implement from scratch two of the most commonly used classifiers - Multi-Layer Perceptron (MLP) and K-Nearest Neighbors (KNN).

The Multi-Layer Perceptron (MLP) is a type of artificial neural network that consists of at least three layers of nodes and is used to classify data that is not linearly separable. On the other hand, K-Nearest Neighbors (KNN) is a simple, instance-based learning algorithm that classifies a new instance based on its similarity to existing instances in the data.

We will be using the MNIST database, a large database of handwritten digits that is commonly used for training and testing in the field of machine learning. The MNIST database will provide us a good playground to understand and evaluate the performance of these classifiers.

The performance of the classifiers will be assessed based on the accuracy of their predictions and the confusion matrix. The confusion matrix is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one. The accuracy is the proportion of the total number of predictions that were correct.

For the MLP, we will experiment with different numbers of units in the hidden layer, and for the KNN, we will experiment with different values of k (the number of nearest neighbors to consider).

In addition to these classifiers, we will also implement a Convolutional Neural Network (CNN) using existing MATLAB functions. CNNs are a class of deep learning neural networks, most commonly applied to analyzing visual imagery. They have applications in image and video recognition, recommender systems and natural language processing.

Through this project, we aim to gain deeper understanding and insights into the working and performance of these classifiers. By analysing the results, we hope to determine which classifier and parameters provide the best performance for digit classification on the MNIST dataset.

Implementation:

MLP:

Sure, I can break down the steps used to calculate the Multi-Layer Perceptron (MLP) in the given code.

1. Data Pre-processing: The first steps involve loading and pre-processing the data. The pixel values of the MNIST images are normalized by dividing by 255 (the maximum value), transforming them from the range [0, 255] to [0, 1]. The data is then split into a training set and a test set.

2. Model Initialization: An instance of the MLP class is created with the number of input nodes equal to the number of features (784 for MNIST), a given number of hidden nodes, and a number of output nodes equal to the number of classes (10 for MNIST). The weights and biases for each layer are initialized in the `__init__` method. Weights are initialized with small random numbers, and biases are initialized to zero.

3. Forward Pass: The `forward` method carries out the forward pass of the network. It first calculates the input to the hidden layer (`self.z1`), which is the dot product of the input `X` and the weights `self.W1`, plus the bias `self.b1`. This is then passed through the activation function, here a hyperbolic tangent function (`np.tanh`), to get the output of the hidden layer (`self.a1`). This process is repeated for the hidden layer to output layer transition, but the softmax function is applied to the output to get probability distributions for each class (`self.probs`).

4. Loss Calculation: The `loss` method computes the cross-entropy loss function, which is commonly used in classification problems. It first carries out a forward pass to get the probabilities, then uses these to calculate the log loss for each input example, and finally returns the mean loss over all examples.

5. Backward Pass (Backpropagation): In the `train` method, the error for each output node is first calculated as `delta`. The error for each node in the hidden layer is then calculated as `delta2`, by backpropagating the errors from the output layer. This involves computing the dot product of `delta` and the transpose of `self.W2`, and then multiplying element-wise by the derivative of the activation function applied to `self.z1`.

6. Weight and Bias Update (Gradient Descent): The gradients of the loss function with respect to the weights and biases are calculated as `dW1`, `db1`, `dW2` and `db2`. The weights and biases are then updated by subtracting the product of the learning rate and the corresponding gradient. This process is repeated for a specified number of epochs.

7. Prediction: The `predict` method performs a forward pass and then returns the class with the highest probability as the prediction for each input example.

After training the MLP, it is used to make predictions on the training set, and the accuracy of these predictions is calculated.

We first set the parameters of our model, namely the input size (based on our training data), the output size (10, corresponding to the 10 classes of digits in the MNIST dataset), the learning rate, and the number of epochs for training. We then define a list of different sizes for the hidden layer of the MLP that we want to evaluate: 32, 128, and 1024 neurons. For each size in this list, we initialize a new MLP model and train it using our training data. We then make predictions on both the training and test data, calculate the accuracy of these predictions, and print these accuracies out. The confusion matrices for the predictions on both the training and test data are also calculated and printed. This process is repeated for each hidden layer size, allowing us to compare the performance of models with different hidden layer sizes.

