Three-layer Neural Network Classifier on Fashion-Mnist Dataset

Zihao Cheng
School of Data Science,
Fudan University,
21307130080@m.fudan.edu.cn

Abstract

In this article, we implement a three-layer neural network and seek for the best hyperparameters on the Fashion-Mnist dataset.

1 Introduction

Dense neural network has long been a first course for deep learning beginners. And the Fashion-Mnist dataset is one of the most popular datasets used. It contains 60,000 figures for training, and 10,000 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 three-neural network trained within minutes is able to reach an accuracy of 91.74% on training dataset without augmentation and 88.64% on the testing ones.

2 Architecture

Our three-layer model is composed by three linear matrices called weights and three biases. For each layer the output is connected to a activation function, ReLU, sigmoid, or linear, to be specific. The implementation in Python can be accessed in GitHub repository. An already-trained model with 88.64% accuracy on testing dataset can be downloaded at the cloud drive with extracting password 'd7rd'. See more details on GitHub for instructions.
cloud drive: https://pan.baidu.com/s/1vM4CFfH0R3SmKgnA4nAvyA
GitHub: https://github.com/football-prince/DATA130051-PJ1

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

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

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

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

And what remains is to compute the derivative $\frac{\partial \mathscr{L}}{\partial \theta_k}$ 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

$$\frac{\partial \mathscr{L}}{\partial \theta_k} = \frac{\partial \mathscr{L}}{\partial x_{k+1}}\frac{\partial x_{k+1}}{\partial \theta_k}, \quad \frac{\partial \mathscr{L}}{\partial x_k} = \frac{\partial \mathscr{L}}{\partial x_{k+1}}\frac{\partial x_{k+1}}{\partial x_k}.$$

Applying $k = m, m-1, \dots, 1$ in order will solve all the $\frac{\partial \mathscr{L}}{\partial x_k}$ and $\frac{\partial \mathscr{L}}{\partial \theta_k}$. Additionally, if L2-regularizations are considered, $\|\theta_k\|^2$ contributes extra gradient to the final loss, which should be added to the gradient $\frac{\partial \mathscr{L}}{\partial \theta_k}$. The algorithm outline, generally known as SGD, can therefore be summarized as follows.

Algorithm 1 SGD

```
1: input: Input data $x_1$. Current parameters $\theta_k$. Model layers (loss function included) $f^{(k)}$.
   Learning rate $\eta$. Regularization term $\lambda$.
2: for $k = 1$ to $m$ do
3:     Record $x_{k+1} = f^{(k)}(x_k;\theta_k)$.
4: end for
5: for $k = m$ to 1 do
6:     Compute $\frac{\partial \mathscr{L}}{\partial \theta_k} = \frac{\partial \mathscr{L}}{\partial x_{k+1}} \cdot \frac{\partial x_{k+1}}{\partial \theta_k}$.
7:     Compute $\frac{\partial \mathscr{L}}{\partial x_k} = \frac{\partial \mathscr{L}}{\partial x_{k+1}} \cdot \frac{\partial x_{k+1}}{\partial x_k}$.
8:     Add regularization gradient $\frac{\partial \mathscr{L}}{\partial \theta_k} \leftarrow \frac{\partial \mathscr{L}}{\partial \theta_k} + \lambda \frac{\partial}{\partial \theta_k}\|\theta_k\|^2$.
9:     Update $\theta_k \leftarrow \theta_k - \eta \frac{\partial \mathscr{L}}{\partial \theta_k}$.
10: end for
```

The overall training process on Fashion-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

```
1: input: Data $x$ and labels $y$. Validation data $\hat{x}$ and corresponding $\hat{y}$.
2: Initial learning rate $\eta$ and decay rate $\beta$. Training epochs $n$, etc.
3: for $i = 1$ to $n$ do
4:     Shuffle $x, y$ simultaneously.
5:     for All batches in order do
6:         Sample a batch of $x, y$ with given batch size as for the inputs and the loss criterion.
7:         Apply the update function SGD on the batch and our model.
8:     end for
9:     Compute the accuracy on validation data by predicting $\hat{y}$ from $F(\hat{x})$.
10:    if Validation accuracy drops then
11:        Learning-rate decay by $\eta \leftarrow \beta\eta$.
12:    end if
13: end for
```

