Two-layer neural network for classification on MNIST dataset

Yuxin Li
SDS, Fudan University, Shanghai, China

Abstract

Neural networks have powerful effects on dataset classification. In this article, we applied a two-layer neural network and trained a network with the best hyperparameters on the MNIST dataset.

Keywords: neural network; classification; MNIST dataset

1 Introduction

Dense neural networks (DNNs) can be the basic part of deep learning and they may perform well in some simple tasks such as classification.

MNIST, one of the most common toy datasets in computer vision, is a great dataset for the handwritten digit classification problem. It is a very authenticated and great dataset for students and researchers. It has 60000 images with 10 classes (0-9) which is enormous in itself. Each image in the MNIST dataset is 28 in height and 28 in weight which makes the image 784-dimensional vectors Deng (2012). Each image in MNIST is a gray-scale image and the range is 0-255 which indicates the brightness and the darkness of that particular pixel.

Many methods like Artificial Neural Networks and Convolutional Neural Networks have been applied in this dataset classification and achieved great accuracy on both training and test dataset. Kriegeskorte and Golan (2019) Although these deep-learning methods have full accuracy on the dataset and can further improve to tackle more complicated datasets, in our experiment, a simple two-neural network trained within seconds is also able to reach an accuracy of 99.998% on training dataset without augmentation and 98.37% on the testing ones.

2 Method

2.1 Architecture

Our two-layer model is composed of two linear matrices called weights and two biases. For each layer, the output is connected to an activation function, ReLU or Sigmoid specifically. The detailed implementation in Python can be accessed in the Github repository.

2.2 Algorithm

We adopt the classic backpropagation strategy. If one treats each layer, linear, activation functions, or loss function, as a parametric multivariate function $f^{(k)}(x;\theta_k)$ where θ_k is the parameter, then the entire model loss can be regarded as a composition

$$\mathcal{L}(x_1) = f^{(m)}(f^{(m-1)}(\cdots f^{(1)}(x_1;\theta_1)\cdots;\theta_{m-1});\theta_m).$$

Our target is to optimize the error, which means $\min_\theta \mathcal{L}(x;\theta)$. The gradient method suggests that we can proceed by Newton's iteration, Horn and Johnson (2012)

$$\theta_k \leftarrow \theta_k - \eta\frac{\partial \mathcal{L}}{\partial \theta_k}.$$

The derivative $\frac{\partial \mathcal{L}}{\partial \theta_k}$ can be computed efficiently by backpropagation algorithm. We denote $x_{k+1} = f^{(k)}(x_k;\theta_k)$, meaning that x_{k+1} is a function of x_k and θ_k, the algorithm innovatively proceses that

$$\frac{\partial \mathcal{L}}{\partial \theta_k} = \frac{\partial \mathcal{L}}{\partial x_{k+1}}\cdot\frac{\partial x_{k+1}}{\partial \theta_k} \quad \frac{\partial \mathcal{L}}{\partial x_k} = \frac{\partial \mathcal{L}}{\partial x_{k+1}}\cdot\frac{\partial x_{k+1}}{\partial x_k}. \tag{1}$$

Applying $k = m, m-1, ..., 1$ in order gives all the $\frac{\partial \mathcal{L}}{\partial x_k}$ and $\frac{\partial \mathcal{L}}{\partial \theta_k}$. Additionally, l_2-regularization can be considered and $\|\theta_k\|^2$ contributes extra gradient to the final loss, which should be added to the gradient $\frac{\partial \mathcal{L}}{\partial \theta_k}$. The complete algorithm, generally known as SGD, is summarized as follows.

Algorithm 1: SGD optimization

Input: data x_1, current parameters θ_k, model layers (loss function included) $f^{(k)}$, learning rate η, regularization term λ

for $k \leftarrow 1$ **to** m **do**
  Record $x_{k+1} = f^{(k)}(x_k, \theta_k)$;
end
for $k \leftarrow m$ **to** 1 **do**
  Compute $\frac{\partial \mathcal{L}}{\partial \theta_k} = \frac{\partial \mathcal{L}}{\partial x_{k+1}} \cdot \frac{\partial x_{k+1}}{\partial \theta_k}$;
  Compute $\frac{\partial \mathcal{L}}{\partial x_k} = \frac{\partial \mathcal{L}}{\partial x_{k+1}} \cdot \frac{\partial x_{k+1}}{\partial x_k}$;
  Add regularization gradient $\frac{\partial \mathcal{L}}{\partial \theta_k} \leftarrow \frac{\partial \mathcal{L}}{\partial x \theta_k} + \lambda \frac{\partial}{\partial \theta} \|\theta\|^2$;
  Update $\theta_k \leftarrow \theta_k - \eta \frac{\partial \mathcal{L}}{\partial \theta_k}$;
end

A minibatch is sampled from the whole dataset and the SGD function is called for the overall training process on the MNIST dataset. The dataset also should be shuffled randomly for stochastic updates. After 1 epoch, we validate the model on validation data and check the accuracy. If the accuracy drops back compared to the former, the learning-rate decay strategy could be introduced for more precise modifications. The complete pseudocode is illustrated in the following.

