toy_nn

November 9, 2018

1 Two-layer Neural Network Workbook for CS145 Homework 3

PRINT YOUR NAME AND UID HERE!
NAME: [Lee, Hun]
UID: [604958834]

Please follow the notebook linearly to implement a two layer neural network.
Please print out the workbook entirely when completed.
The goal of this workbook is to give you experience with training a two layer neural network.

Import libraries and define relative error function, which is used to check results later.

```
In [1]: import random
        import numpy as np
        from cs145.data_utils import load_CIFAR10
        import matplotlib.pyplot as plt

        %matplotlib inline
        plt.rcParams['figure.figsize'] = (10.0, 8.0)
        %load_ext autoreload
        %autoreload 2

        def rel_error(x, y):
            """ returns relative error """
            return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

1.1 Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass.

In [2]:
```
from lib.neural_net import TwoLayerNet
```

In [3]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

1.1.1 Compute forward pass scores

In [4]:
```
## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
```

```
        print('Difference between your scores and correct scores:')
        print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231204052648e-08
```

1.1.2 Forward pass loss

The total loss includes data loss (MSE) and regularization loss, which is,

$$L = L_{data} + L_{reg} = \frac{1}{2N}\sum_{i=1}^{N}\left(\boldsymbol{y}_{\text{pred}} - \boldsymbol{y}_{\text{target}}\right)^2 + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

More specifically in multi-class situation, if the output of neural nets from one sample is $y_{\text{pred}} = (0.1, 0.1, 0.8)$ and $y_{\text{target}} = (0, 0, 1)$ from the given label, then the MSE error will be $Error = (0.1 - 0)^2 + (0.1 - 0)^2 + (0.8 - 1)^2 = 0.06$

Implement data loss and regularization loss. In the MSE function, you also need to return the gradients which need to be passed backward. This is similar to batch gradient in linear regression. Test your implementation of loss functions. The Difference should be less than 1e-12.

```
In [5]: loss, _ = net.loss(X, y, reg=0.05)
        correct_loss_MSE = 1.8973332763705641 # check this number

        # should be very small, we get < 1e-12
        print('Difference between your loss and correct loss:')
        print(np.sum(np.abs(loss - correct_loss_MSE)))
```

```
Difference between your loss and correct loss:
0.0
```

1.1.3 Backward pass (You do not need to implemented this part)

We have already implemented the backwards pass of the neural network for you. Run the block of code to check your gradients with the gradient check utilities provided. The results should be

automatically correct (tiny relative error).

If there is a gradient error larger than 1e-8, the training for neural networks later will be negatively affected.

```
In [6]: from cs145.gradient_check import eval_numerical_gradient

        # Use numeric gradient checking to check your implementation of the backward pass.
        # If your implementation is correct, the difference between the numeric and
        # analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

        loss, grads = net.loss(X, y, reg=0.05)

        # these should all be less than 1e-8 or so
        for param_name in grads:
            f = lambda W: net.loss(X, y, reg=0.05)[0]
            param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
            print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, grad
```