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,

$$L_{MSE} = \frac{1}{2N} \sum_{j=0}^{t} (y_j - y'_j)^2 \tag{1}$$

$$L_{BCE} = -\frac{1}{N} \sum_{j=0}^{t} \left(y_j \log y'_j + (1 - y_j) \log(1 - y'_j) \right). \tag{2}$$

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

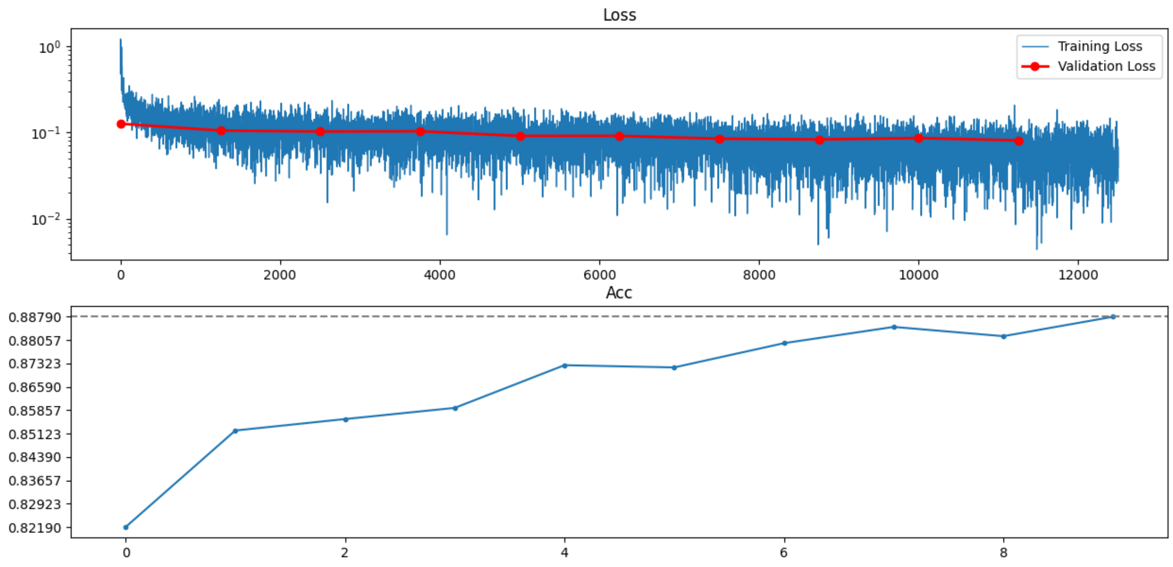

Figure 1: MSE

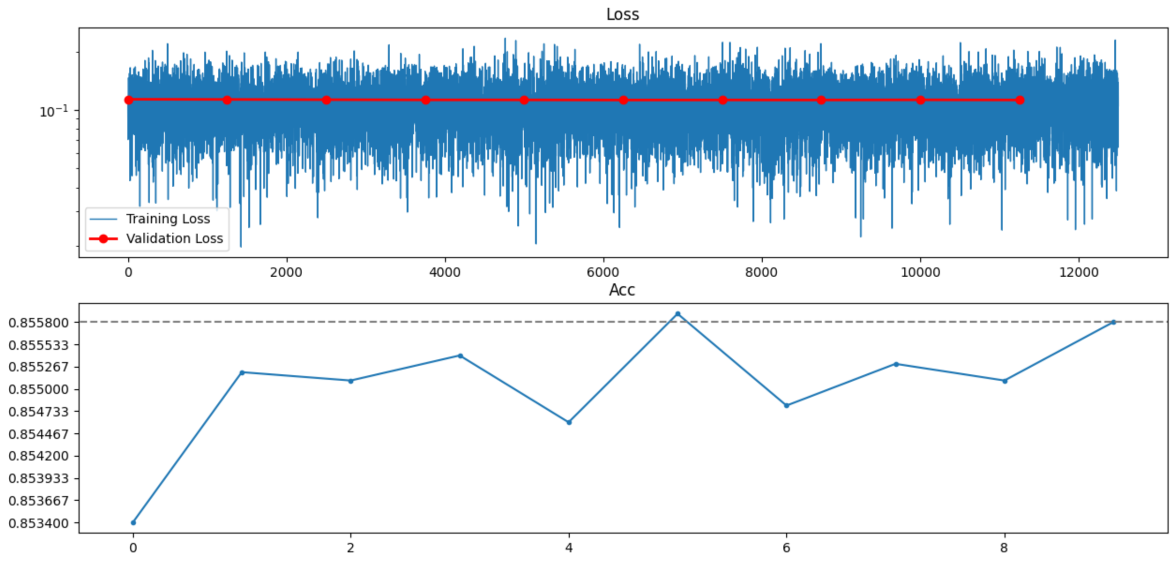

Figure 2: BCE

4.2 Activation Function

As the neural network we implement is only composed by three dense layers, there are three 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 two activation, we have tried ReLU and sigmoid.