Algorithm 2: Training process

Input: Data x and labels y. Validation data $\hat{x}$ and corresponding $\hat{y}$. Initial learning rate η and decay rate β. Training epochs n, etc.

for $i \leftarrow 1$ **to** n **do**
  Shuffle x, y simultaneously;
  for *All batches in order* **do**
    Sample a batch of x, y with a given batch size as for the inputs and the loss criterion ;
    Apply the update function SGD on batch and the model ;
  end
  Compute the accuracy on the validation data by predicting $\hat{y}$ from $F(\hat{x})$;
  if *Validation accuracy drops* **then**
    Learning-rate decays by $\eta \leftarrow \beta\eta$;
  end
end

3 Training detalis

3.1 Loss function

We consider two classic loss functions, the mean square error, and the binary cross entropy loss. According to our dataset, suppose there are t categories for a classification task and for each sample, $y_j (j = 1, 2, ..., t)$ is boolean, indicating whether it is in the category or not Meshkini, Platos, and Ghassemain (2020). Suppose p_{y_j} denotes the probability that our model predicts the sample belongs to y_j, then intuitively p_{y_j} should approximate y_j as the samples grow. This leads to the cross-entropy(CE) loss and mean squared error(MSE) loss functions which give a measure of the above approximation:

$$\mathcal{L}_{MSE} = \frac{1}{2N} \sum_{j=0}^{t} (y_j - p_{y_j})^2 \tag{2}$$

$$\mathcal{L}_{CE} = -\frac{1}{N} \sum_{j=0}^{t} y_j \log p_{y_j} \tag{3}$$

The results below show that cross-entropy loss performs better than MSE loss due to its more stable training loss tendency, especially in this two-layer neural network classifier. Thus, CE loss is preferred for the following experiments.

