Basics of Deep Learning - Neural Network with NumPy

Etai Zilberman, Bar Ara

Submitted as mid-semester project report for Basics of Deep Learning course, Colman, 2024

1 Introduction

Deep learning is a powerful subset of machine learning inspired by the structure of the human brain. It has become instrumental in solving complex problems such as image recognition, natural language processing, and autonomous driving. In this project, we aimed to develop a neural network to classify two classes of hand images (binary classification). The primary objective was to implement forward and back-propagation for training the model, with the restriction of using the NumPy library only, without using any other libraries such as PyTorch and scikit-learn (although a few imports from libraries like scikit-learn were allowed for tasks such as evaluating accuracy and other metrics). We also needed to select hyperparameters and evaluate the model's performance, using sigmoid as the activation function and binary cross-entropy as the loss function.

1.1 Data

The dataset was extracted from the file `/content/dataset_labels.npz`. It contains 5000 images of hands, 500 of each hand representing a number, and each labeled with one of ten classes from 0 to 9. For this project, we selected two classes to implement a binary classification. The data was split into training (80%) and testing (20%) sets, ensuring a balanced distribution of the two classes in both subsets.

1.2 Problem

The problem was to train a binary classification model to distinguish between the two chosen classes. The challenges included implementing the whole training phase, including forward and back-propagation from scratch, choosing appropriate hyperparameters (e.g., hidden layer neurons, learning rate, number of epochs), and ensuring the model generalized well to unseen data.

2 Solution

2.1 General Approach

The approach involved the following steps:

- Data preprocessing: Normalizing the pixel values of the images to a range of 0 to 1 and reshaping the data appropriately for the neural network.
- Data split: Splitting the dataset to 80% train set and 20% test set.
- Model architecture and training: Implementing feed-forward, loss calculation, back-propagation, and gradient descent.
- Evaluation: Measuring the model's loss and accuracy on the test dataset.

2.2 Design

The model was implemented in Python using Jupyter Notebook computing platform and the NumPy library. Some of the key design choices included:

- Activation function: Sigmoid function for hidden and output layers.
- Loss function: Binary cross-entropy for the two-class classification problem.
- Network architecture: Input layer with 784 features, one hidden layer with 64 neurons, and an output layer with 1 neuron.

2.2.1 Technical Challenges:

Implementing matrix operations for back-propagation and debugging shape mismatches were the primary challenges. Normalizing the dataset and splitting it into train/test sets, as well as finding the balance between the number of epochs, learning rate, and number of neurons in the hidden layer - also required careful attention.

2.2.2 Gradient Descent Algorithm

Algorithm 1 Gradient Descent for Binary Classification

```
for each epoch $i$ in $[0, \text{epochs})$ do
    Initialize avg_epoch_loss $\leftarrow 0$
    for each example $j$ in $[0, \text{num\_of\_examples})$ do
        Compute $Z_1 = W_1 X[:, j] + b_1$
        Apply activation: $A_1 = \sigma(Z_1)$
        Compute $Z_2 = W_2 A_1 + b_2$
        Apply activation: $A_2 = \sigma(Z_2)$
        Compute loss: $\mathcal{L} = -(Y_j \log(A_2) + (1 - Y_j)\log(1 - A_2))$
        Accumulate loss: avg_epoch_loss $\leftarrow$ avg_epoch_loss $+ \mathcal{L}$
        Compute gradients:
                $dZ_2 = (A_2 - Y_j) \cdot A_2 \cdot (1 - A_2)$
                $dW_2 = dZ_2 A_1^T$, $db_2 = \sum dZ_2$
                $dA_1 = W_2^T dZ_2$
                $dZ_1 = dA_1 \cdot A_1 \cdot (1 - A_1)$
                $dW_1 = dZ_1 X[:, j]^T$, $db_1 = \sum dZ_1$
        Update weights and biases:
                $W_2 \leftarrow W_2 - \alpha dW_2$, $b_2 \leftarrow b_2 - \alpha db_2$
                $W_1 \leftarrow W_1 - \alpha dW_1$, $b_1 \leftarrow b_1 - \alpha db_1$
    end for
    Compute average loss: avg_epoch_loss $\leftarrow \frac{\text{avg\_epoch\_loss}}{\text{num\_of\_examples}}$
    Append avg_epoch_loss to loss_list
    Append epoch index $i$ to epoch_list
end for
```

3 Base Model

The base model was implemented with a single hidden layer. The forward part involved computing the weighted sum of inputs, applying the sigmoid activation, and predicting probabilities. Back-propagation updated the weights and biases along the network.

3.1 BCE Loss Function

The binary cross-entropy loss function was chosen for this project as it is well-suited for binary classification tasks. It measures the difference between the predicted probabilities and the true labels, penalizing incorrect predictions more heavily the more they are far from each other. This choice aligns with the project's objective of accurately distinguishing between two classes.

3.2 Loss Graph

The loss graph provides a visual representation of the model's learning progress over epochs. A decreasing training loss indicates that the model is effectively learning the data patterns.

