Deep Learning 2020 **(Due: [6.11.2020])**

Submission Assignment #[1]

Instructor: Jakub Tomczak *Name:* [XingKai Wang], *Netid:* [xwg510]

1 Problem statement

The production of artificial neural networks (ANN) is inspired by the working mechanism of the human brain and plays an important role in the field of deep learning. Neural network is an effective method used to solve problems in artificial intelligence, such as excellent performance in prediction and classification problems. The basic structure of neural network is mainly composed of input layer, hidden layer and output layer. Among them, the hidden layer contains many interconnected neurons, and each neuron is a mathematical expression composed of weights and activation functions. Usually, the performance of a neural network on certain specific tasks is represented by the loss function, and we want the value of loss function to be as small as possible. Backpropagation is commonly used in optimizing neural networks. Backpropagation is based on the gradient descent method to calculate the optimization direction of the weights in the neural network to reduce the loss function.

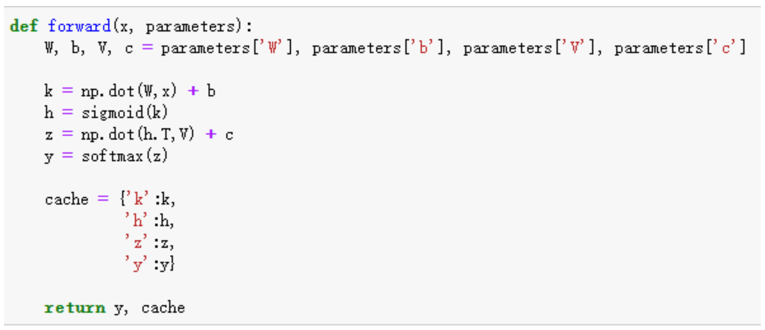
In this research, we built a tensor neural network which vectorize the operation of the neural network. The purpose of this research is to optimize the weights in the neural network through the stochastic gradient descent method to reduce the loss function. And the validation data is used to find the optimal hyperparameters, learning rate in this case. Finally, the best set of hypermeters will be chosen to train the neural network and do the test to see how well the neural network can learn. The cross entropy function(Loss) is used as our loss function, and we use the MNIST dataset to train and test our model.

$$Loss = -\sum(xi \cdot \log yi + (1 - xi) \cdot \log(1 - yi)) \tag{1.1}$$

2 Methodology

The initial approach of this research was vectorizing the operation of the neural network using numpy. Different equations are created to implement different parts of the neural network, such as forward propagation, back propagation, and final model building. The neural network uses a single hidden layer of 300 nodes, with the Sigmoid activation function for the hidden layer, and with the softmax activation function for the output layer.

firstly, the initial weights W and V are set to random sampling on the normal distribution, and the initial bias b, c are set to 0. In the forward propagation process, the input x is linearly combined with the weight W and the bias b ($k = W \cdot x + b$), and then the sigmoid activation function is used on it ($h = sigmoid(k)$), after that, the result of sigmod function is linearly combined with the weights V and bias c ($z = V \cdot h + c$), and finally the softmax activation function is used to output the result ($y = softmax(z)$).

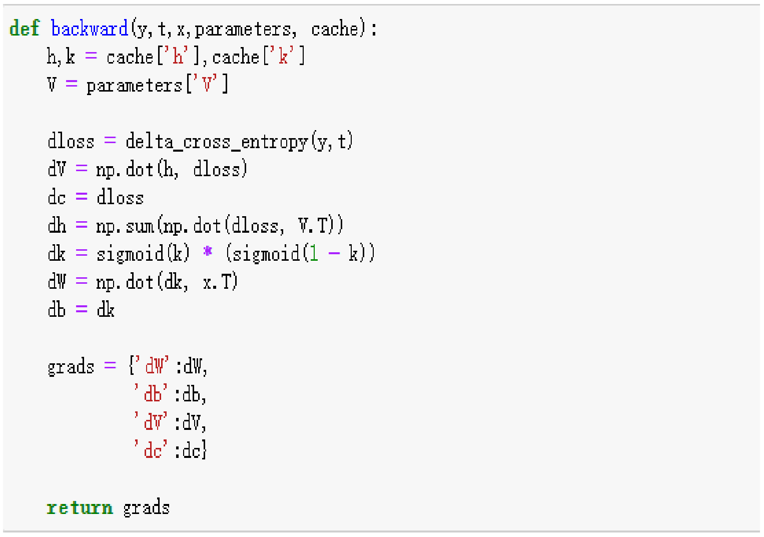


```python
def forward(x, parameters):
    W, b, V, c = parameters['W'], parameters['b'], parameters['V'], parameters['c']

    k = np.dot(W,x) + b
    h = sigmoid(k)
    z = np.dot(h.T,V) + c
    y = softmax(z)

    cache = {'k' :k,
             'h' :h,
             'z' :z,
             'y' :y}

    return y, cache
```