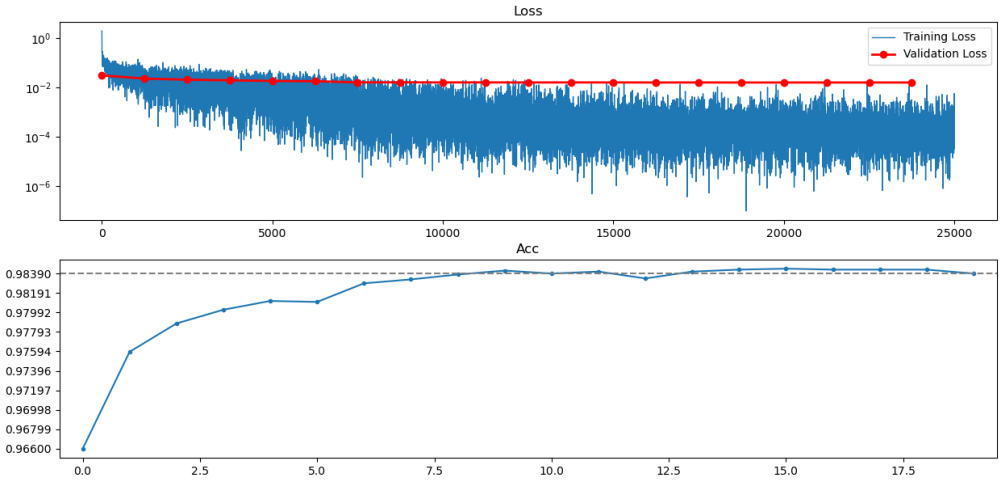

Figure 1: Network performance with MSE loss

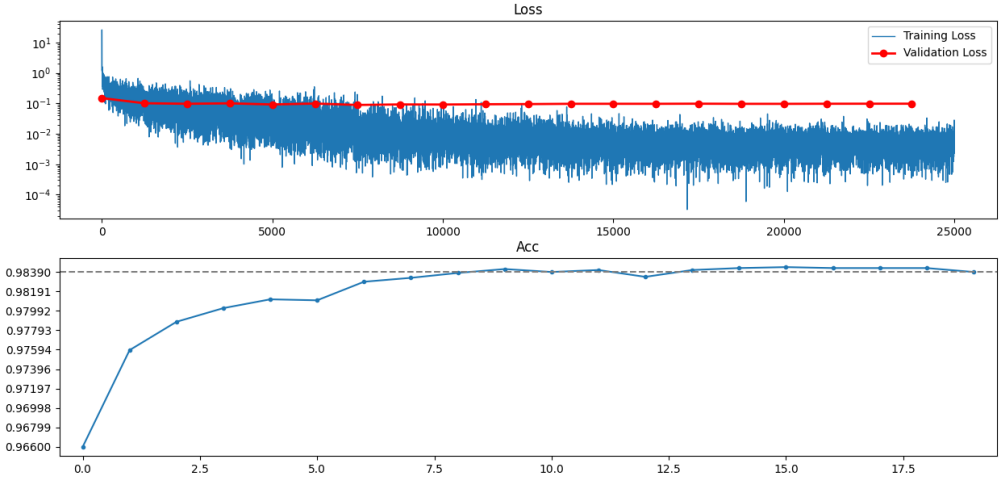

Figure 2: Network performance with CE loss

3.2 Activation function

Our neural network is only composed of two dense layers and two activation functions attached afterward. Due to the classification task, the output should belong to [0,1] and the sigmoid is chosen for the second activation function. As for the first activation function, we select between ReLU and sigmoid.

With a fixed learning rate of 0.03, hidden size 1000 and 20 epochs to train the result performed with ReLU is shown in Fig 1, whilst the one with sigmoid in Fig 2. ReLU generally outperforms the sigmoid with considerably higher accuracy on the validation set. (We compare the result fixed with CE loss here).

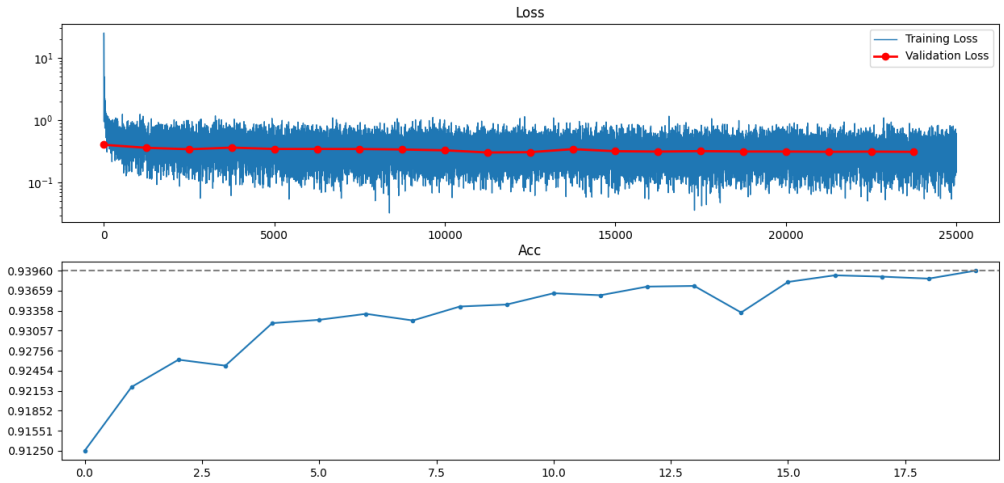

Figure 3: Network performance with Sigmoid+Sigmoid activation functions

3.3 Hidden size

In the neural network framework, identifying the number of neurons in the hidden layer influences the effect of the neural network considerably. Using too few neurons in the hidden layer will lead to underfitting. Conversely, using too many neurons can also cause problems of overfitting and a long time for training.

We choose the parameter for hidden size with a fixed learning rate of 0.03 and 20 epochs. The result is illustrated in the table below, suggesting that 1000 neurons in the hidden layer could be preferable.

Table 1: Hidden size selection

Size	50	100	200	500	600	800	1000
ACC	0.9743	0.9779	0.9811	0.9822	0.9825	0.9831	**0.9843**

3.4 Learning rate

The learning rate is an important hyperparameter in neural network training. Choosing the optimal learning rate is crucial because it determines whether a neural network can converge to the global minimum. If a higher learning rate is chosen, it may result in undesirable consequences on the loss function, and thus it may never reach the global minimum because it is likely to skip over it. Choosing a lower learning rate can help the neural network converge to the global minimum, but it will take a lot of time and is also more likely to trap the neural network in a local minimum.

As illustrated in Algorithm 2, we validate after each epoch and check whether the accuracy falls back. If so, the learning rate is halved and the training process continues. Additionally, the initial learning rate might impact the training. The learning rate varies from 3 to 3×10^{-5} and the model is trained for 20 epochs with identical network architecture. We notice from Table 2 that the large learning rates which are greater than 0.03 cause model collapse. When the learning rate is less than 0.03, underfitting occurs in the model within 20 epochs and the accuracy decreases.