Setting learning rate 3×10^{-3}, hidden size $[784, 256]$,$[256, 128]$,$[128, 10]$ and 10 epochs to train, the result performed with ReLU is shown in figure 1, whilst the one with sigmoid shown in figure 3. We conclude that ReLU generally outperforms the sigmoid with notably higher accuracy on validation set.

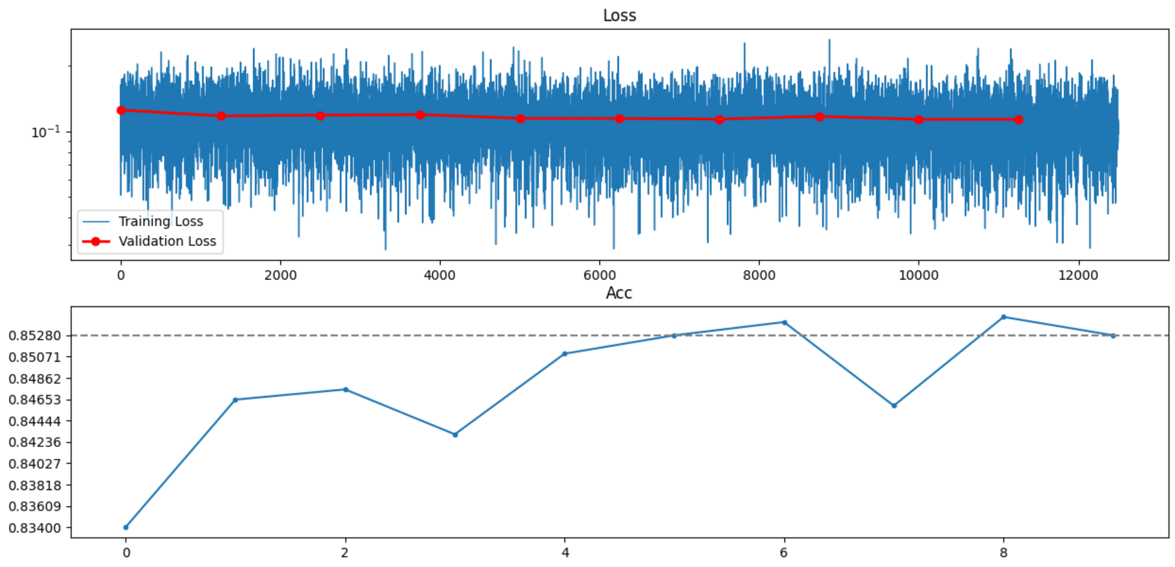

Figure 3: Sigmoid

Because the three-layer neural network is very complex and the performance of the two-layer neural network is similar to that of the three-layer neural network, we conducted experiments using the two-layer neural network in the next few parts.