Figure 1: implementation of forward propagation

The basic idea of backpropagation is to use chain rule to calculate the derivative of parameters W, V, b, c. For example, when calculating derivative of the weight dV, the derivative of loss function and the derivative of the product of linearly combination is needed. The equation below shows how to calculate derivative of weight V(dV) by using chain rule. All derivative of other parameters W, b,c can also be calculated by apply chain rule.

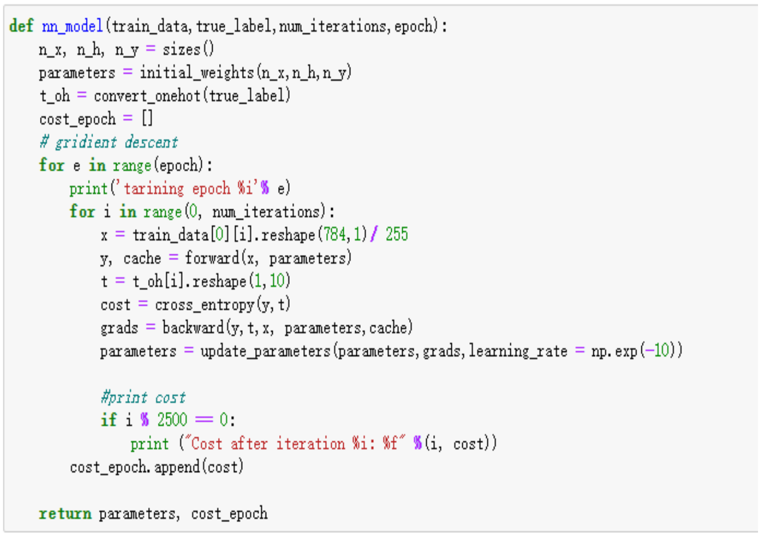
$$dV = \frac{\partial l}{\partial V} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial V} \tag{2.1}$$



```
def backward(y,t,x,parameters, cache):
    h,k = cache['h'],cache['k']
    V = parameters['V']

    dloss = delta_cross_entropy(y,t)
    dV = np.dot(h, dloss)
    dc = dloss
    dh = np.sum(np.dot(dloss, V.T))
    dk = sigmoid(k) * (sigmoid(1 - k))
    dW = np.dot(dk, x.T)
    db = dk

    grads = {'dW':dW,
             'db':db,
             'dV':dV,
             'dc':dc}

    return grads
```

Figure 2: implementation of backward propagation

After calculating all the derivatives, the parameters can be updated by the rule $W = W - \alpha \cdot dW$, where the α is the learning rate that determines the step size taken for each descent in the process of gradient descent. In this research, stochastic gradient decent (SGD) which calculating the loss over one instance at a time is used. By doing number of iterations of forward propagation, backward propagation and parameters update, the best set of parameters can be found and the final model will be trained.



```
def nn_model(train_data,true_label,num_iterations,epoch):
    n_x, n_h, n_y = sizes()
    parameters = initial_weights(n_x,n_h,n_y)
    t_oh = convert_onehot(true_label)
    cost_epoch = []
    # gridient descent
    for e in range(epoch):
        print('tarining epoch %i'% e)
        for i in range(0, num_iterations):
            x = train_data[0][i].reshape(784,1)/ 255
            y, cache = forward(x, parameters)
            t = t_oh[i].reshape(1,10)
            cost = cross_entropy(y,t)
            grads = backward(y,t,x, parameters,cache)
            parameters = update_parameters(parameters,grads,learning_rate = np.exp(-10))

            #print cost
            if i % 2500 == 0:
                print ("Cost after iteration %i: %f" %(i, cost))
        cost_epoch.append(cost)

    return parameters, cost_epoch
```

Figure 3: implementation of neural network

3 Experiments

In this neural network, the activation function uses the sigmoid function and the softmax function, and the initial weights W and V are set to random sampling on the normal distribution, and the initial bias b, c are set to 0. And the neural network will be trained limit to 5 epochs.