```
W2 max relative error: 8.80091875172355e-11
b2 max relative error: 2.4554844805570154e-11
W1 max relative error: 1.7476665046687833e-09
b1 max relative error: 7.382451041178829e-10
```

1.1.4 Training the network

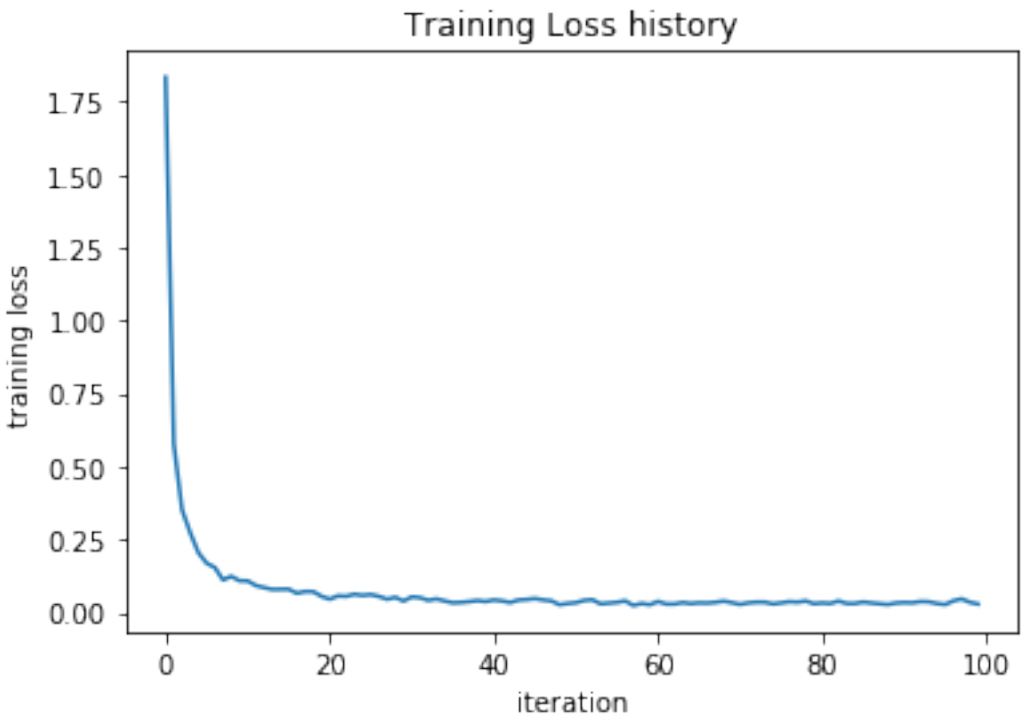
Implement neural_net.train() to train the network via stochastic gradient descent, much like the linear regression.

```
In [7]: net = init_toy_model()
        stats = net.train(X, y, X, y,
                    learning_rate=1e-1, reg=5e-6,
                    num_iters=100, verbose=False)

        print('Final training loss: ', stats['loss_history'][-1])

        # plot the loss history
        plt.plot(stats['loss_history'])
        plt.xlabel('iteration')
        plt.ylabel('training loss')
        plt.title('Training Loss history')
        plt.show()
```

```
1
Final training loss:  0.02950555626206818
```



1.2 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

```
In [8]: from cs145.data_utils import load_CIFAR10

        def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
            """
            Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
            it for the two-layer neural net classifier. These are the same steps as
            we used for the SVM, but condensed to a single function.
            """
            # Load the raw CIFAR-10 data
            cifar10_dir = './cs145/datasets/cifar-10-batches-py'
            X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

            # Subsample the data
            mask = list(range(num_training, num_training + num_validation))
            X_val = X_train[mask]
            y_val = y_train[mask]
            mask = list(range(num_training))
            X_train = X_train[mask]
            y_train = y_train[mask]
            mask = list(range(num_test))
```

```
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

1.2.1 Running SGD

If your implementation is correct, you should see a validation accuracy of around 15-18%.

```
In [72]: input_size = 32 * 32 * 3
         hidden_size = 50
         num_classes = 10



         net = TwoLayerNet(input_size, hidden_size, num_classes)

         # Train the network
         stats = net.train(X_train, y_train, X_val, y_val,
```

```
                    num_iters=1000, batch_size=200,
                    learning_rate=1e-5, learning_rate_decay=0.95,
                    reg=0.1, verbose=True)

        # Predict on the validation set
        val_acc = (net.predict(X_val) == y_val).mean()
        print('Validation accuracy: ', val_acc)

        # Save this net as the variable subopt_net for later comparison.
        subopt_net = net
```

```
iteration 0 / 1000: loss 0.50003647600486
iteration 100 / 1000: loss 0.4998172523897806
iteration 200 / 1000: loss 0.49950831945646734
iteration 300 / 1000: loss 0.4991745496257983
iteration 400 / 1000: loss 0.4988088642261781
iteration 500 / 1000: loss 0.49834701055489905
iteration 600 / 1000: loss 0.49753307147981274
iteration 700 / 1000: loss 0.49623349082986795
iteration 800 / 1000: loss 0.4948135794460125
iteration 900 / 1000: loss 0.49286215767798275
Validation accuracy:  0.158
```

