HW1: Neural Networks Report

王昊文

2021 年 11 月 8 日

1 Overview

The whole homework strictly follows the object-orientated philosophy, which treats *datatset*, *data loader*, *trainer*, *logger* as different objects, so that it can be very convinient to write your own custom objects. You can combine all your settings and put them under `config.json` inside the root folder.

Also, you will find that the structure is highly similar to pytorch template, but actually without any Pytorch! This gives me the chance to implement a basic neural netork from scratch, and writing base classes.

In this homework, I will be using the **MNIST** dataset with mini-batch SGD.

2 Details

2.1 Data

In the MNIST dataset, we have a total of 60000 of training data, and 10000 testing data. We use a train:val split of 7:3, which indicates we are going to have 42000 training samples and 18000 validation samples. This data is about classifying hand-written digits. Each input feature is $28 \times 28 = 784$ dimension. Since we are not using convolution, we will reshape the input features as 784×1 for convinience. The train:test ratio can be modified inside the `config.json` file under the root file.

Since we are using cross-entropy loss(configured inside json.config) , we

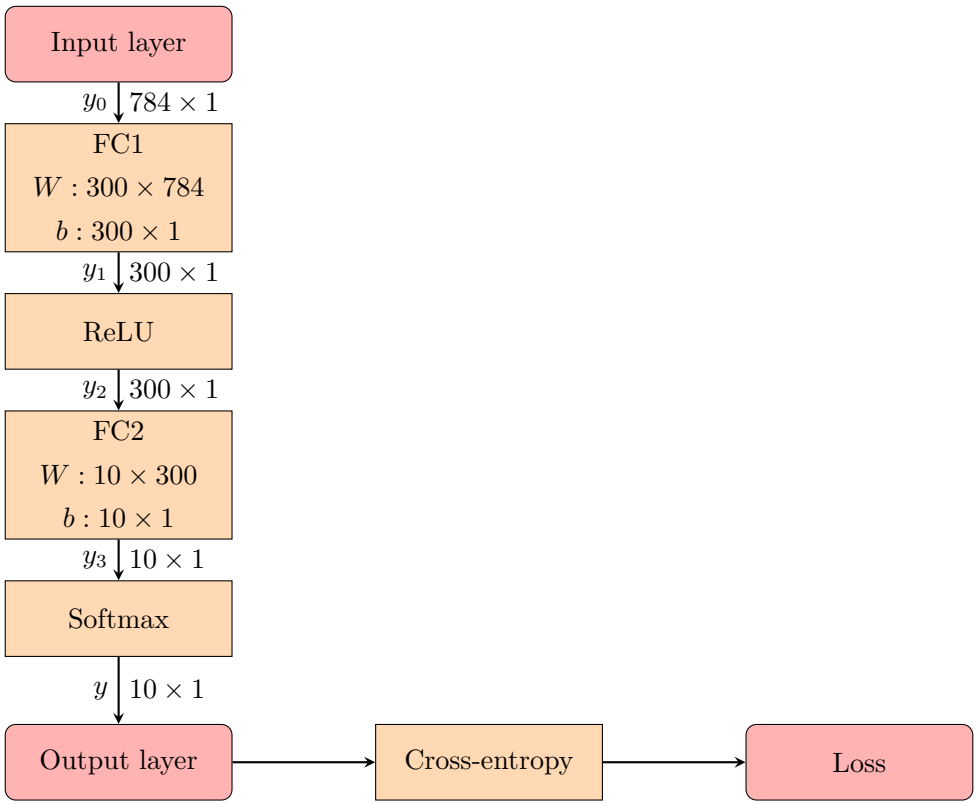

Figure 1: Architecture of our model

.

have to one-hot encode the labels. Since the dataset contains ten-digit, the lables will be one-hot encoded into a vector of shape (10×1), with the true lable be 1, and others be 0.

All features will be normalized by a factor of 255, so that the weights will not be affected by a super large value. Note, the reading of dataset is handled be the *MNIST dataset* class and the preprocessing and spliting by the *MNIST data loader* class.

2.2 Architecture

The model architecture is shown in figure 1. Note that is defined under `model.py`.