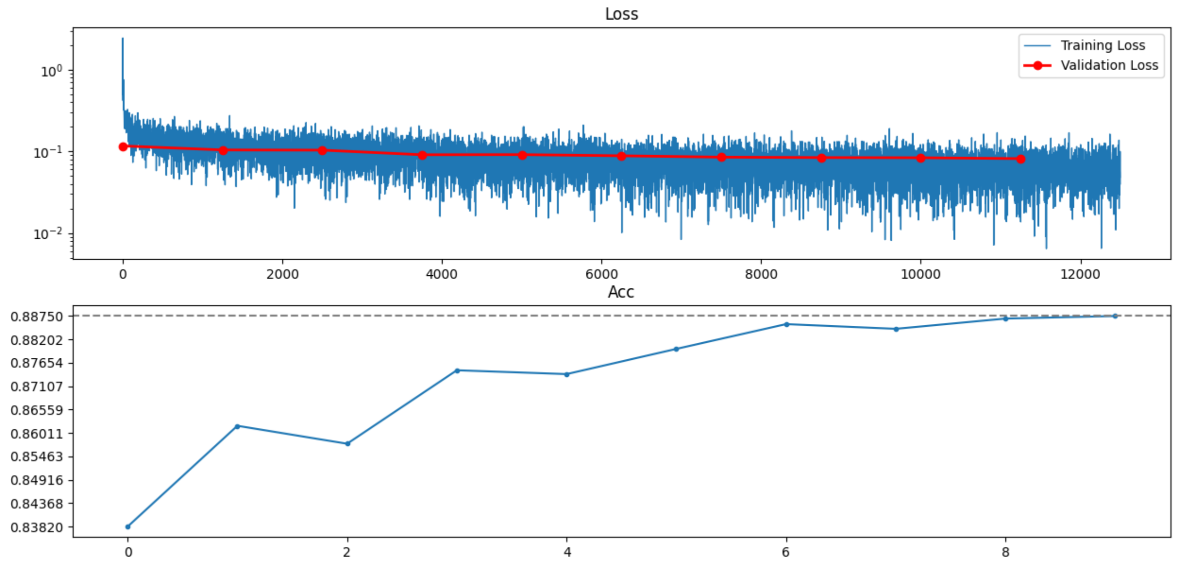

Figure 4: two layers neural network

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 1000 neurons in the hidden layer might be most preferable.

Table 1: Hidden Size

Size	50	100	200	500	600	700	800	1000
Acc	85.66%	86.02%	87.00%	88.02%	87.61%	88.12%	88.34%	88.59%

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 varying from 0.3 to 3×10^{-5} and training for 10 epochs with identical network architecture. From table 2 we learn that large learning rates, for example, 0.3, 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.

Table 2: Learning Rate

Size	3	3e-1	3e-2	3e-3	3e-4	3e-5
Acc	9.96%	9.96%	90.00%	87.61%	83.88%	74.35%

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^{-7} is presumably a nice choice.

Table 3: Regularization

Size	1e-2	1e-3	1e-4	1e-5	1e-6	1e-7	0
Acc	61.74%	81.83%	88.83%	89.79%	89.86%	90.06%	90.00%

5 Validation

Training Accuracy (91.54%): The model shows a good level of proficiency on the training data, indicating effective learning but suggesting there might be room for improvement, possibly by further tuning or expanding the training dataset to cover more varied scenarios.
Validation Accuracy (88.79%): The drop in accuracy from training to validation suggests the model may be overfitting to the training data. This highlights a need for techniques to improve generalization, such as implementing dropout, regularization, or adjusting the model's architecture.
Testing Accuracy (88.33%): The accuracy on the testing set is slightly lower than on the validation set, but close, which is encouraging as it shows the model's ability to generalize to new, unseen data remains consistent.

6 Visualization

The samples that our model fails to classify in the training data is displayed in figure 5. 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 6.

