Implementation of Artificial Neural Network to classify handwritten digits

CSC3034 Computational Intelligence | Bachelor's (Hons) of Computer Science

Assignment 2

Prepared by:

Lee Boon Hoe (21024179)
Liong Zhen Yu (21017827)
Lim Zheng Jie (21058045)
Asilbek Abdullaev (20053286)

Table of Contents

Introduction

The Purpose of Applying a Neural Network in MNIST Dataset

The main purpose of using neural networks on the Kaggle MNIST dataset is to create a model that can correctly recognize handwritten digits. This task is crucial in the field of machine learning as it challenges the model to recognize the complex patterns and features that are present in the image in an intricate manner. Neural networks are well suited for this task because they can break down the digits into distinct features by learning a layered representation of the data, thus effectively capturing the basic structure of handwritten digits.

In the case of the MNIST dataset, the neural network can learn to identify the features of each handwritten digit. For example, the neural network can learn that the number 0 is usually round-shaped, while the number 1 typically appears slenderer in its handwritten form. Once the neural network learns these features, it can use them to recognize and classify new handwritten digits.

Also, neural networks are better than most other AI models at classifying images. This is because the images themselves are represented by numbers for each pixel, which complies with how neural networks process information. It is easy to normalize each pixel value to the range between 0 and 1 since each neuron determines its degree of activation by a continuous value between 0 and 1, so there will not be any issues of scale difference after normalization.

Dataset

The Source of Dataset

Kaggle is a website where data scientists and machine learning enthusiasts share their knowledge and help each other with their projects. A lot of published datasets can be found and used to create AI models of your own. The URL of this specific is https://www.kaggle.com/datasets/hojjatk/mnist-dataset.

Dataset Description

Our group is using the MNIST Dataset to create the neural network. MNIST Dataset is a widely used dataset for handwritten digit classification. It contains 60,000 training images and 10,000 testing images of handwritten digits from 0 to 9. The images are all 28x28 pixels in size, and each pixel is represented by an 8-bit grayscale value.

Model Architecture

The Architecture of The Neural Network

The chosen neural network architecture is composed of three key layers. The input layer, represented by `Dense(784, input_shape=(784,), activation='relu')`, has 784 neurons, corresponding to the flattened pixel values of the 28x28 images. The activation function used here is the Rectified Linear Unit (ReLU), a popular choice for introducing non-linearity into the model.

The hidden layer, defined as `Dense(100, activation='relu')`, consists of 100 neurons. This layer enables the neural network to learn complex hierarchical representations from the input data. The ReLU activation function is again employed to introduce non-linearity and capture intricate features.

To combat overfitting, a Dropout layer is incorporated with a dropout rate of 0.25. This is implemented using `Dropout(0.25)` after the hidden layer. Dropout randomly deactivates a fraction of neurons during each training iteration, preventing the network from becoming overly dependent on specific features and enhancing its ability to generalize to new data.

Finally, the output layer, defined as `Dense(10, activation='softmax')`, consists of 10 neurons representing the ten digit classes (0 through 9). The Softmax activation function is applied, converting the raw output into probability distributions. This allows the model to provide class probabilities, facilitating multi-class classification.

The Justification of the Architecture

The architecture is intended to find a balance between complexity and capability. The input layer corresponds to the flattened picture dimensions, while the hidden layer with ReLU activation serves as a feature extractor. The Dropout layer addresses overfitting problems by providing randomness during training. Softmax activation in the output layer is appropriate for multi-class classification since it provides normalized class probabilities.

Training and Implementations

Data Processing

Data preprocessing involves two crucial steps. First, normalization is performed on the pixel values of the images, scaling them to a range between 0 and 1. This is achieved with the code snippet

```python
X_train = X_train / 255
X_test = X_test / 255
```

Second, the images are flattened from 2D arrays to 1D arrays using

```python
X_train_flattened = X_train.reshape(len(X_train), 28*28)
X_test_flattened = X_test.reshape(len(X_test), 28*28)
```

This step transforms the image data into a format suitable for input into the neural network.