Firstly, the training loss per epoch and the validation loss per epoch will be compared to verify the difference between train data and validation data. Secondly, the SGD will be run 3 times and the average loss per epoch and the standard deviation will be plot. What's more, the learning rate will be set to 0.01, 0.0001 and 0.05 to train the neural network, and the validation data will be used to calculate the accuracy to analyze how different learning rate influences the final performance of neural network. Finally, the learning rate with best performance will be chosen to train the model with full train data and do the test on the final test data. The result of final accuracy on the test data will show how well this neural network can learn if we limit it to 5 epochs.

$$sigmoid = 1/(1 + e^{-x}) \tag{3.1}$$

$$softmax = e^{xi}/(\sum e^{xi}) \tag{3.2}$$

4 Results and discussion

Table 1 presents the loss per epoch on train data and validation data. These values are the average loss after the training of the neural network per epoch. The first very significant result can be observed: after each epoch training, the loss has a decrease trend for both training data and validation data, which shows that the neural network is trained, the performance of neural network has obvious promote. Compared the loss per epoch of train data and validation data, it is easily to see the loss of validation data are higher than the loss of train data except epoch5. This makes sense because after each epoch training, the neural network is much more familiar with the training data than the validation data. In other words, the neural network keeps more information about train data.

Table 1: loss per epoch on train data and validation data

datasets	epoch 1	epoch 2	epoch3	epoch4	epoch5
train_data	1.02	0.58	0.48	0.44	0.38
validation_data	1.83	1.58	1.07	1.02	0.22

Figure4 elaborate the average loss with standard deviation on the train data. It is obviously that the average loss on the train data has a large standard deviation, It means that there is a big difference between most of the values and the average value. The reason for this situation may be due to the stochastic gradient descent method, which calculates the gradient for one instance at a time.

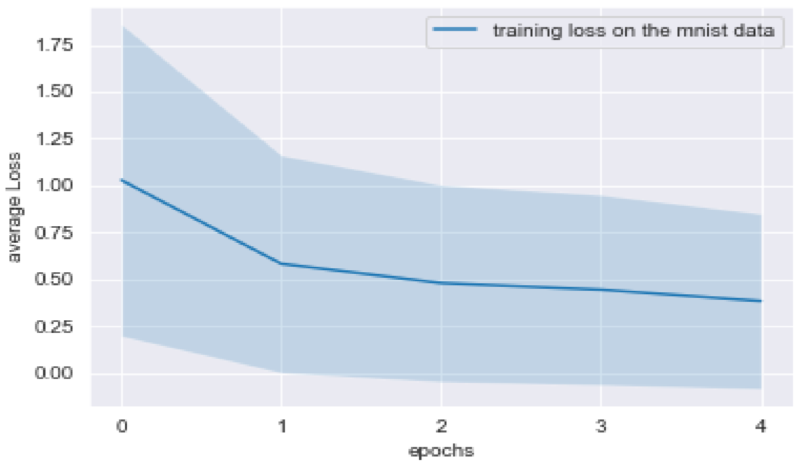


Figure 4: average loss with standard deviation on mnist data

Table2 presents how the different learning rate influences the final performance of the neural network. 0.0001, 0.01, 0.05 three learning rate are compared. It is obviously that the neural network has the best performance when the learning rate is 0.01, which has the highest accuracy of 92.48% on the validation data. Figure5, figure6, figure7 show the training loss with different learning rate. When the learning rate is set to 0.0001, the training loss decrease slowly from around 1.0 to 0.4, the low learning rate leads to slow model learning and convergence speed. Therefore, within 5 epochs, the model did not completely finish the training, so the final accuracy rate was low, only 56.76%. When the learning rate is too large, 0.05 for example we can obviously see that the training loss even increases after epoch1, which shows that the step length when we perform gradient descent is too long, which reduces the efficiency of convergence. 0.01 is the most suitable learning rate. Not only does the validation data have the highest accuracy, the training loss can also converge to a lower value within 5 epochs.