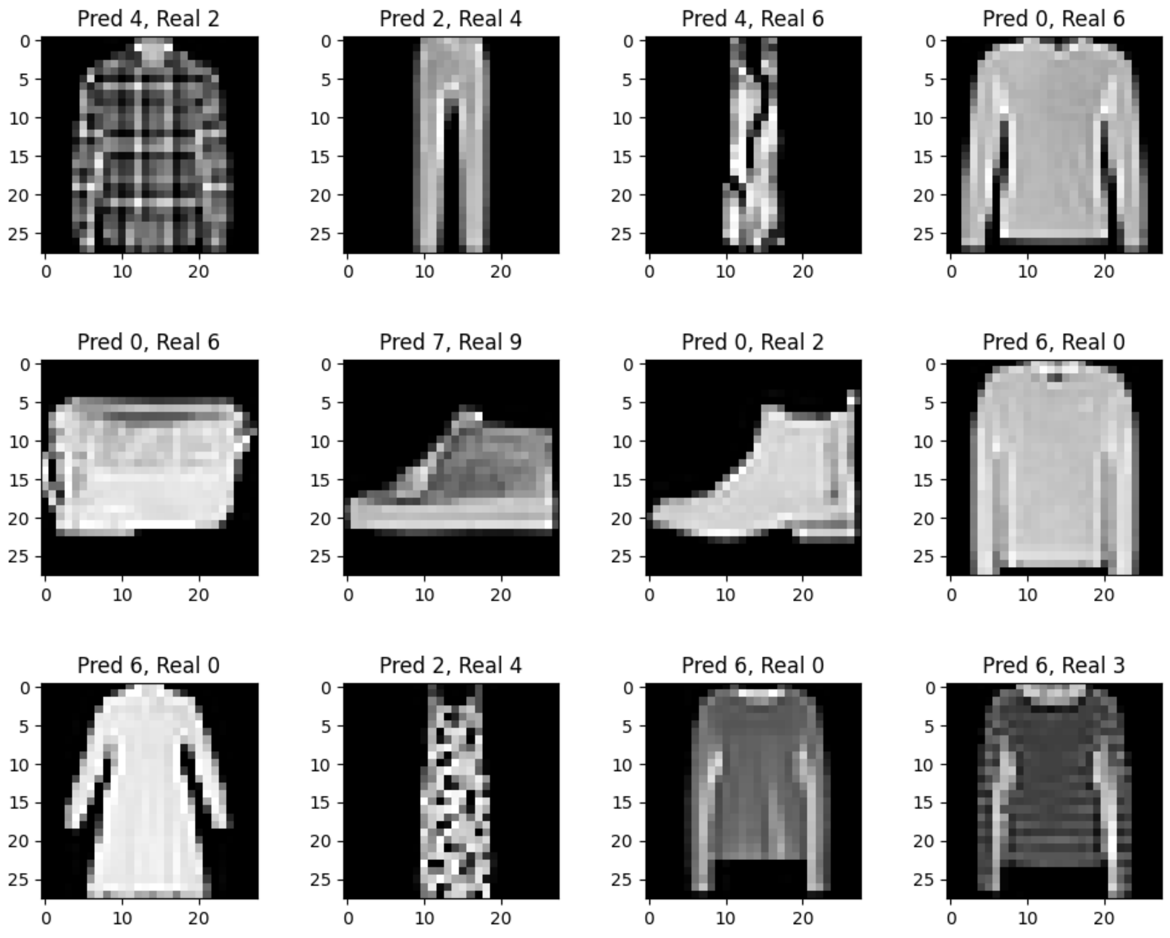

Figure 5: train failures

Some of which, for example the error in the fourth and the eleventh figures, are far from tolerable, implying that there are still shortages for our model.

