TWO-LAYER NEURAL NETWORK CLASSIFIER ON MNIST DATASET

Zehao Zhang

Abstract. In this article we implement a two-layer neural network and seek for the best hyperparameters on MNIST dataset.

1. Introduction

Dense neural network has long been a first course for deep learning beginners. And the MNIST handwriting dataset [3] is probably one of the most popular dataset to work with. It contains 50000 figures for training, 10000 for validation and another 10000 for testing. Depsite more advanced tools like convolution layers and structures has achieved full accuracy on the dataset and can further progress to more comlicated tasks, we shall here be astonished with the fact that, a simple two-neural network trained within seconds is able to reach an accuracy of 99.994% on training dataset without augmentation and 98.46% on the testing ones.

2. Architecture

Our two-layer model is composed by two linear matrices called weights and two biases. For each layer the output is connected to a activation function, ReLU or sigmoid, to be specific. The implementation in Python can be accessed in GitHub repository. An already-trained model with 98.46% accuracy on testing dataset can be downloaded at the cloud drive with extracting password **owor**. See more details on GitHub for instructions.

3. Algorithm

We adopt the classic back propagation strategy. If one treats each layer, linear, activation functions or loss functions, 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

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

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

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

And what remains is to compute the derivative $\frac{\partial\mathcal{L}}{\partial\theta_k}$ efficiently. If 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 back propagation algorithm innovatively proposes that

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

Applying $k = m, m-1, \ldots, 1$ in order will solve all the $\frac{\partial \mathcal{L}}{\partial x_k}$ and $\frac{\partial \mathcal{L}}{\partial \theta_k}$. Additionally, if l2-regularizations are considered, $\|\theta_k\|^2$ contributes extra gradient to the final loss, which should be added to the gradient $\frac{\partial \mathcal{L}}{\partial \theta_k}$. The algorithm outline, generally known as SGD, can therefore be summarized as follows.

Algorithm 1: SGD

input: Input data x_1. Current parameters θ_k. Model layers (loss function included) $f^{(k)}$. Learning rate η. Regularization term λ.

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

The overall training process on MNIST dataset is by sampling a minibatch from the whole dataset and call the SGD function. One should first randomly shuffle the dataset for stochastic updates.

After 1 epoch or so, we validate the model on the validation data and check the accuracy. If the accuracy drops back compared to the former, we could introduce the learning-rate decay strategy for more precise modifications. The pseudocode is illustrated in the next chunk.

Algorithm 2: Training

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

4. Hyperparameters

4.1. **Loss Function.** There are two loss functions we have taken into account, the mean square error and the binary cross entropy loss. Suppose there are t categories for a classification task and for each sample, y_j $(j = 1, 2, \ldots, t)$ is boolean, indicating whether it is in the category or not. Suppose y'_j stands for the probability that our model predicts that the sample belongs to y_j, then intuitively y'_j should approximate y_j. This leads to the two loss functions that measure the approximation,

$$\begin{aligned} \mathcal{L}_{MSE} &= \frac{1}{2N}\sum_{j=0}^{t}(y_j - y'_j)^2 \\ \mathcal{L}_{BCE} &= -\frac{1}{N}\sum_{j=0}^{t}\left(y_j \log y'_j + (1-y_j)\log(1-y'_j)\right). \end{aligned} \tag{2}$$

As far as we are concerned, BCE does not perform well with merely 2 linear layers. It faces numerical instability oftentimes and the accuracy notably lags behind MSE. A typical performance is plotted in figure 1.