Table 2: accuracy on the validation data

learning rate	0.0001	0.01	0.05
accuracy	56.76%	92.48%	79.12%

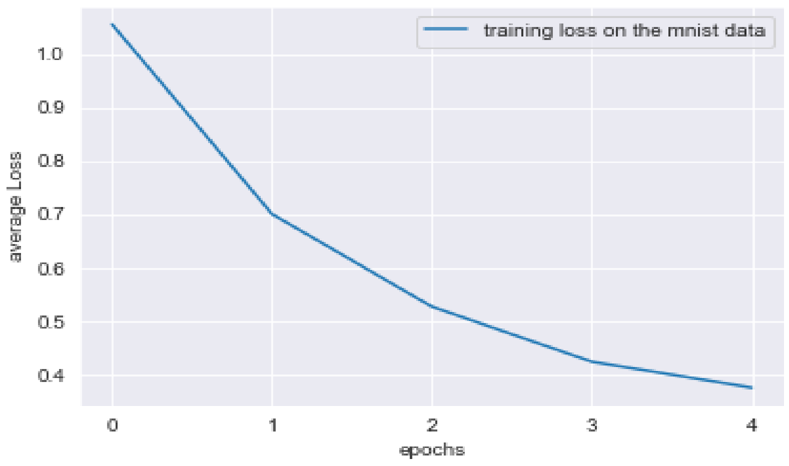


Figure 5: average loss on the mnist data with 0.0001 learning rate

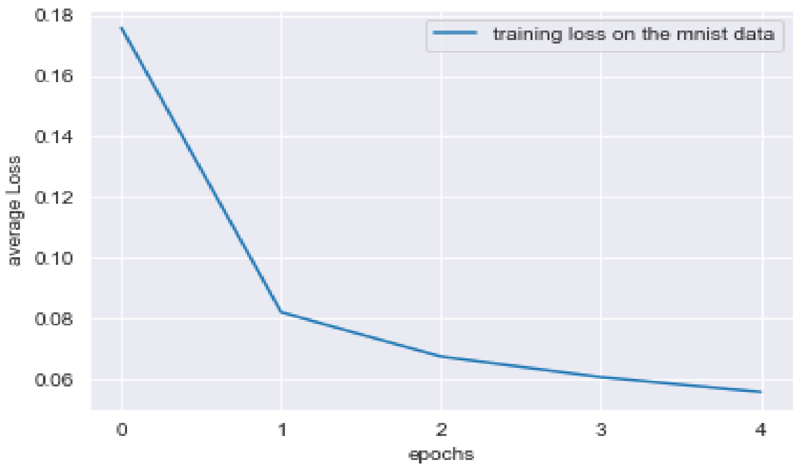


Figure 6: average loss on the mnist data with 0.01 learning rate

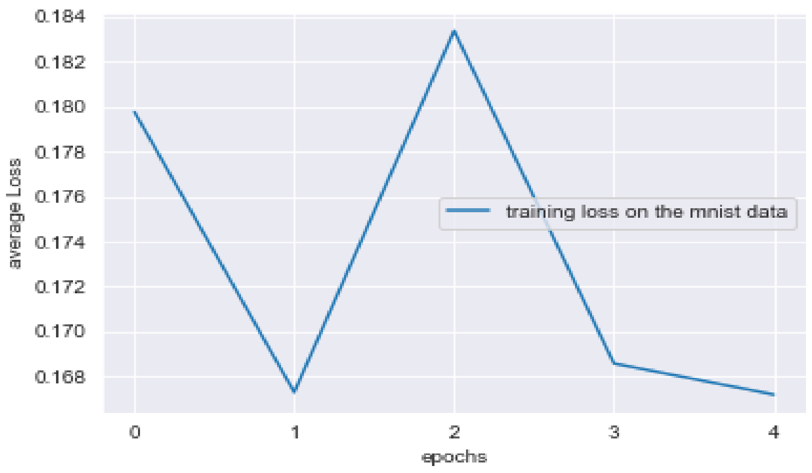


Figure 7: average loss on the mnist data with 0.05 learning rate

According to the results above, the hyperparameter of learning rate is decided to set to 0.01 to train our model and do the final test on the test data. Table3 shows the final accuracy on the test data is 89.89%, This shows that when the learning rate is 0.01, this neural network has a relatively good performance after 5 epochs of training. Regarding how to optimize neural networks in the future, I think the choice of hyperparameters is particularly important for the performance of neural networks, so we should make full use of the validation data for testing to find the optimal set of hyperparameters, such as the learning rate, batch size, etc. Secondly, the selection of the initial weight can also be