Model Compilation

The model is compiled using the Adam optimizer, chosen for its efficiency in updating the model weights during training. The loss function selected is `sparse_categorical_crossentropy`, fitting the multi-class classification nature of the task. The `metrics` parameter is set to 'accuracy' to monitor the model's performance during training.

```python
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Model Training

The training process involves iterating over the dataset for a specified number of epochs. In this case, the model is trained for 10 epochs using the fit method. A validation split of 20% `(validation_split=0.2)` is employed, allocating a portion of the training data for validation. This split helps monitor the model's performance on unseen data during training.

```python
history = model.fit(X_train_flattened, y_train, epochs=10,
validation_split=0.2)
```

Stopping Criteria

The training process was carefully monitored to identify the optimal number of iterations, balancing between model performance and preventing overfitting. By inspecting the training accuracy and validation accuracy diagrams plotted over epochs, it became evident that the training accuracy approached approximately 98% and displayed minimal improvement beyond 10 iterations.

Additionally, examining the confusion matrix using a higher number of epochs hinted at potential overfitting, with the model becoming too specialized to the training data. As a result, stopping the training at 10 iterations emerged as an optimal choice, preventing overfitting and ensuring a model that generalizes well to unseen data.

Results and Analysis

First, we look at the result of the training log that illustrates the training and validation performance across 10 iterations for the Neural Network to ensure that it is working correctly as expected:

```
Epoch 1/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.2382 - accuracy: 0.9284 - val_loss: 0.1177 - val_accuracy: 0.9653
Epoch 2/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.1028 - accuracy: 0.9692 - val_loss: 0.1017 - val_accuracy: 0.9697
Epoch 3/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.0696 - accuracy: 0.9786 - val_loss: 0.0945 - val_accuracy: 0.9732
Epoch 4/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.0534 - accuracy: 0.9835 - val_loss: 0.0824 - val_accuracy: 0.9762
Epoch 5/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.0424 - accuracy: 0.9865 - val_loss: 0.1011 - val_accuracy: 0.9733
Epoch 6/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.0329 - accuracy: 0.9891 - val_loss: 0.1031 - val_accuracy: 0.9745
Epoch 7/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.0309 - accuracy: 0.9902 - val_loss: 0.1092 - val_accuracy: 0.9741
Epoch 8/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.0271 - accuracy: 0.9911 - val_loss: 0.1022 - val_accuracy: 0.9766
Epoch 9/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.0237 - accuracy: 0.9927 - val_loss: 0.0975 - val_accuracy: 0.9793
Epoch 10/10
1500/1500 [==============================] - 10s 6ms/step - loss: 0.0185 - accuracy: 0.9940 - val_loss: 0.1069 - val_accuracy: 0.9784
```

Training Loss and Accuracy: The training loss steadily decreases from 0.2382 to 0.0185 over epochs, while the accuracy increases from 0.9284 to 0.9940. This signifies that the model is learning and improving its predictions on the training dataset. The decreasing loss indicates that the model is reducing its prediction errors.

Validation Loss and Accuracy: The validation loss starts at 0.1177 and fluctuates but remains relatively low, ending at 0.1069. Simultaneously, the validation accuracy starts at 0.9653 and reaches a peak of 0.9793 before settling at 0.9784. This suggests that the model performs well on unseen validation data, showcasing generalization ability.

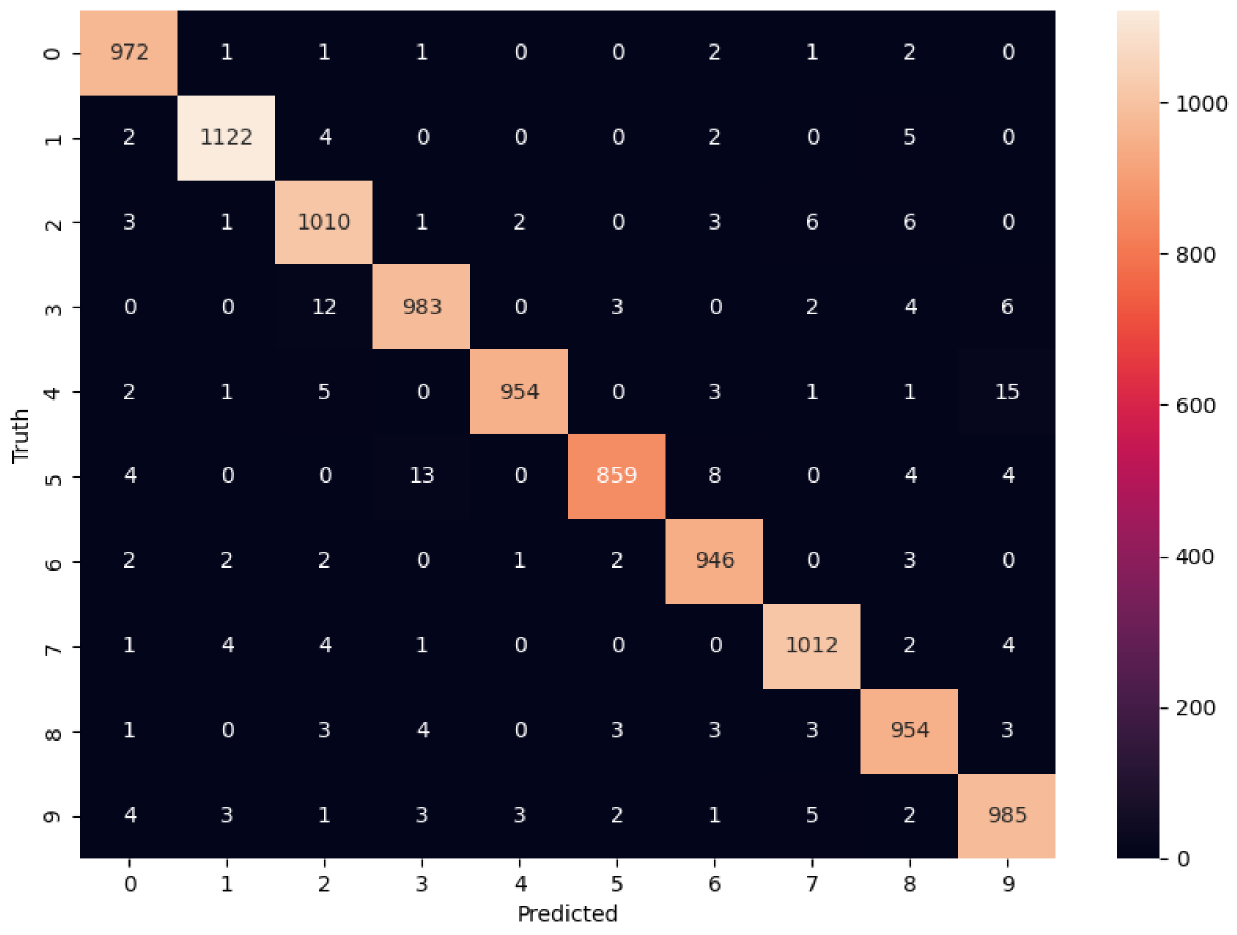
Thus, we can conclude that the neural network was effective in training the model with high accuracy without overfitting.

Next, the output of the Neural Network shows the results of evaluating the model with the test data.