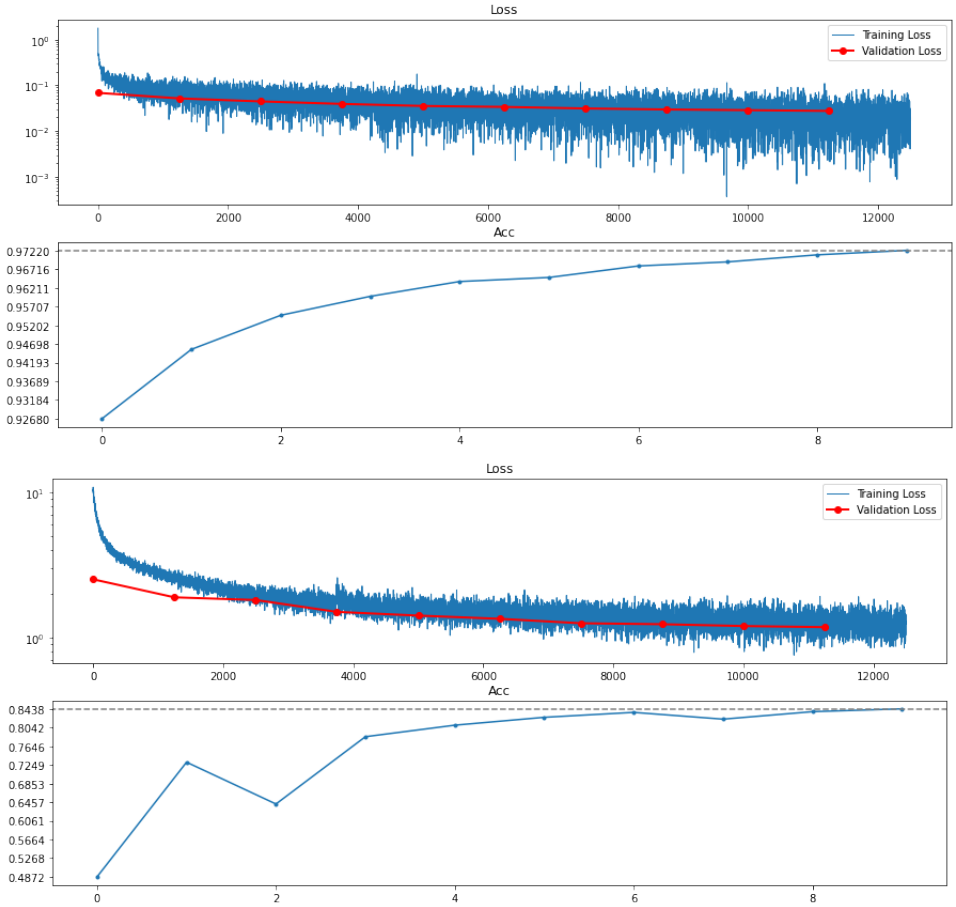

Figure 1. MSE (first) and BCE (second)

4.2. **Activation Function.** As the neural network we implement is only composed by two dense layers, there are only two activation functions attached afterwards. Further, the second activation function is fixed by sigmoid, since it should map the output to $[0, 1]$ and there are hardly any common choices besides sigmoid and softmax. As for the first activation, we have tried ReLU and sigmoid.

Setting learning rate 3×10^{-3}, hidden size 500 and 10 epochs to train, the result performed with ReLU is shown in figure 1, whilst the one with sigmoid shown in figure 2. We conclude that ReLU generally outperforms the sigmoid with notably higher accuracy on validation set.

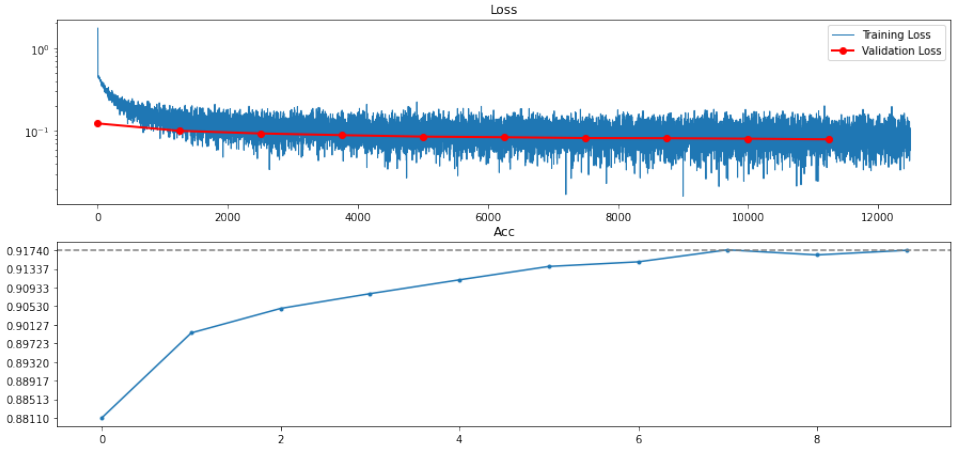

FIGURE 2. Sigmoid

4.3. **Hidden Size.** We have picked up sizes such as $50, 100, 200, 500, 600, 700, 800$ and 1000 with fixed hyperparameters learning rate 3×10^{-3} and 10 epochs. The result is illustrated in the table below, suggesting that 800 neurons in the hidden layer might be most preferable.