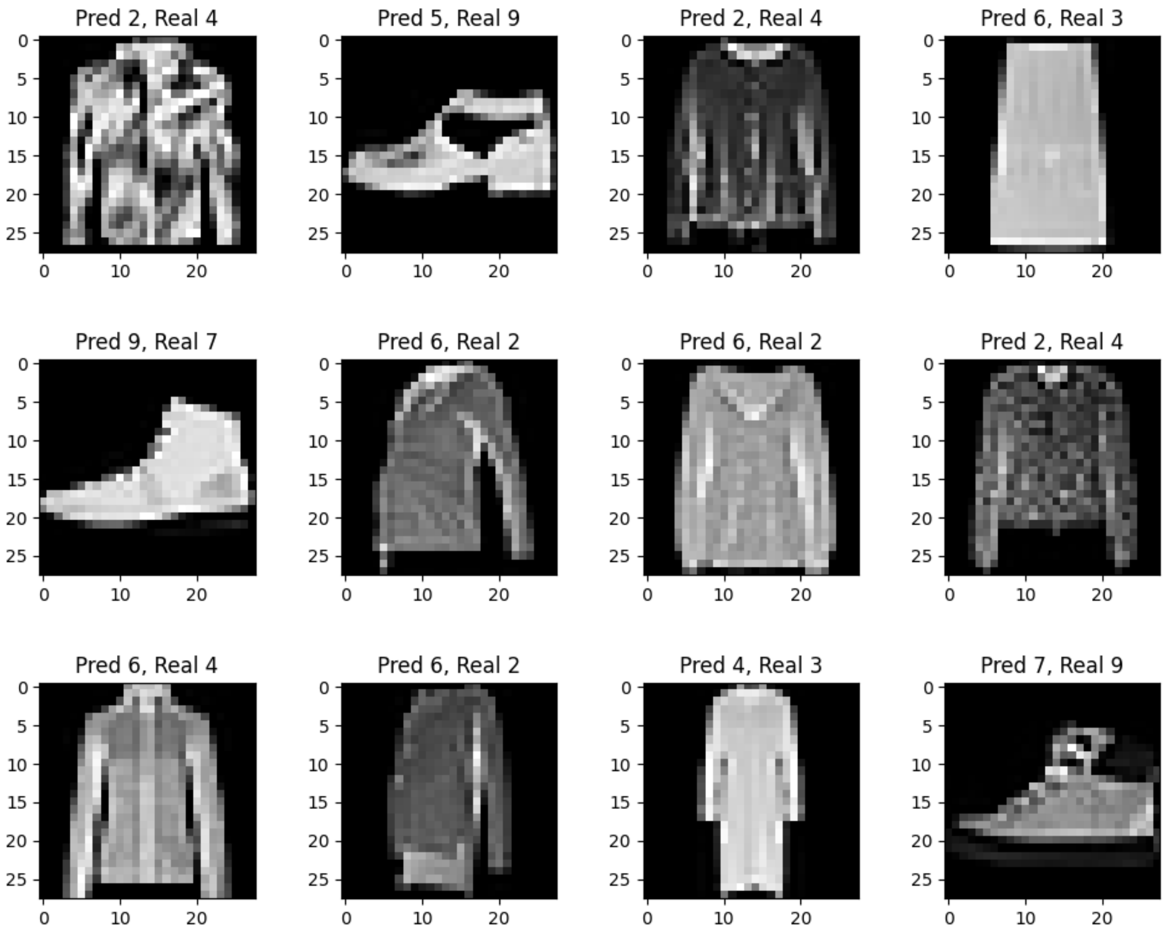

Figure 6: test failures

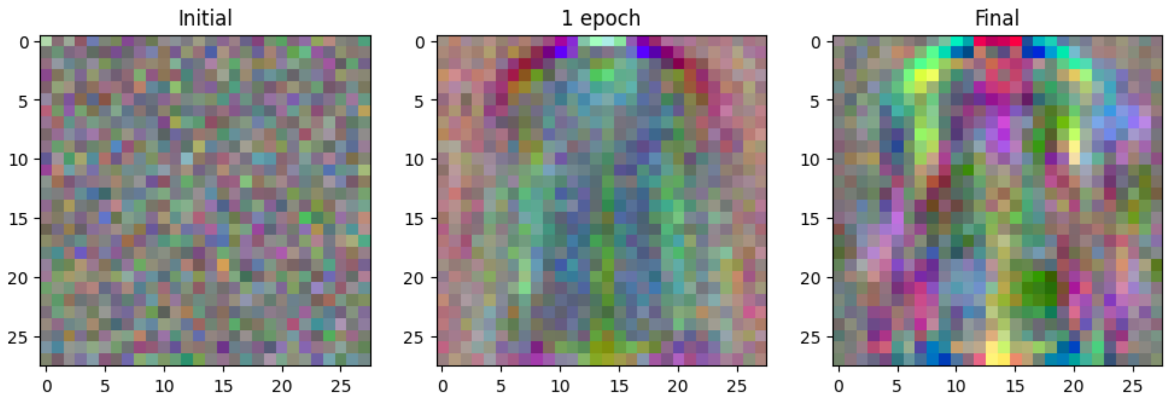

Figure 7: weights

Further, figure 7 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 87.21% accuracy on the validation set. The final one depicts the weights after training for 10 epochs and it has notable features. It indicates that the neural network is somehow learning a certain pattern from the data distribution.

7 Conclusions

We have already seen that a simple three-layer network is able to reach a 88.64% accuracy on Fashion-Mnist dataset, possibly outperforms any traditional method one can imagine. Yet this is not its limit. Within three 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.