3.3 Training the Network

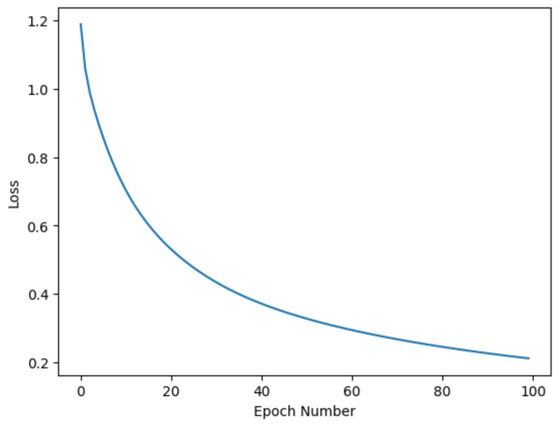
Training involved optimizing the weights and biases using back-propagation and gradient descent. This iterative process adjusted the parameters to minimize the loss function. Balancing the number of epochs and learning rate ensured the model converged to a good solution without overfitting or underfitting.

3.4 Network Architecture

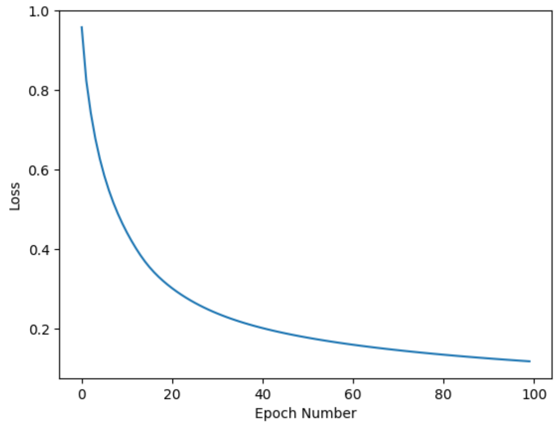
The neural network consisted of an input layer (each 28x28 image reshaped to a vector the size of 784), a single hidden layer (size 64), and an output layer - decision between 0 and 1. More hidden layers could be added on to the network, but for this task it seemed to be unnecessary, and one hidden layer was sufficient.

3.5 Hyperparameters

The hyperparameters, including the hidden layer size, learning rate, and number of epochs, were carefully selected to optimize network performance. The learning rate controlled the step size for parameter updates, balancing speed and stability in convergence. The difference between learning rates can be seen in Figure 1. The number of epochs ensured sufficient training without overfitting. These hyperparameter choices represented a trade-off between model complexity, accuracy, and training time.



(a) Learning Rate: 0.004



(b) Learning Rate: 0.009

Figure 1: Comparison of different learning rates: 0.004 (left) and 0.009 (right).

3.6 Activation Function

The sigmoid activation function outputs values in the range of 0 to 1, which is ideal for binary classification tasks as it can be interpreted as probabilities. This allows the model to output a probability for each class, with values close to 1 indicating a strong likelihood of one class and values close to 0 indicating a strong likelihood of the other class.

3.7 Performance on the Training and Test Set

The model achieved high accuracy of 94% on the test set, indicating that it performed reasonably well. This supported the project's aim of creating a model capable of accurately distinguishing visually similar classes under real-world conditions.

3.8 Training Times

Training times were kept reasonable by maintaining a simple architecture. This was critical for iterating on hyperparameter tuning and ensuring the project could be completed within the given time frame. With our architecture and selected hyperparameters, it took about 90 seconds from start to finish.

3.9 Results and Metrics

The results indicated that the model could effectively classify the selected challenging classes ('7' and '8', more about it in section 4). The confusion matrix provided insights into misclassifications, while metrics such as accuracy and loss confirmed that the model met the project's objectives. The model achieved an accuracy of `94%` on the test set. The result validated the chosen architecture, hyperparameters, and training process as appropriate for the task. Below are the results:

Metric	**Training Set**	**Test Set**
Accuracy	/	94%
Loss	0.1132	/

Table 1: Results of the Model

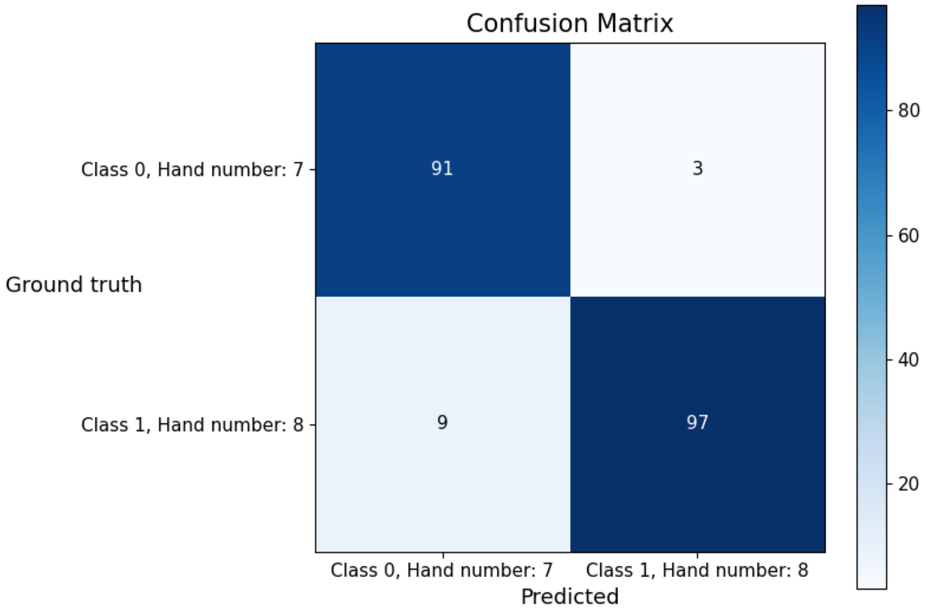
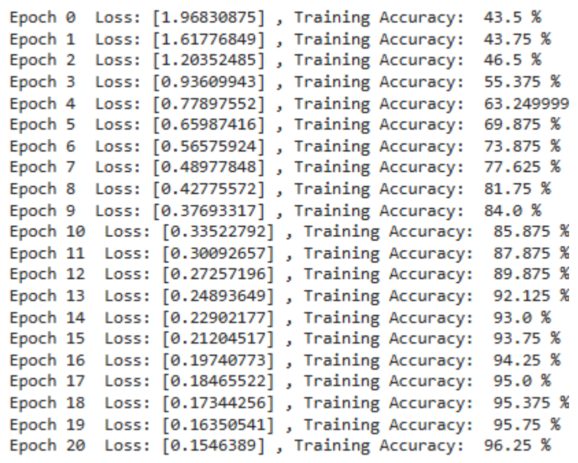


Figure 2: Confusion Matrix

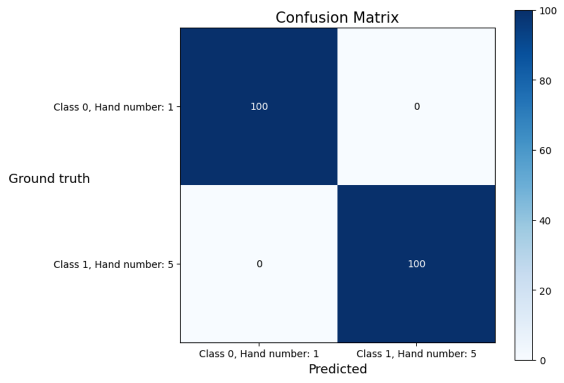
4 Discussion

For a more challenging training process, we selected the classes '7' and '8', as they share visual similarities and are harder to distinguish. Choosing more distinct classes, such as '1' and '5', resulted in a lower loss and higher accuracy under the same hyperparameters, but also lead to overfitting. We will refer to the model trained with classes '1' and '5' as 15model.
In Figure 3 below, we present a plot showing the output of 15model, which is more easily distinguishable compared to '7' and '8'. This highlights why the latter pair of classes presents a greater challenge for the model.



```
Epoch 0  Loss: [1.96830875] , Training Accuracy:  43.5 %
Epoch 1  Loss: [1.61776849] , Training Accuracy:  43.75 %
Epoch 2  Loss: [1.20352485] , Training Accuracy:  46.5 %
Epoch 3  Loss: [0.93609943] , Training Accuracy:  55.375 %
Epoch 4  Loss: [0.77897552] , Training Accuracy:  63.249999
Epoch 5  Loss: [0.65987416] , Training Accuracy:  69.875 %
Epoch 6  Loss: [0.56575924] , Training Accuracy:  73.875 %
Epoch 7  Loss: [0.48977848] , Training Accuracy:  77.625 %
Epoch 8  Loss: [0.42775572] , Training Accuracy:  81.75 %
Epoch 9  Loss: [0.37693317] , Training Accuracy:  84.0 %
Epoch 10  Loss: [0.33522792] , Training Accuracy:  85.875 %
Epoch 11  Loss: [0.30092657] , Training Accuracy:  87.875 %
Epoch 12  Loss: [0.27257196] , Training Accuracy:  89.875 %
Epoch 13  Loss: [0.24893649] , Training Accuracy:  92.125 %
Epoch 14  Loss: [0.22902177] , Training Accuracy:  93.0 %
Epoch 15  Loss: [0.21204517] , Training Accuracy:  93.75 %
Epoch 16  Loss: [0.19740773] , Training Accuracy:  94.25 %
Epoch 17  Loss: [0.18465522] , Training Accuracy:  95.0 %
Epoch 18  Loss: [0.17344256] , Training Accuracy:  95.375 %
Epoch 19  Loss: [0.16350541] , Training Accuracy:  95.75 %
Epoch 20  Loss: [0.1546389] , Training Accuracy:  96.25 %
```

(a) First 20 epochs of 15model



(b) Confusion matrix of 15model

Figure 3: Comparison of loss and confusion matrix

With a learning rate of 0.004, we can see that about 18 to 20 epochs could be sufficient for 15model, as it reached an accuracy of 95% to 96% at that point. After 100 epochs, 15model reached an accuracy of 99.875%. On the test set, the accuracy reached 100%, which might indicate a sign of overfitting.

5 Code

The complete code is available in the Colab notebook at the following link: Colab Notebook