2.3 Implementation of Feed Forward and Back propogation

Below are some details of implementaion:

```
import numpy as np
def forward(self, x):
    # Input layer
    self.tensors['y0'] = x
    # Input layer -> fc1
    self.tensors['y1'] = np.matmul(self.fc1['W'], self.tensors['y0']) +
self.fc1['b']
    # ReLU layer
    self.tensors['y2'] = F.relu(self.tensors['y1'])
    # Output Layer
    self.tensors['y3'] = np.matmul(self.fc2['W'], self.tensors['y2']) +
self.fc2['b']
    # Output layer
    self.tensors['y'] = F.softmax(self.tensors['y3'])
    # return classified results
    return self.tensors['y']

```

Listing 1: feed forward

The above code will match figure 1.

```
def backward(self, L_bp, m_batch):
    """
       Implement back prop process and compute the gradients
       :m_batch: mini batch size
       :L_bp: Backward pass of the loss
       :return: gradients
    """
    gradients = {}
    # The upstream gradient from the loss with softmax loss
    d_y = L_bp
    # Calculate the local gradients and multiply with upstream gradient
    gradients['dW2'] = (1. / m_batch) * np.matmul(d_y, self.tensors['y2'
].T)
    gradients['db2'] = (1. / m_batch) * np.sum(d_y, axis=1, keepdims=
True)
    # Calculate gradient of ReLU
    d_y2 = np.matmul(self.fc2['W'].T, d_y)
    d_y1 = np.multiply(d_y2, np.where(self.tensors['y1'] <= 0, 0, 1))
    gradients['dW1'] =  (1. / m_batch) * np.matmul(d_y1, self.tensors['
y0'].T)
```

```
        gradients['db1'] =  (1. / m_batch) * np.sum(d_y1, axis=1, keepdims=True)
        return gradients

```

Listing 2: back propogation

`L_bp` is the back propogation value from the cross-entropy loss. Note that when softmax and cross-entropy combined, the backward pass is only the true-class label, which is the one-hot encoded vector, mimus the predicted softmax score.

All the gradients are calculated by multiplying the upstream gradient and the local gradient. Here we list out the local gradients.

1. $\frac{\partial}{\partial W}(Wx + b) = x^\intercal$
2. $\frac{\partial}{\partial b}(Wx + b)$
   The resulting derivative will be a column vector with all ones in it. Assume the matrix Wx has shape $j \times k$, then the vector should be $k \times 1$. Hence if we calculate the upstream gradient times the local gradient, it would be equivalent of summing over the rows of the upstream gradient.
3. $\frac{\partial L}{\partial o_i} = p_i - y_i$
   Where L is the cross-entropy loss, o_i is the hidden layer value before the softmax activation, p_i is the predicted score of the softmax activation, and y_i is the true label's score for class i, that is 1 for true class and 0 for negative.
4. $\frac{\partial}{\partial x_i} \max(0, x_i)$
   $$= \begin{cases} 0, & \text{if } x_i < 0 \\ 1, & \text{if } x_i > 0 \end{cases}$$

2.4 Training and validation

As shown in figure 2 and 3, we can see the change in loss and accuracy for training and validation at the training stage, when training with 100 epochs using SGD with a mini-batch size of 128. We can see from the

accuracy graph that the training accuracy increases, while the validation accuracy flattens near 98%. This suggests that our model starts to overfit after 40 epochs, as we might not get an accuracy over 98% at the testing stage. This hypothesis can be proved at the testing stage.