```
#################################
Evaluating the model with test data

313/313 [==============================] - 1s 3ms/step - loss: 0.0910 - accuracy: 0.9797
```

Across 313 samples, it is denoted that 97.97% (accuracy) of predictions are correct against the truth (actual numbers), while 9.10% (low) suggests that the model's predictions are very close to the actual target values in the test dataset. Despite the high accuracy and low loss, we believe there is still much more room for improvement, and we should aim for higher accuracy like 99% and above. In real-world applications such as medication dosage, accurate digit recognition in prescription labels or medical records is essential. Misinterpreting numbers like an initial digit recognition error of 50ml and 10ml could potentially lead to critical implications, incorrect dosages and potentially impacting matters of life and death.



Lastly, the output from the Neural Network presents the Confusion Matrix visualized through a heatmap, showcasing the relationship between predicted and actual values derived from the model's predictions on the test dataset. This matrix offers a comprehensive overview of classification performance, aiding in the assessment of the model's accuracy and error tendencies across different classes. From here, we can analyze which digits are more difficult to classify by the training model. Noteworthy is the observation that the digits "4" and "5" demonstrate the highest false positive ratio (correct predictions out of all data), with 0.028 and 0.037, respectively. These findings suggest a need for focused adjustments within the Neural Network to enhance the accurate recognition of these specific digits. However, the overall Confusion Matrix remains consistent with anticipated results, depicting a predominance of correct predictions.

The comprehensive assessment underscores the necessity for further enhancements within the Neural Network to achieve higher accuracy and lower loss. Prospective adjustments might encompass adjusting the hidden layers, fine-tuning pivotal hyperparameters such as batch size, epochs, or activation functions, and increasing the training dataset through editing techniques like image rotation, flipping, or scaling. Unfortunately, due to time constraints, we are unable to conduct any more additional trial and error experiments. Thus, these are potential improvements to be explored in the future, should we ever have the opportunity to continue this project again.

Hybrid Model and Its Purpose

The selected hybrid model that is proposed to modify the existing neural network is evolutionary neural network architecture. It is a model which uses genetic algorithms to help neural networks learn. While neural networks already have their own learning algorithm of backpropagation, it is computationally expensive especially when there are a lot of parameters, genetic algorithm is much faster to train as it does not require any differentiation or linear algebra. Neural networks can have an unstable learning curve because of bad initialization of starting weights due to arrangement of data where the model converges very slowly. Using genetic algorithm to train improves the exploration capabilities within the solution space by applying random mutations to increase training performance. Also, it can learn the effectiveness of different hyperparameter configurations, which neural networks cannot do. Our proposed application will replace backpropagation for the learning algorithm.

The Application of Evolutionary Neural Network

In our case, we would like to learn the weights of each connection and the activation function for each layer. For simplicity purposes, a fully connected neural network is maintained and each neuron in a hidden layer will have the same activation function as the others in the same layer. The following will highlight the important steps and details about the evolution process.

1. Encoding:
   For each neural network, the weights can be represented by a series of groups of numbers. A group is all incoming weights going to the neuron, and the groups are in the order of top to bottom, left to right. An example of 4 neurons in order is shown below.

0.3	0.7	-0.2	0.8	-0.1	0.2	0.3	0.7	-0.6	-0.5

   As for the activation functions, we can use binary representation to encode. Our current hidden layers utilizes ReLu, but let's say we include other activation functions which are sigmoid, tanh, or a step function, the encoding will be 00, 01, 10, and 11 respectively. So, a neural network with 2 hidden layers will have 4 extra bits as 2 new groups, the output layer will utilize Softmax activation, so it does not need encoding. We can add these groups at the end of weight encoding.

2. Initialization:
   First create a population of the desired amount, each neural network will have a randomized set of weights within an interval and randomized bits for all the activation functions.

3. Fitness function:
   The fitness value of a neural network represents how good the configuration is at training the model. The higher the fitness, the higher the chance it is selected in the next generation. After feeding training data to the network, we can calculate the sum of squared errors and use its reciprocal as the fitness function.

4. Crossover:
   After selecting the desired number of parents in the next generation, each pair of parents will perform a crossover by choosing a group from the two parents for each group. This will result in two children with a combination of both parents.

5. Mutation:
   For each group, it will have a very small chance to mutate and change its value. If it is a gene for weight, it will be mutated by adding a small number between –1 and 1; and if it is a gene for activation function, the bits are mutated by flipping between 0 and 1.

6. Termination loop:
   A loop will consist of crossover, mutation, and selection of children for the next generation by calculation of fitness value. We can either terminate this loop by setting a maximum number of iterations or terminate if there are no significant improvements over consecutive generations.

Performing all the steps above, at the end of the latest generation, the algorithm will have found the population containing an NN with the optimum hyperparameters and weights. It will be the fittest NN of all in the population.

Code (Google Collab):

We also have put our code snippet into the Google Collab environment, it can be accessed through this link:
Ci assignemnt 2.ipynb