Results:

```
Train accuracy: 0.9229821428571429
Train confusion matrix:
[[5388    0   30   13   12   29   35    6   43    4]
 [   0 6096   29   21    9   17   10   17   70    8]
 [  52   46 5059   58   77    8  101   68  128   13]
 [  21   27  113 5108    3  193   22   71   98   52]
 [  10   30   27    3 5175    5   56    6   24  193]
 [  78   27   29  175   58 4360   97   16  150   50]
 [  49   19   50    1   43   52 5235    0   31    0]
 [  30   43   77   19   70    8    1 5404   12  126]
 [  27  105   63  120   27  123   40   20 4877   66]
 [  46   27   16   75  184   37    5  130   33 4985]]

Test accuracy: 0.9202142857142858
Test confusion matrix:
[[1299    0    6    1    4    9   12    3    9    0]
 [   0 1552    3   14    1    6    1    5   16    2]
 [  12   12 1241   14   21    2   28   19   29    2]
 [   4    5   26 1288    1   43    9   19   25   13]
 [   4    6    9    2 1197    0   12    2    3   60]
 [  15    7    3   57   14 1109   17    2   39   10]
 [   7    6   18    0   18   18 1324    0    5    0]
 [  11   11   22    3   11    3    0 1409    2   31]
 [   6   26   17   47    9   34   16    8 1179   15]
 [  12    9    7   15   41    6    0   34   11 1285]]

Training MLP with hidden layer size 128
```

```
Train accuracy: 0.9217321428571429
Train confusion matrix:
[[5383    0   20   14    9   36   36    7   50    5]
 [   2 6090   33   18    8   28    8   18   63    9]
 [  44   43 5039   78   90   16   79   68  132   21]
 [  18   22  117 5103    3  203   23   66  102   51]
 [  10   27   28    4 5153    3   63    8   30  203]
 [  74   26   33  170   55 4372   95   18  143   54]
 [  38   19   44    2   47   61 5235    1   33    0]
 [  27   40   71   20   51    7    3 5393   11  167]
 [  28  108   56  114   23  138   45   13 4872   71]
 [  42   28   16   70  168   35    4  157   41 4977]]

Test accuracy: 0.9199285714285714
Test confusion matrix:
[[1299    0    5    1    3    7   14    1   12    1]
 [   0 1547    4   13    1    9    1    5   18    2]
 [   7   15 1237   13   21    5   29   19   30    4]
 [   3    7   30 1285    1   44    9   18   22   14]
 [   4    4    8    3 1195    0   13    4    6   58]
 [  13    8    4   53   14 1113   14    4   40   10]
 [   3    5   15    0   18   23 1329    0    3    0]
 [   8    9   23    5   11    5    0 1407    2   33]
 [   7   25   10   45    7   36   17    8 1190   12]
 [   9   10    7   14   41    9    0   44    9 1277]]
```

```
Train accuracy: 0.9171071428571429
Train confusion matrix:
[[5381    0   21   16    9   36   38    6   48    5]
 [   2 6091   31   16    5   30    7   16   71    8]
 [  39   50 4999   85   93   24   77   73  142   28]
 [  19   24  125 5076    3  214   22   66  100   59]
 [  11   27   33    4 5138    3   63    9   35  206]
 [  74   34   40  177   59 4336  107   21  140   52]
 [  43   20   45    2   49   60 5214    7   38    2]
 [  21   38   77   21   60    7    3 5358   13  192]
 [  34  114   64  129   23  144   42   21 4822   75]
 [  36   28   18   75  172   33    4  185   44 4943]]
```

```
Test accuracy: 0.9148571428571428
Test confusion matrix:
[[1295    0    5    1    3   11   14    2   11    1]
 [   0 1540    5   12    1   10    1    4   24    3]
 [   6   14 1225   18   17    6   28   23   36    7]
 [   5    7   32 1278    1   46    9   18   22   15]
 [   4    4    7    5 1194    1   14    5    6   55]
 [  12    8    6   57   12 1107   15    3   43   10]
 [   4    5   12    1   16   21 1333    1    2    1]
 [   8   10   24    6   11    5    0 1398    2   39]
 [   8   28   13   52    7   37   17    7 1173   15]
 [   8   10    9   14   45    7    0   50   12 1265]]
```

MLP Result Discussion:

1. Model Performance: All models - MLPs with hidden layer sizes 32, 128, and 1024 - perform well with high accuracy on both the training and testing datasets, indicating that they all are able to capture the underlying patterns in the data. The test accuracy ranges from 91.48% to 92.02%, which is quite high.

2. Model Complexity vs Performance: As we increase the size of the hidden layer from 32 to 128, the accuracy on the test dataset increases slightly. However, further increasing the size to 1024 does not yield an improvement. In fact, the test accuracy decreases slightly, which could suggest that the model is becoming overly complex and thus starts to overfit on the training data.

3. Loss Reduction: The loss function decreases significantly for all models over the training epochs, indicating that the models are learning and improving their ability to predict the target variable. However, the rate of decrease is higher for the models with larger hidden layers, suggesting they learn more quickly.

4. Confusion Matrix Insights: The diagonal elements of the confusion matrix represent correctly classified instances, and off-diagonal elements represent misclassifications. A glance at the confusion matrices shows that all the models are doing a decent job at classifying most instances correctly. There are certain misclassifications, but they are relatively low.