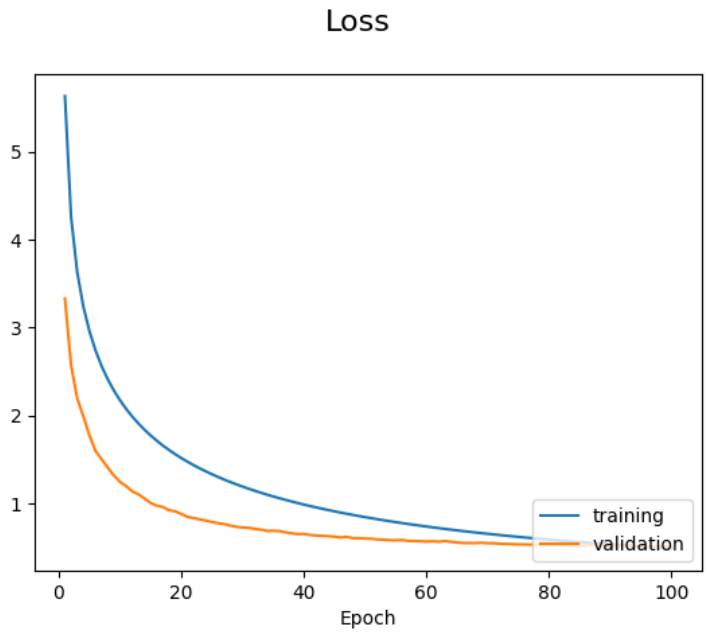

Figure 2: Loss curve, Training v.s. Validation

2.5 Performace

I have provided a best model `best.npz` and the config `config.json` under the root folder. To run `test.py`, type `python test.py -c config.json -r best.npz` and run. As suggested in the previous section, You should get approximately 97% - 98% of testing accuracy.

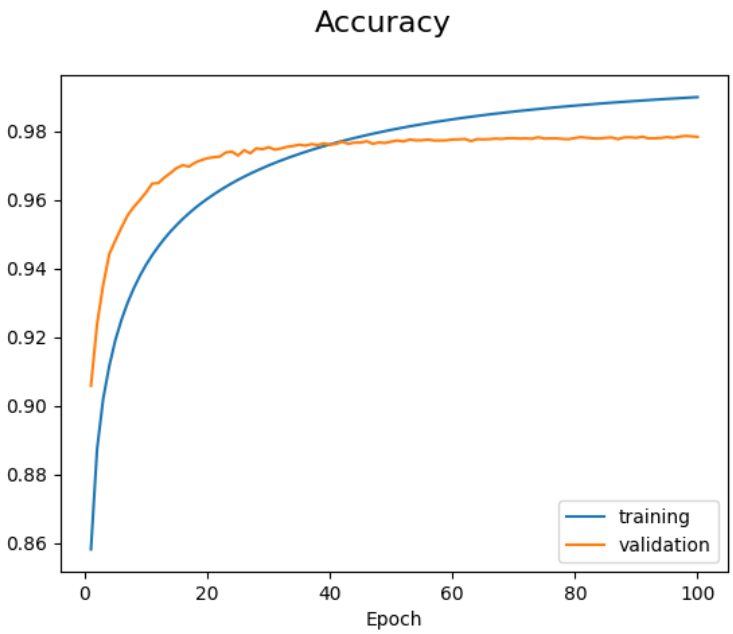

Figure 3: Accuracy curve, Training v.s. Validation

3 Short questions

3.1 Deep neural network v.s. accuracy

If we use a very deep NN with a large number of neurons, will the accuracy increase? Well, sure, the *training accuracy* will increase, since the variance is getting larger as we get deeper(under the assumption that we are using activations for every fully-connected layers). However, it does not guarantee better *testing accuracy*, as it might be just overfitting to the distribution of the training data. We must monitor the bias-variance trade-off in order to gain better performance, not just by increasing the variance of the model.

3.2 Why validation?

Why do we need to valudate our model? If we don't validate our model, then there is no other way to check for overfitting at the training stage. Note

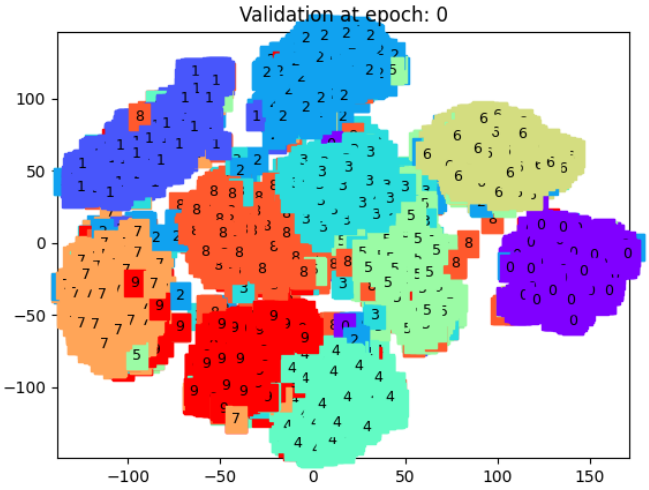

Figure 4: t-SNE visualization of validation data at epoch 0

that we are **not allowed** to use the testing data while training.