adjusted, and the use of different initialization methods may have an impact on the training of the model. Finally, extending the training time and increasing the number of training data can make the model get more adequate training and can also improve the performance of the neural network.

Table 3: accuracy on the test data

data	accuracy
test_data	89.87%

5 Question 1

when i = j:

$$\frac{\partial yi}{\partial Oj} = \frac{e^{Oi} \cdot \sum e^{Oj} - e^{Oj} \cdot e^{Oi}}{(\sum e^{Oj})^2} = \frac{e^{Oi}}{\sum e^{Oi}} \cdot \frac{\sum e^{Oi} - e^{Oj}}{\sum e^{Oj}} = yi \cdot (1 - yj) \tag{5.1}$$

when i≠ j:

$$\frac{\partial yi}{\partial Oj} = \frac{0 - e^{Oj} \cdot e^{Oi}}{(\sum e^{Oj})^2} = \frac{-e^{Oj}}{\sum e^{Oi}} \cdot \frac{e^{Oi}}{\sum e^{Oi}} = -yj \cdot yi \tag{5.2}$$

Therefore, the derivative of yi can be concluded:

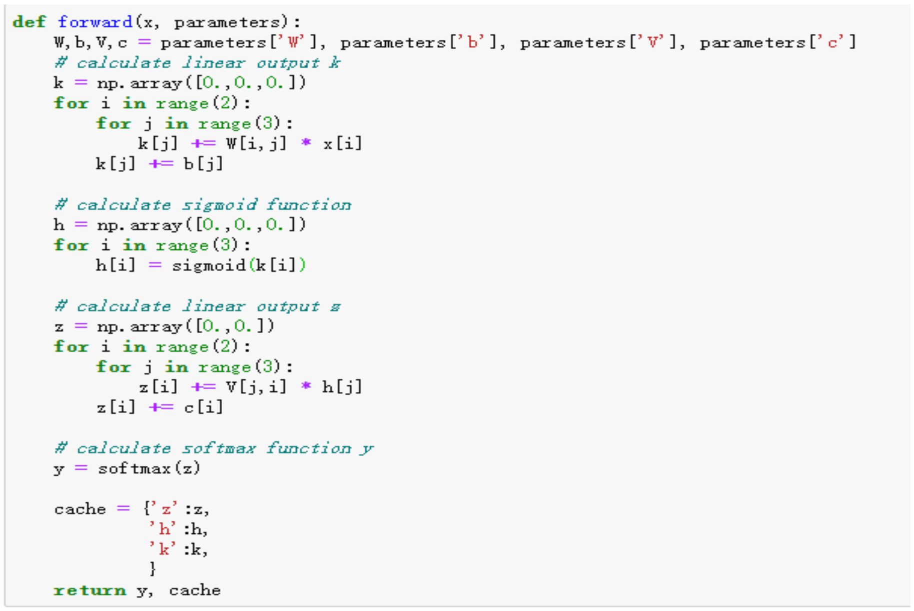
$$\frac{\partial yi}{\partial Oj} = \begin{cases} yi \cdot (1 - yj) & \text{i=j} \\ -yj \cdot yi & \text{i≠j} \end{cases} \tag{5.3}$$

The derivative of l:

$$\frac{\partial l}{\partial yi} = \frac{\partial - \log yi}{\partial yi} = -\frac{1}{yi} \tag{5.4}$$

