DATA130011.01 Neural Network and Deep Learning Project 1

吴嘉骜 21307130203

May 7, 2024

Abstract

This report presents the implementation of handwritten digit classifiers using a Fully Connected Neural Network (FCNN), into which different features are incorporated to improve the model's performance. The classifier is implemented in Python, without using any deep learning libraries. The activation function is `tanh` for hidden layers and the optimizer is stochastic gradient descent (SGD) with Momentum. The best test error on this dataset is 0.1180 when the settings are: only one hidden layer with 10 units, a learning rate decay with momentum, the cross-entropy loss function, and a bias added to every hidden layer. In this report, we present modification details and analyze experiment results for 10 tasks. The source code consists of ten Jupyter notebooks, each corresponding to a task.

Project description

The objective of this project is to investigate handwritten digit classification based on Neural Networks. A demo code is provided, and the task is to modify this training procedure to optimize performance.

Experiment environment

Windows 11 VsCode Python 3.8.18

1 Introduction

In this project, we will construct a neural network from scratch to classify handwritten digits. We first introduce the dataset and the neural network architecture provided by the demo codes. Then we present the modifications for each task and the corresponding results.

1.1 Dataset

The dataset comprises 5000 training images, 5000 validation images, and 1000 test images. Each image is a 16×16 grayscale depiction of a handwritten digit (0-9). It is in the format of `.mat` files, which can be loaded using the `scipy.io.loadmat` function in Python.

1.2 Neural Network of the Demo Code

The demo code provides a fully connected neural network with adjustable number of hidden layers. The original network is a two-layer neural network with 10 hidden units in the hidden layer. The activation

function for the hidden layers is `tanh`, the loss function is the squared error, and the optimizer is stochastic gradient descent (SGD, 1 sample per iteration) with a fixed learning rate. The structure of a general neural network is sketched in Figure 1.

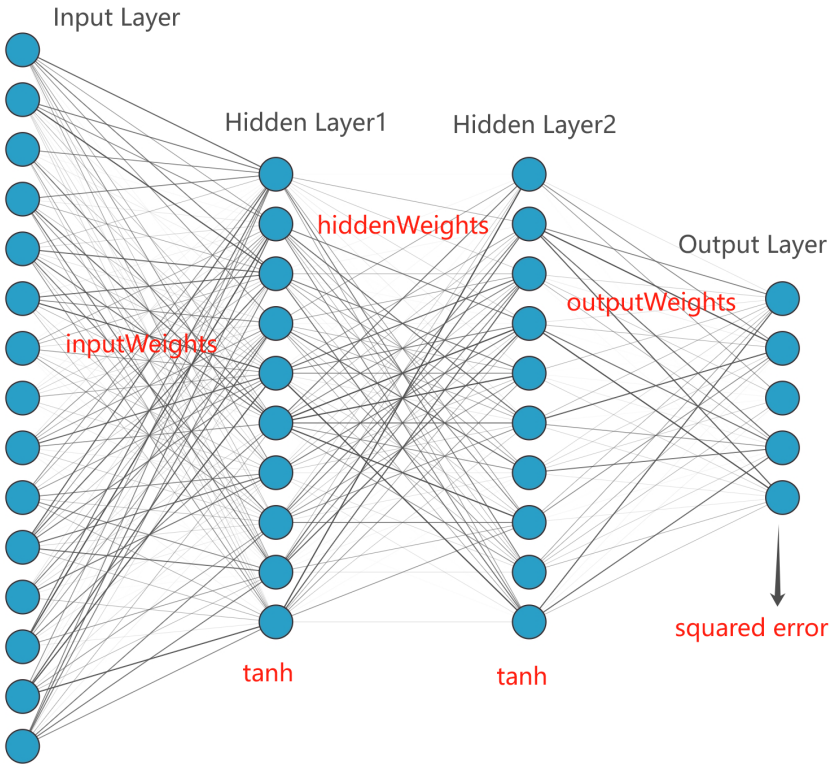

Figure 1: Three-layer neural network architecture

2 Tasks

A total of ten tasks are proposed to improve the performance of the neural network classifier. We utilize the demo code as a basic structure and make modifications according to the requirements of each task. The implementation follows the standard procedure of training a neural network, referring to the course materials and the textbook [1].

2.1 Task 1: Change the network structure

Task description: Change the network structure: the vector `nHidden` specifies the number of hidden units in each layer.