4 Bonus: t-SNE results

Figure 4, 5, 6 shows the t-SNE results on our data on the validation stage, during different epochs. Figure 7 shows the t-SNE results on the testing data.

Since the visualization process consumes a lot of time, I commented this part out from my code, and hence sklearn is not required to run my code.

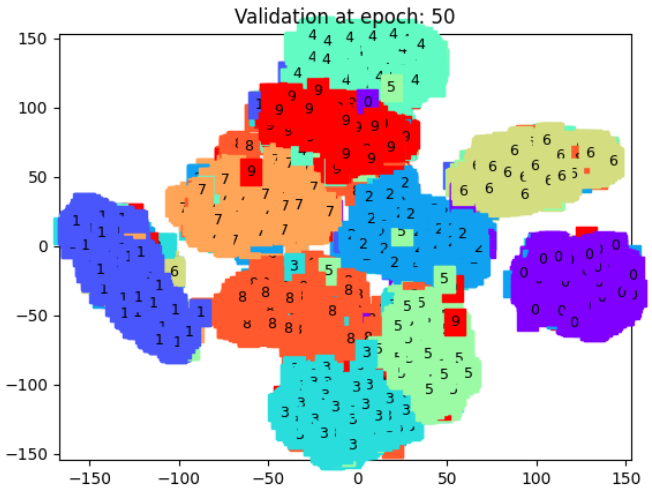

Figure 5: t-SNE visualization of validation data at epoch 50

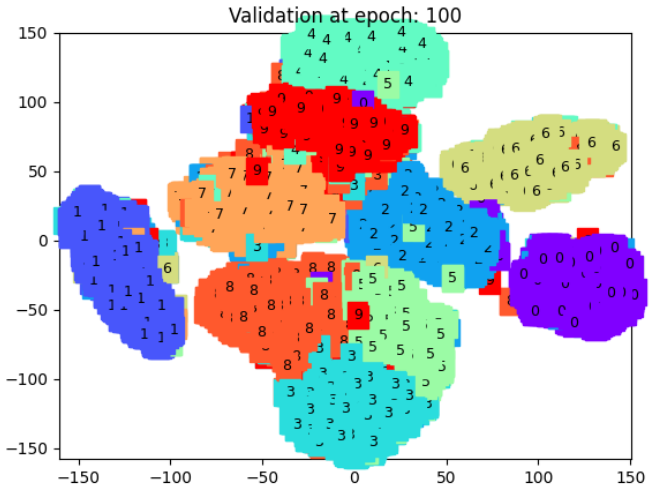

Figure 6: t-SNE visualization of validation data at epoch 100

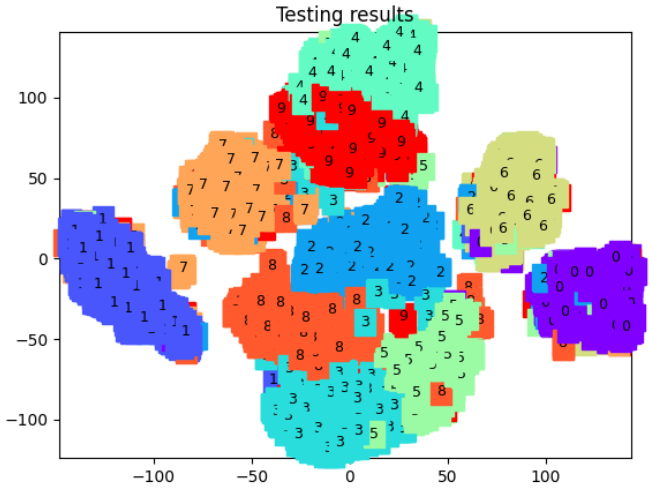

Figure 7: t-SNE visualization of testing data