Table 2: Learning rate selection

Size	3	0.3	0.03	3×10^{-3}	3×10^{-4}	3×10^{-5}
ACC	0.103	0.103	**0.9825**	0.975	0.9426	0.8869

3.5 Regularization

In order to avoid overfitting in the neural network model, a variety of regularizations on weights(not including the bias) have been chosen to validate the model. As shown in Table 3, a larger regularization penalty negatively impacts the performance and we may choose the coefficient 10^{-6} for the best performance.

Table 3: Learning rate selection

Size	0.01	1×10^{-3}	1×10^{-4}	1×10^{-5}	1×10^{-6}	1×10^{-7}	0
ACC	0.103	0.1038	0.9693	0.982	**0.9826**	0.9824	0.9825

4 Result and visualization

We summarized all the results with different hyperparameters and settings in the neural network and obtained one of the best models. It has a 1000 hidden size with random seed 20230318, utilizes ReLU and Sigmoid as the first and second activation function respectively, and adopts CE as the loss function for 20 epochs training without regularization terms. It finally reached 99.998% accuracy on the training test while the accuracy on the validation and test set are 98.39% and 98.37% respectively. The result is shown in Figure 2 and we notice that the validation loss hardly improves in spite of a noticeable decreasing trend in training loss.

The first 12 samples which are failed to classify in the test dataset are displayed in the following. It seems that in some cases our model does provide more reasonable predictions than their actual labels. However, some of which, for example, the error in the third and the ninth figures, are far from the ground truth, indicating that our model still needs improvements.