The original network has only one hidden layer with 10 hidden units. We try to change both the number of hidden layers and the number of hidden units in each layer. A brief result summary is shown in Table 1. We observe that validation error decreases as the number of hidden units increases or the number of layers increases. The increase of unit numbers affects the result more significantly. However, as the structure becomes more complex, the training time increases as well. So in next tasks, we will mainly focus on training the model the same as the original one, except for some specific situations.

Table 1: Neural Network Training Results (100000 iterations)

nHiddens	Last Validation Error	Test Error
[10]	0.2250	0.2130
[128]	0.1740	0.1670
[10, 10]	0.2274	0.2190
[128, 128]	0.1540	0.1650
[10, 10, 10]	0.2912	0.2960
[128, 128, 128]	0.1426	0.1330

2.2 Task 2: Change the learning rate

Task description: Change the training procedure by modifying the sequence of step-sizes or using different step-sizes for different variables (using momentum).
We first implement the learning rate decay, which is a common technique to improve the convergence of the model. The formula is given by

$$\text{lr} = \text{lr} \times \frac{1}{1 + \text{decay} \times \text{iteration}}$$

where lr is the initial learning rate, decay is the decay rate, and iteration is the current iteration number. We set the decay rate to 1e-5 and try different initial learning rates. The results are shown in Table 2.

Table 2: Training Results by Initial Step Size (100000 iterations)

Initial Step Size	Last Validation Error	Test Error
0.1	0.9012	0.9000
0.01	0.2146	0.2300
0.001	0.2028	0.2200
1e-4	0.5156	0.5030
1e-5	0.8210	0.8350

We notice that an initial learning rate of 0.01 or 0.001 is good for this problem.
Then we implement the learning rate decay with momentum. The formula is given by

$$w^{t+1} = w^t - \text{lr} \times \nabla L(w^t) + \text{momentum} \times (w^t - w^{t-1})$$

where momentum is the momentum strength. The results of setting different momentum strengths are shown in Table 3.
We observe that the momentum strength of 0.9 is the best choice. From this task on, we will adopt 0.001 as the initial learning rate and 0.9 as the momentum strength.

2.3 Task 3: Vectorize and speed up the code

Task description: Vectorize evaluating the loss function (e.g., try to express as much as possible in terms of matrix operations), to do more training iterations in a reasonable amount of time.

Table 3: Training Results by Momentum (100000 iterations)

Momentum	Last Validation Error	Test Error
0.5	0.2138	0.2140
0.6	0.2024	0.2110
0.7	0.2094	0.2030
0.8	0.2336	0.2320
0.9	0.1920	0.1840

We improve the code by utilizing matrix operations as much as possible. The main modification is to replace the loop for instances with batch operations, so in prediction and backpropagation, we can calculate the results for all instances in one step. We count the time for both versions and find that the vectorized version is 5-7 times faster than the original one. This is a remarkable improvement, and we will use the faster version in the following tasks.

2.4 Task 4: Add regularization or apply early stopping

Task description: Add $l2$ regularization (or $l1$ regularization) of the weights to your loss function. An alternate form of regularization that is sometimes used is early stopping, which is stopping training when the error on a validation set stops decreasing.

From this task, to check whether the model is overfitting, we print both the training and validation losses. We first implement the early stopping technique, and we set a hyperparameter `patience` which is the number of iterations to wait before stopping training if the validation loss does not decrease. Another modification is that we retain the best weight with the lowest validation loss. In the experiment, we set the patience to 5, and Early Stopping is triggered after 80000 iterations with a validation error 0.1976. The test error with final model is 0.2110.

Then we try the $l2$ regularization. We set different regularization strengths and the results are shown in Table 4. We observe that the regularization strength of 0.01 is the best choice, and large Lambdas lead to underfitting.

Table 4: Training Results by L2 Lambda (100000 iterations)

Regularization Lambda	Last Training Error	Last Validation Error	Test Error
1.0	0.3880	0.3938	0.3860
0.5	0.2006	0.2064	0.2100
0.2	0.1962	0.2164	0.1940
0.1	0.1602	0.1722	0.1650
0.01	0.1378	0.1658	0.1420
0.001	0.1714	0.1938	0.1420

2.5 Task 5: Use Softmax and the Cross-Entropy Loss Function

Task description: Instead of using the squared error, use a softmax (multinomial logistic) layer at the end of the network so that the 10 outputs can be interpreted as probabilities of each class. Replace squared error with the negative log-likelihood of the true label under this loss.