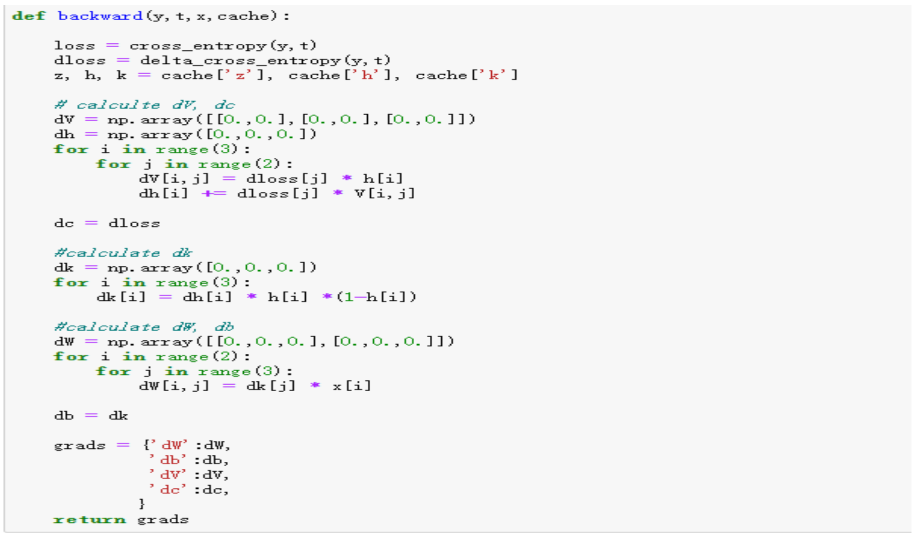
6 Question 2

I this question, the neural network in scalar term is built. The code of forward propagation ,backward propagation and the derivative of all parameters are shown in figure8, figure9 and figure10 respectively.



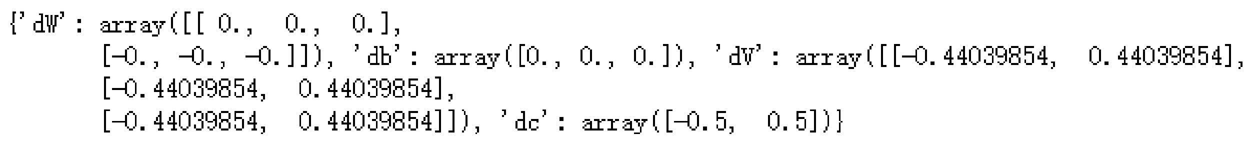
```
def forward(x, parameters):
    W,b,V,c = parameters['W'], parameters['b'], parameters['V'], parameters['c']
    # calculate linear output k
    k = np.array([0.,0.,0.])
    for i in range(2):
        for j in range(3):
            k[j] += W[i,j] * x[i]
        k[j] += b[j]

    # calculate sigmoid function
    h = np.array([0.,0.,0.])
    for i in range(3):
        h[i] = sigmoid(k[i])

    # calculate linear output z
    z = np.array([0.,0.])
    for i in range(2):
        for j in range(3):
            z[i] += V[j,i] * h[j]
        z[i] += c[i]

    # calculate softmax function y
    y = softmax(z)

    cache = {'z':z,
             'h':h,
             'k':k,
            }
    return y, cache
```

Figure 8: implementation of forward propagation



```
def backward(y,t,x,cache):

    loss = cross_entropy(y,t)
    dloss = delta_cross_entropy(y,t)
    z, h, k = cache['z'], cache['h'], cache['k']

    # calculte dV, dc
    dV = np.array([[0.,0.],[0.,0.],[0.,0.]])
    dh = np.array([0.,0.,0.])
    for i in range(3):
        for j in range(2):
            dV[i,j] = dloss[j] * h[i]
            dh[i] += dloss[j] * V[i,j]

    dc = dloss

    #calculate dk
    dk = np.array([0.,0.,0.])
    for i in range(3):
        dk[i] = dh[i] * h[i] *(1-h[i])

    #calculate dW, db
    dW = np.array([[0.,0.,0.],[0.,0.,0.]])
    for i in range(2):
        for j in range(3):
            dW[i,j] = dk[j] * x[i]

    db = dk

    grads = {'dW':dW,
             'db':db,
             'dV':dV,
             'dc':dc,
            }
    return grads
```

Figure 9: implementation of backward propagation



```
{'dW': array([[ 0.,  0.,  0.],
       [-0., -0., -0.]]), 'db': array([0., 0., 0.]), 'dV': array([[-0.44039854,  0.44039854],
       [-0.44039854,  0.44039854],
       [-0.44039854,  0.44039854]]), 'dc': array([-0.5,  0.5])}
```

Figure 10: the derivative of all parameters

7 Question 3

In this question, the synthetic data will be input to train the neural network. And the stochastic gradient decent will be used to calculate the gradient and update all parameters. The initial weight W,V are random sampled from normal distribution and bias b,c are set to 0. The learning rate is e^{-4} in the training process. Figure11 presents that after 60 epochs, the training loss decrease from 0.7 to around 0.1.