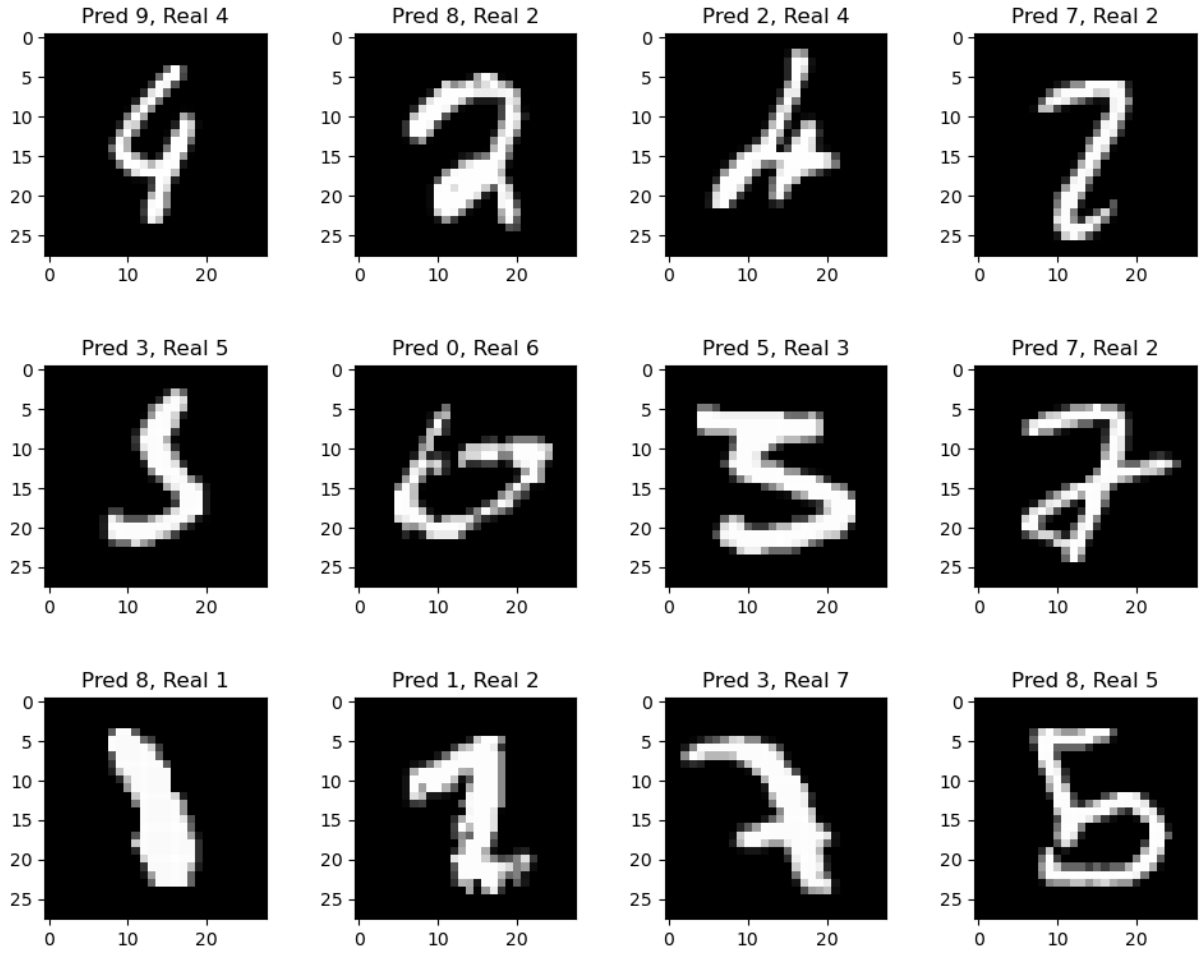

Figure 4: Failed cases in test dataset

Furthermore, the weights of the first layer are visualized by embedding the 784×1000 matrix to 784×3 using PCA. Then this embedded matrix is reshaped into $28 \times 28 \times 3$, with entries clipped to [-1,1] and mapped to [0,1] by an affine transformation.

As plotted, in the beginning, the weights are initialized randomly and the corresponding PCA result is merely a random distribution of pixels. After one epoch's training, the center forms a prototype, which corresponds to 96.6% accuracy on the validation set. The final one depicts the weights after 20 epochs of training and it has

remarkable features. It indicates that the neural network is somehow learning a certain pattern from the data distribution.

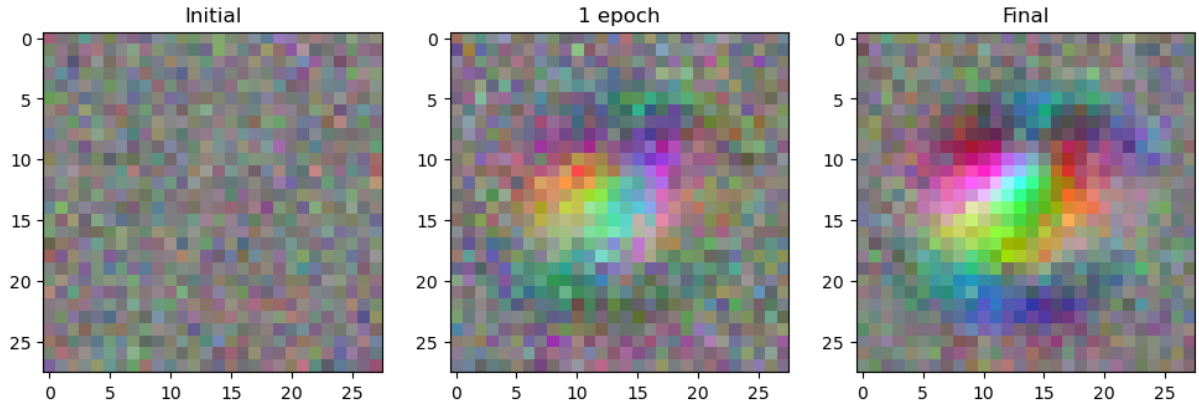

Figure 5: Weights in the first layer of NN visualization via PCA

5 Conclusion

We have already seen that a simple two-layer network is able to reach a 98.37% accuracy on the MNIST dataset. Actually, within two layers, there are still many tricks like data augmentation which can improve the final result, and there are other techniques such as fine-tuning beyond what we have ever tried. We may explore the simple model more and apply it to other tiny datasets for the general conclusion.

References

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, *29*(6), 141–142.

Horn, R. A., & Johnson, C. R. (2012). *Matrix analysis*. Cambridge university press.

Kriegeskorte, N., & Golan, T. (2019). Neural network models and deep learning. *Current Biology*, *29*(7), R231–R236.

Meshkini, K., Platos, J., & Ghassemain, H. (2020). An analysis of convolutional neural network for fashion images classification (fashion-mnist). In *Proceedings of the fourth international scientific conference "intelligent information technologies for industry"(iiti'19) 4* (pp. 85–95).

Supplementary Materials

The implementation in Python can be accessed in the GitHub repository via the link https://github.com/Grit1021/Two-layer-neural-network-for-classification-on-MNIST-dataset.

An already-trained model with 98.37% accuracy on the test dataset can be downloaded in the cloud drive via the link https://pan.baidu.com/s/1wIt9RMwZCdqEtjDOjDOigA?pwd=cvcv by extracting password cvcv. See more details on GitHub for instructions.