5. Best Model Selection: Given these results, the MLP with a hidden layer size of 32 or 128 seems to be a better choice. It provides a good balance between model complexity (and thus computational efficiency) and prediction accuracy. The larger model with 1024 hidden units doesn't seem to provide any benefit in terms of accuracy, and might just be more computationally expensive to train and use.

KNN Implementation:

1. The first step is to define the K-Nearest Neighbors (KNN) class. This class has three main methods: `fit`, `predict`, and `_predict`. The `fit` method simply stores the training data and labels. The `predict` method applies the `_predict` method to each data point in the provided dataset. The `_predict` method calculates the Euclidean distance between a given data point and each data point in the training set, identifies the k data points in the training set closest to the given data point, and returns the most common label among these k neighbors.

2. Next, the MNIST dataset is loaded using the `datasets.load_digits()` method from `sklearn`.

3. The data and targets from the loaded dataset are assigned to `X` and `y` respectively.

4. The data is then split into training and test sets using the `train_test_split` function from `sklearn`. The test size is set to 0.2, meaning that 20% of the data will be used for testing and the rest for training. The `stratify` parameter is set to `y`, ensuring that the proportion of different classes in both the training and test sets is the same as the original dataset.

5. A list of different k values to be tested is defined.

6. For each value of k in the list, a KNN classifier is initialized with the current value of k, and the classifier is fit to the training data.

7. The classifier then predicts the labels of the test set.

8. The accuracy of the classifier is computed by comparing the predicted labels to the true labels of the test set using the `accuracy_score` function from `sklearn`.

9. The confusion matrix is also computed using the `confusion_matrix` function from `sklearn`, providing a more detailed view of the classifier's performance.

10. The results (accuracy and confusion matrix) for each value of k are printed out. This process is repeated for each k value, allowing the performance of the KNN classifier to be evaluated for different numbers of neighbors.

KNN Results:

```
Results for k=3:
Accuracy: 0.9861111111111112
Confusion Matrix:
[[36  0  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0  0 34  1  0  0  0  0  0  0]
 [ 0  0  0 37  0  0  0  0  0  0]
 [ 0  0  0  0 36  0  0  0  0  0]
 [ 0  0  0  0  0 36  0  0  0  1]
 [ 0  0  0  0  0  0 36  0  0  0]
 [ 0  0  0  0  0  0  0 35  0  1]
 [ 0  2  0  0  0  0  0  0 33  0]
 [ 0  0  0  0  0  0  0  0  0 36]]
```

```
Results for k=5:
Accuracy: 0.9916666666666667
Confusion Matrix:
[[36  0  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0  0 34  1  0  0  0  0  0  0]
 [ 0  0  0 37  0  0  0  0  0  0]
 [ 0  0  0  0 36  0  0  0  0  0]
 [ 0  0  0  0  0 37  0  0  0  0]
 [ 0  0  0  0  0  0 36  0  0  0]
 [ 0  0  0  0  0  0  0 36  0  0]
 [ 0  2  0  0  0  0  0  0 33  0]
 [ 0  0  0  0  0  0  0  0  0 36]]
```

```
Results for k=7:
Accuracy: 0.9916666666666667
Confusion Matrix:
[[36  0  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0  0 34  0  0  0  0  1  0  0]
 [ 0  0  0 37  0  0  0  0  0  0]
 [ 0  0  0  0 36  0  0  0  0  0]
 [ 0  0  0  0  0 37  0  0  0  0]
 [ 0  0  0  0  0  0 36  0  0  0]
 [ 0  0  0  0  0  0  0 36  0  0]
 [ 0  2  0  0  0  0  0  0 33  0]
 [ 0  0  0  0  0  0  0  0  0 36]]
```

```
Results for k=9:
Accuracy: 0.9916666666666667
Confusion Matrix:
[[36  0  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0  0 34  0  0  0  0  1  0  0]
 [ 0  0  0 37  0  0  0  0  0  0]
 [ 0  0  0  0 36  0  0  0  0  0]
 [ 0  0  0  0  0 37  0  0  0  0]
 [ 0  0  0  0  0  0 36  0  0  0]
 [ 0  0  0  0  0  0  0 36  0  0]
 [ 0  2  0  0  0  0  0  0 33  0]
 [ 0  0  0  0  0  0  0  0  0 36]]
```

KNN Result Discussion:

1. Effect of K-Value: As the value of 'k' increased from 3 to 9, we can see that the accuracy of the K-Nearest Neighbors model improved slightly. The accuracy was lowest for k=3 at 98.61% and it increased to 99.17% for k=5, k=7, and k=9. This indicates that the model with a higher 'k' value is more effective at classifying the digit images in this dataset, up to a certain point.