We implement the softmax function and the cross-entropy loss function. An advantage of combining softmax and cross-entropy loss is that it can provide a more accurate and easy-computing gradient. Setting hyperparameters as previous tasks described, and after 100000 iterations we gain a better test error of 0.1310 compared to the squared error loss function. The training process is plotted in Figure 2. We observe that the model is a little bit overfitting, but the accuracy is still good.

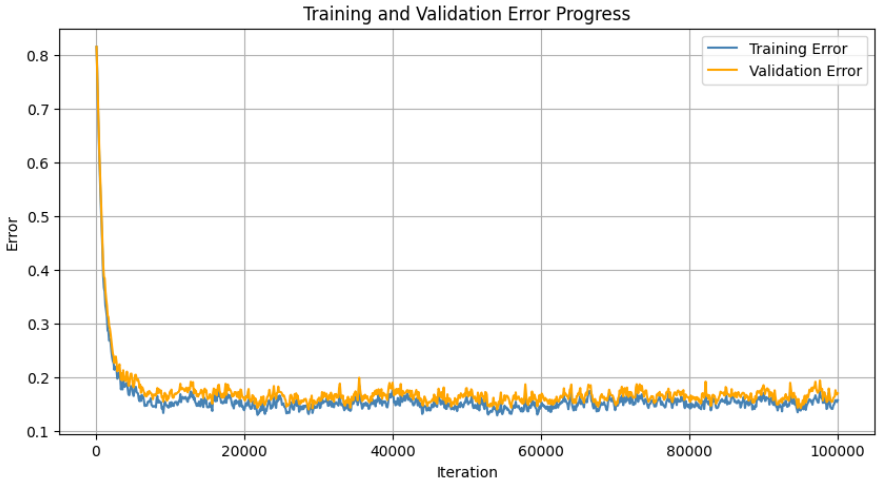

Figure 2: Training and validation errors with cross-entropy loss function

2.6 Task 6: Add a bias to every hidden layer

Task description: Instead of just having a bias variable at the beginning, make one of the hidden units in each layer a constant, so that each layer has a bias.

This implementation is straightforward but we have to be careful with the dimensions of weights. We add 1 to the last dimension of weights in each hidden layer, but the output of each layer remains the same. This indicates that whenever multiplying `hiddenWeight` with `activation`, we should stack a column of 1s to the latter. And the backpropagation part is much more intricate. After 100000 iterations, the test error is 0.1180, which is the best result so far. The training process is plotted in Figure 3, similar to the previous task.

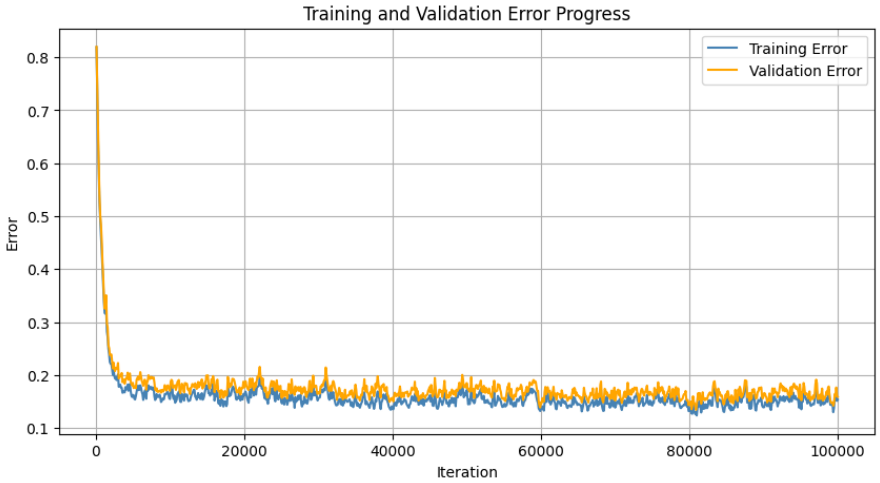

Figure 3: Training and validation errors with biases in hidden layers

2.7 Task 7: Implement Dropout

Task description: Implement "dropout", in which hidden units are dropped out with probability p during training. A common choice is p = 0.5.

We implement the dropout technique in the training process. The dropout rate is set to 0.5 as suggested. A dropout mask is generated for each hidden layer, and the forward and backward propagation are modified accordingly. Each mask is divided by the dropout rate to keep the expected value of the output the same. We gain a test error of 0.1210 after 100000 iterations with task 6 settings. The training process is plotted in Figure 4. The overfitting problem is slightly alleviated, but the errors become more fluctuating.