Size	50	100	200	500	600	700	800	1000
Acc	95.95%	96.58%	96.84%	97.23%	97.35%	97.25%	97.31%	97.27%

TABLE 1. Hidden Size

4.4. **Learning Rate.** As illustrated in algorithm 2, we validate after each epoch and check whether the accuracy falls back. If so, half the learning rate and continue training. Still, the initial learning rate might impact the training. We have conducted our search with learning rate varing from 3 to 3×10^{-5} and training for 10 epochs with identical network architecture. From table 2 we learn that large learning rates, 0.3 for example, cause collapse in the models. When the learning rate is below 3×10^{-2}, the model faces underfitting within 10 epochs and has decreasing accuracy.

Size	3	3e-1	3e-2	3e-3	3e-4	3e-5
Acc	9.91%	9.16%	98.14%	97.26%	93.60%	86.62%

TABLE 2. Learning Rate

4.5. **Regularization.** We have tried a variety of regularizations on weights (not including the bias). As shown in table 3, large regularization penalty negatively impacts the performance. Regularization with coefficient 10^{-5} is persumably a nice choice.

Size	1e-2	1e-3	1e-4	1e-5	1e-6	1e-7	0
Acc	9.91%	76.16%	98.01%	98.35%	98.15%	98.20%	98.24%

TABLE 3. Regularization

5. VISUALIZATION

One of the best models we have trained has hidden size 600 with random seed 2023, and 20 epochs plus zero regularizations. It utilizes the ReLU as the first activation function and MSE as the loss function. It has reached 99.994% accuracy on the training set, where there are 3 images out of 50000 that it fails to classify. The loss and the accuracy on validation set is 98.25% and on the testing set 98.46%. However, the plotted validation loss hardly improves in spite of a noticeable downcreasing trend in training loss.

The three samples that our model fails to classify in the training data is displayed in figure 4. It seems, intuitively, that our model does provide more reasonable predictions on these ones than their labels.

The first 12 failed cases in the testing dataset are showcased in figure 5. Some of which, for example the error in the second and the ninth figures, are far from tolerable, implying that there are still shortages for our model.

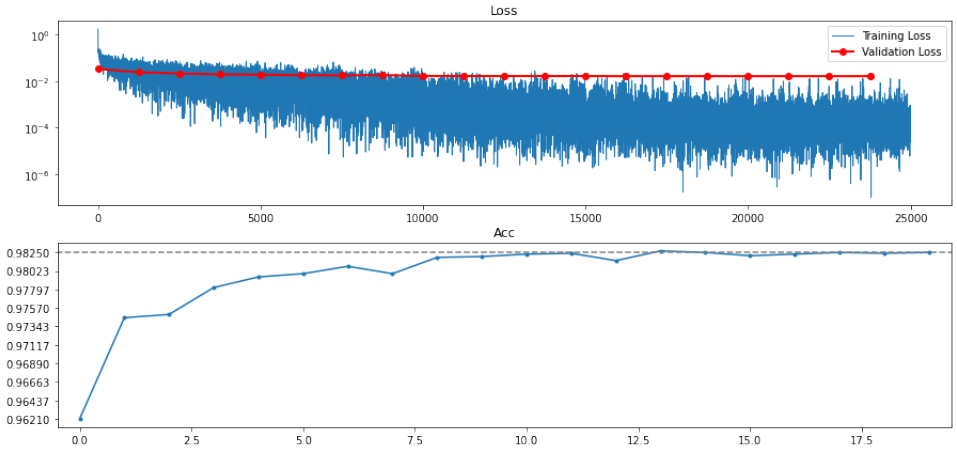

FIGURE 3. Best Model

We have also trained a model that achieves 99.998% accuracy on the training dataset where there is only 1 exception. But it is overfitting with sacrificed accuracy on testing set, around 98.20%.