2. Best Performance: The KNN model achieved the best performance when k=5, k=7, and k=9, with an accuracy of 99.17%. At these k values, the model made very few misclassifications.

3. Confusion Matrix: The confusion matrices show that the model's errors were generally consistent across different values of 'k'. For instance, for all values of 'k', the digit '8' was occasionally misclassified as '1'. The digit '2' was sometimes misclassified as '3' for k=3 and k=5 but this error didn't appear for k=7 and k=9. Also, the digit '5' was misclassified as '9' for k=3 but this error was corrected for higher values of 'k'.

4. Misclassifications: Despite the high accuracy of the KNN models, there were still some misclassifications. For example, the digit '8' was occasionally misclassified as '1', especially for k=3, k=5, k=7, and k=9. It is possible that these misclassifications are due to similarities in the digit shapes, especially when the digits are not clearly written.

5. Overall Performance: Overall, the K-Nearest Neighbors algorithm performed very well on the MNIST dataset. It demonstrated a high level of accuracy across different 'k' values, indicating that it is an effective tool for digit classification tasks. However, the optimal 'k' value seems to be around 5, as increasing it further did not yield any additional improvements in accuracy.

CNN Implementation:

1. Import Necessary Libraries: The first step is to import the necessary modules and libraries. In this case, TensorFlow is used as the main library for building and training the model. The Keras API within

TensorFlow is used for constructing the neural network. Other libraries like sklearn.metrics and numpy are used for evaluating the model and handling arrays, respectively.

2. Load and Preprocess Dataset: The MNIST dataset is loaded using the `datasets.mnist.load_data()` function provided by Keras. The images are normalized by dividing the pixel values by 255, bringing them into the range [0, 1]. The images are also reshaped to have an extra dimension, representing the single color channel in the grayscale images.

3. Define Model Architecture: The model is built using the `Sequential` model API from Keras. It starts with a convolutional base, consisting of alternating Conv2D and MaxPooling2D layers. The Conv2D layers learn local patterns in the data, while the MaxPooling2D layers reduce the spatial dimensions of the data, helping to control overfitting. The output of the convolutional base is flattened and passed through two dense layers. The final dense layer has 10 units, corresponding to the 10 classes in the MNIST dataset.

4. Compile Model: The model is compiled with the Adam optimizer, the Sparse Categorical Crossentropy loss function (because the labels are integers, not one-hot encoded vectors), and accuracy as the performance metric.

5. Train Model: The model is trained on the training data for 5 epochs.

6. Evaluate Model: The model's performance is evaluated on the test data using the `evaluate` method, which returns the loss value and metrics values for the model. The test accuracy is printed out.

7. Compute Confusion Matrix: Predictions are made on the test images using the `predict` method of the model. The predictions are probabilities for each class, so the class with the highest probability is chosen as the predicted class using `np.argmax`. The confusion matrix is computed by comparing the predicted classes to the true test labels using the `confusion_matrix` function from sklearn. The confusion matrix is then printed out.

CNN Result:

```
Test accuracy: 0.9900000095367432
313/313 [==============================] - 1s 3ms/step
Confusion Matrix:
 [[ 977    0    0    0    0    0    1    2    0    0]
 [   0 1127    2    0    0    0    1    4    0    1]
 [   1    1 1021    0    1    0    0    8    0    0]
 [   0    0    1 1005    0    1    0    3    0    0]
 [   0    0    1    0  974    0    1    0    0    6]
 [   0    0    1   15    0  874    2    0    0    0]
 [   1    4    0    0    3    1  949    0    0    0]
 [   0    1    3    1    0    0    0 1020    0    3]
 [   4    0    3    0    0    2    0    3  957    5]
 [   0    0    0    0    3    3    1    6    0  996]]
```

CNN Result Discussion:

1. Accuracy and Loss: The Convolutional Neural Network (CNN) trained on the MNIST dataset achieved a high accuracy of 99.00% on the test set after 5 epochs, which is slightly higher than the best performance of the K-Nearest Neighbors (KNN) model. The loss decreased over each epoch, which suggests that the model was effectively learning and improving its predictions over time.

2. Confusion Matrix: From the confusion matrix, it can be seen that the CNN model made very few misclassifications. Most of the digits were correctly identified with only a few exceptions. For instance, the digit '5' was sometimes misclassified as '3', and the digit '8' was occasionally misclassified as '0', '5', or '9'. These minor errors could be due to similarities in the shapes of these digits or variations in handwriting.

3. Comparison with KNN: When comparing the results of the CNN to those of the KNN, the CNN model had a slightly better performance. The CNN had a higher accuracy and fewer misclassifications, which could be due to the fact that CNNs are better suited for image classification tasks. They can detect local patterns in images, such as edges or shapes, which might give them an advantage over KNNs for this specific task.