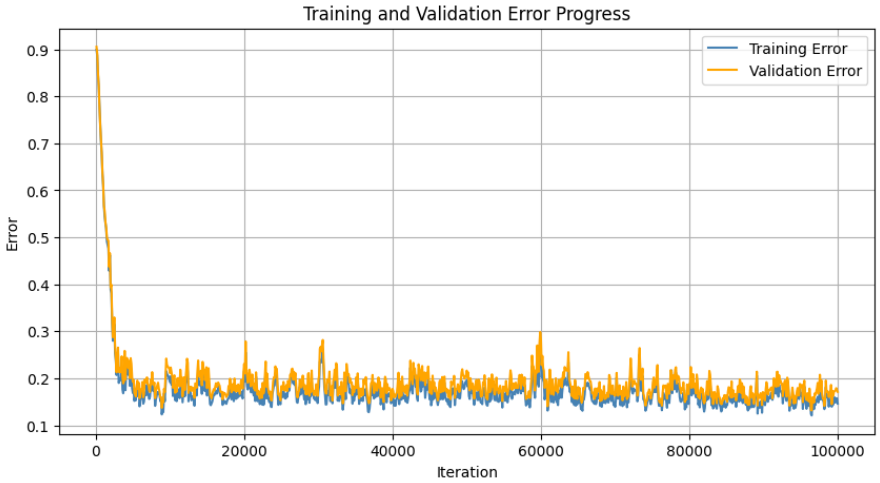

Figure 4: Training and validation errors with dropout

2.8 Task 8: Fine-tune the last layer

Task description: Do 'fine-tuning' of the last layer. Fix the parameters of all the layers except the last one, and solve for the parameters of the last layer exactly as a convex optimization problem. E.g., treat the input to the last layer as the features and use techniques from earlier in the course (this is particularly fast if you use the squared error, since it has a closed-form solution).

To gain a closed-form solution, we can use the squared error loss function, as used in Tasks 1-4. To simplify the problem, we only adopt the $l2$ regularization, without early stopping or dropout.

The last layer is treated as a linear regression problem, and we can solve it by the normal equation:

$$\min_W \frac{1}{2}||W^T A - y||^2 + \frac{\lambda}{2}||W||^2$$

where W is the weight matrix of the last layer, A is the input to the last layer, and y is the ground truth. The closed-form solution is $W = (A^T A + \lambda I)^{-1} A^T y$.

To compare the results, we first train the model with the last layer as a fully connected layer, and then fine-tune the last layer with the closed-form solution. The first model has a test error of 0.1490, and the fine-tuned model has a test error of 0.1510. We observe that the fine-tuned model has a slightly worse performance, which may be due to the overfitting problem.

2.9 Task 9: Augment data

Task description: Artificially create more training examples, by applying small transformations (translations, rotations, resizing, etc.) to the original images.

We implement image transformations, including translation, rotation, and scaling, to augment the training dataset. We generate 5000 new images by randomly applying these transformations to the original training images with ramdom parameters as well. The new training set is then used to train the model. The settings are the same as Task 6. But after some trails, we find that only 10 units in the hidden layer are not insufficient for this task, so we change the number of hidden units to 50.

After 100000 iterations, the test error is 0.1190, while the training error is 0.2178, and a training plot is shown in Figure 5. This is preculiar since the training error should be lower than the validation error. We guess that the artificial training data may be too noisy or different from the original data, but the test data are high-quality and clean, so the model performs better on the test data. This may indicate that the model is actually "learning" something rather than just memorizing the training data.

