Multi-layer Perceptron for Handwritten Number Detection

Yanjun Shao [*] [1]

Abstract

This is the first project for Computer Vision 2022 Spring. In this project, we basically built a multi-layer perceptron with Python and Numpy. We visualized the training and testing loss as well as error. Also, we visualized the parameters of each layer with dimension reduction techniques.

1. Descriptions

My implementation tried to imitate the architecture of Pytorch, which is much more readable but might lose efficiency during training and inference. The entire package is listed under the directory ./dnn. The code can be found at github.com/super-dainiu/mnist and the trained model can be found under the guidance of *README.md*.

To be more specific,

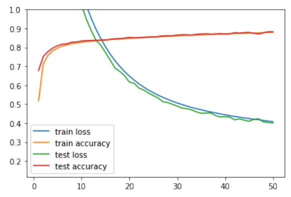
- *data.py* provides a function to load the MNIST dataset, which is located in ./mnist_data.
- *functional.py* provides all of the loss function, activation functions and linear layers required for this project. Specially, the Softmax classifier connected to the CrossEntropyLoss does not involve in the back-propagation, because the gradient of these two combined is easier to compute.
- *nn.py* provides a Module template for our net.
- *optim.py* provides an SGD optimizer.
- *train.py* provides the details of training procedure.
- *utils.py* provides a DataLoader() class analogous to the torch.utils.data.DataLoader().

2. Parameter Searching

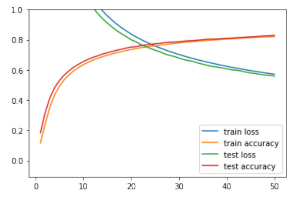
The best model selected was a model with [Linear(784, 256), ReLU(), Linear(256, 10), Softmax()], learning_rate=0.5, weight_decay=1e-4. The model was trained with 60000 items for training, 10000 items for testing, batch_size=64, num_epochs=50 using the traditional MNIST dataset.

2.1. ReLU v.s Sigmoid

With experiment under controlled variables, we discover that ReLU() is better than Sigmoid() after several epochs.



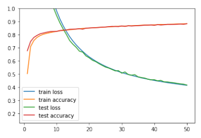
(a) ReLU



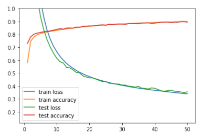
(b) Sigmoid

2.2. Learning rate (0.05 v.s 0.1 v.s 0.5)

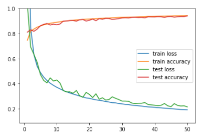
With experiment under controlled variables, we discover that learning rate should be set larger to converge more quickly.



(c) lr=0.05



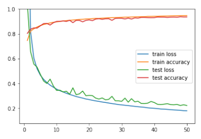
(d) lr=0.1



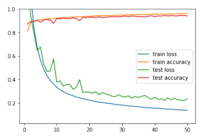
(e) lr=0.5

2.3. Hidden Layers (64 v.s 128 v.s 256)

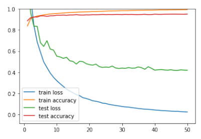
With experiment under controlled variables, we discover that larger hidden layers can fit the training data better, but larger hidden layers also cause overfitting.



(f) hiddens=64



(g) hiddens=128



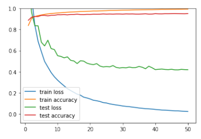
(h) hiddens=256

[1]Department of Data Science, Fudan University, China. Correspondence to: Yanjun Shao <19307110036@fudan.edu.cn>.

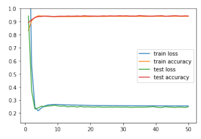
Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

2.4. Weight Decay (None v.s 1e-3 v.s 1e-4)

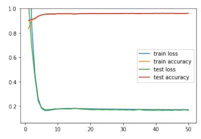
With experiment under controlled variables, we discover that huge weight decay may lead to underfitting, but weight decay is helpful to avoid overfitting..



(i) decay=None



(j) decay=1e-3



(k) decay=1e-4

3. Visualization

The weight of the first layer of size (784, 256) is reshaped and reduced to (36, 28, 28) with the technique of principle component analysis (that is, taking the top 36 PCs).

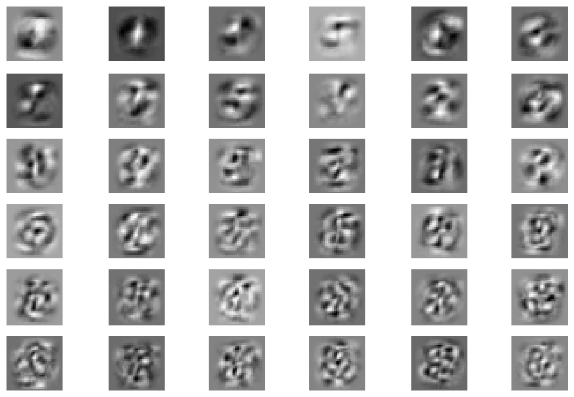


Figure 1. Layer 1

The weight of the second layer of size (256, 10) is reshaped to (10, 256). It is worth noticing that this layer shares no relationship with the spatial structure of the image. Therefore, the layer should be displayed as an 1d array.

Figure 2. Layer 2