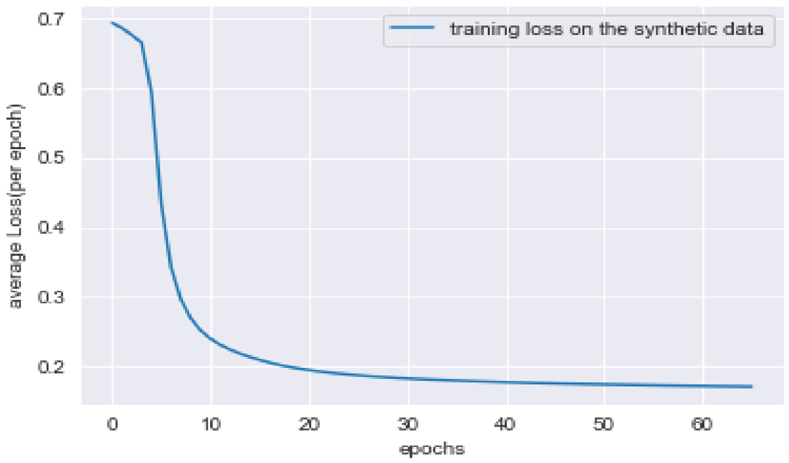
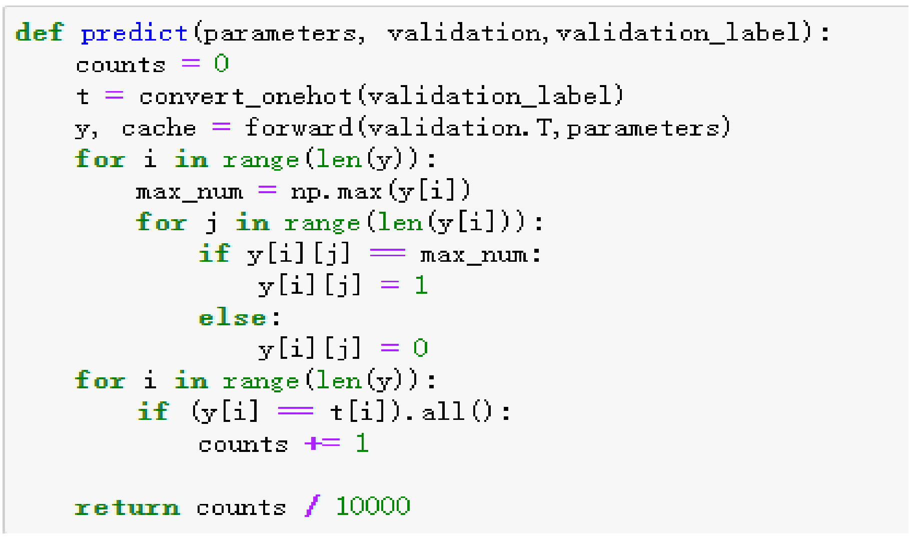


Figure 11: mean training loss on synthetic data

8 A code snippet



```
def predict(parameters, validation,validation_label):
    counts = 0
    t = convert_onehot(validation_label)
    y, cache = forward(validation.T,parameters)
    for i in range(len(y)):
        max_num = np.max(y[i])
        for j in range(len(y[i])):
            if y[i][j] == max_num:
                y[i][j] = 1
            else:
                y[i][j] = 0
    for i in range(len(y)):
        if (y[i] == t[i]).all():
            counts += 1

    return counts / 10000
```

Figure 12: implementation of prediction and accuracy calculation

References

Oludare Isaac Abiodun (2018) ”State-of-the-art in artificial neural network applications: A survey”.