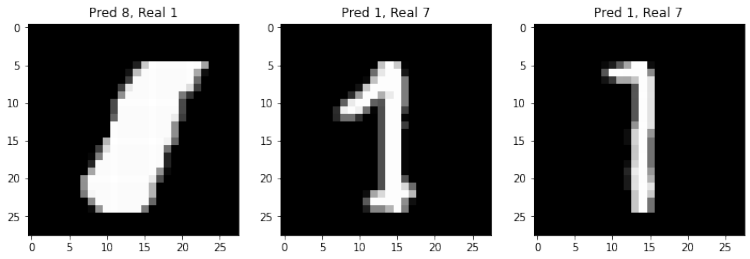

FIGURE 4. Failure in Training

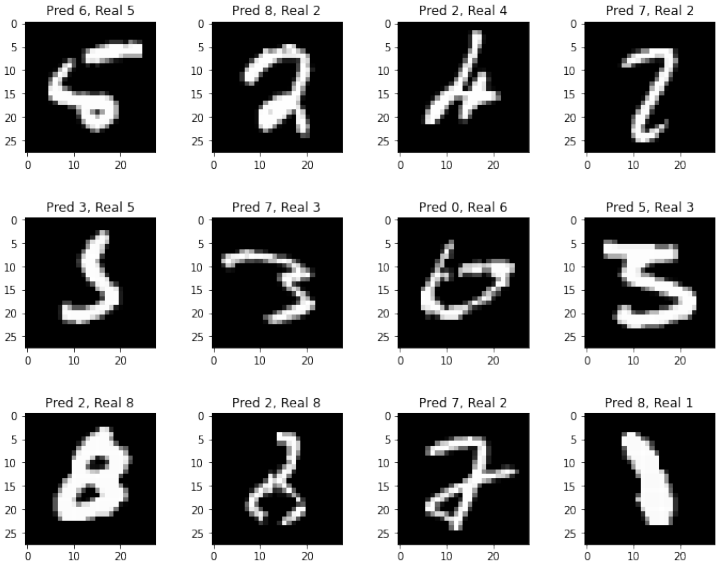

FIGURE 5. Failure in Testing

Further, figure 6 visualizes the weights of the first layers by embedding the 784×500 matrix to 784×3 using PCA. Then the 784×3 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, in the center forms a prototype. It corresponds to 96.21% accuracy on the validation set. The final one depicts the weights after training for 20 epochs and it has notable features. It indicates that the neural network is somehow learning a certain pattern from the data distribution.

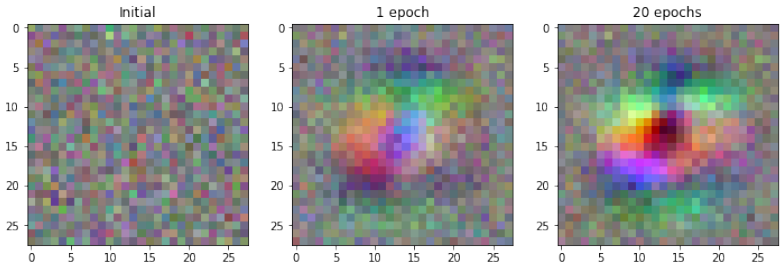

FIGURE 6. Weights

6. CONCLUSIONS

We have already seen that a simple two-layer network is able to reach a 98.46% accuracy on MNIST dataset, possibly outperforms any traditional method one can imagine. Yet this is not its limit. Within two layers, there are still tricks like data augmentation we have not yet applied, and there are other techniques as fine-tuning beyond we have ever tried.

REFERENCES

[1] F. Chollet. *Deep Learning with Python*, Manning Publications, 2017.
[2] F. Pedregosa and Varoquaux et al. Scikit-learn: Machine Learning in Python, *JMLR* 12, pp. 2825-2830, 2011.
[3] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6), pp. 141-142, 2012.
[4] M. A. Nielsen. *Neural Networks and Deep Learning*, Determination Press, 2015.
[5] R. A. Horn and C. R. Johnson. *Matrix Analysis*, 2nd ed., Cambridge University Press, Cambridge, UK, 2013.

School of Data Science, Fudan University
Shanghai, People's Republic of China
E-mail address: 20307130201@fudan.edu.cn