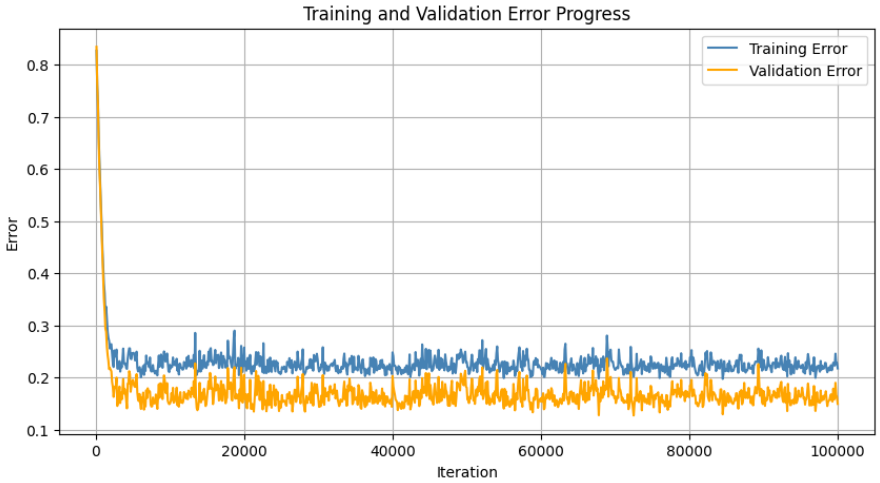

Figure 5: Training and validation errors with data augmentation

2.10 Task 10: Replace the first layer with a convolutional layer

Task description: Replace the first layer of the network with a 2D convolutional layer. Filters of size 5 by 5 are a common choice.

We implement a convolutional layer with 5x5 filters and stride 1. The output size is 12x12. We then flatten the output and feed it into the fully connected layer. In Python, we use `scipy.signal.convolve2d` to perform 2D convolution, similar to `conv2` in MATLAB. The settings are simpler this time and are the same as Task 4, except for the additional first convolutional layer and 50 hidden units in the fully connected layer. After 100000 iterations, the test error is 0.1900. The training process is plotted in Figure 6.

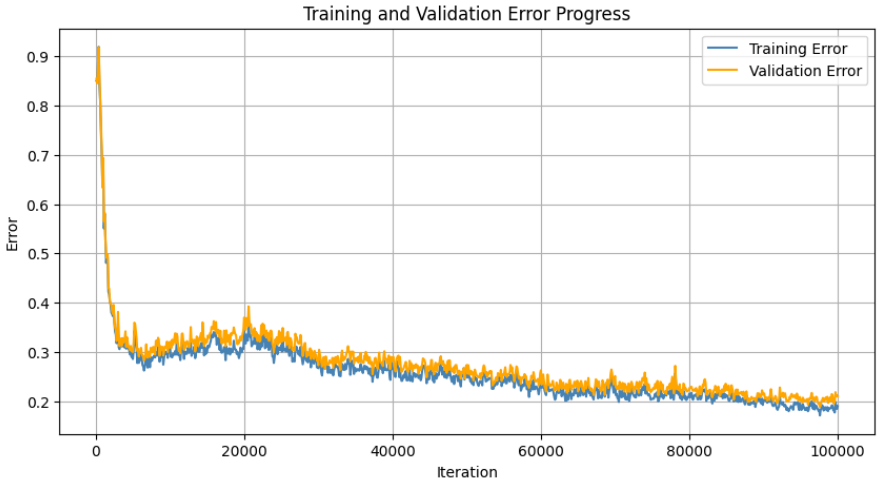

Figure 6: Training and validation errors with convolutional layer

Note that the training process is not as smooth as the previous tasks, which may be due to the convolutional layer's complexity. The test error is not as good as the previous tasks, which may be due to the lack of more convolutional layers or pooling layers.

3 Discussion

3.1 Conclusion

In this project, we have implemented a fully connected neural network to classify handwritten digits. We have explored various modifications to the network architecture, training procedure, and loss functions to improve the model's performance. The best test error achieved was 0.1180, which is a significant improvement over the initial model (test error of 0.4437). We have demonstrated the importance of hyperparameter tuning, regularization, and data augmentation in improving the generalization performance of the model. The source code is comprised of ten Jupyter notebooks, each corresponding to a task, and can be run independently to reproduce the results.

3.2 Limitations and future work

Despite the successes achieved by the current model, there are several areas that could benefit from further exploration and development:

1. **Neural Network Architecture:** The current model utilizes a relatively simple fully connected neural network. Future work could explore the addition of more hidden layers or the adoption of more complex architectures such as Convolutional Neural Networks (CNNs), which are particularly well-suited for image classification tasks.

2. **Optimizer:** While Stochastic Gradient Descent (SGD) with momentum was used in this project, other optimization algorithms like Adam might offer improvements in convergence rates and overall performance.

3. **Regularization Techniques:** The current model implementation focuses on L2 regularization. Future work could explore the use of other regularization techniques such as L1 regularization or combine comprehensive methods together to further improve generalization performance.

References

[1] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Cambridge, Mass: The MIT press.