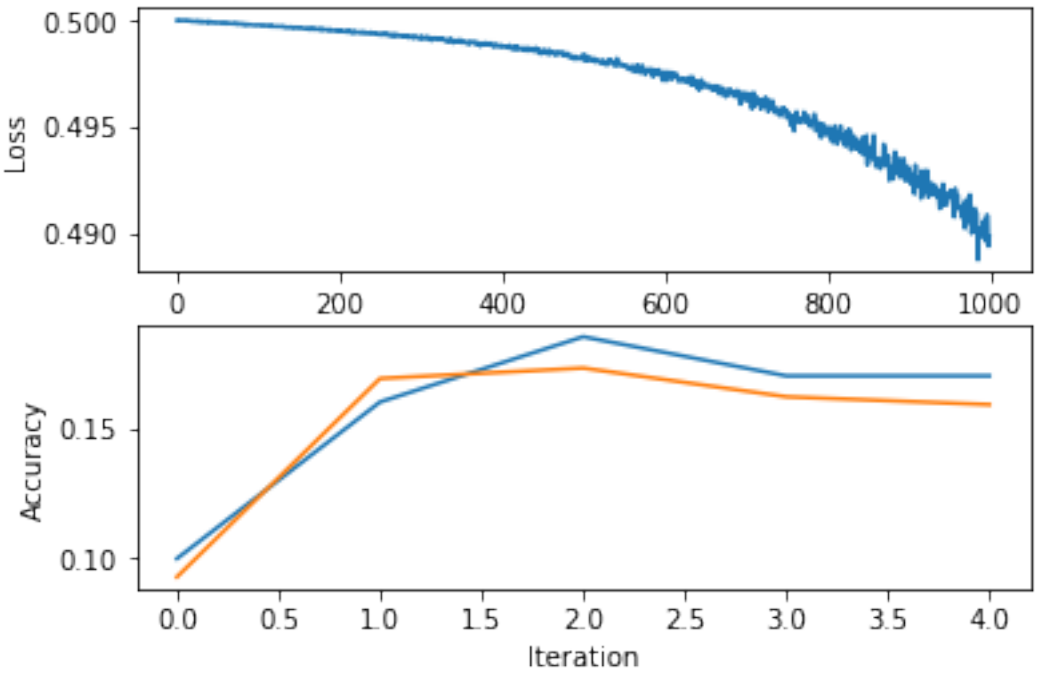
```
In [65]: stats['train_acc_history']
```

```
Out[65]: [0.07, 0.13, 0.185, 0.19, 0.18]
```

```
In [73]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(stats['loss_history'])
         plt.xlabel('Iteration')
         plt.ylabel('Loss')

         plt.subplot(2, 1, 2)
         plt.plot(stats['train_acc_history'], label='train')
         plt.plot(stats['val_acc_history'], label='val')
         plt.xlabel('Iteration')
         plt.ylabel('Accuracy')

         plt.show()
```



1.2.2 Questions:

The training accuracy isn't great. It seems even worse than simple KNN model, which is not as good as expected.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

1.2.3 Answers:

1. Some of the reasons why the training accuracy is not good is that, we are using the wrong value for hyperparameters that are producing poor result. The hyperparameters chosen above are not the best picks.

2. We can select a certain set of hyperparameters that can output the better result with higher accuracy. I decided to randomly select different values of hyperparameters to form a set and used iterations to pick the best performing set.

1.3 Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

In [55]:
```
best_net = None # store the best model into this

# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 45% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied by:
#      min(floor((X - 23%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size = 50)!
# ================================================================ #

# todo: optimal parameter search (you can use grid search by for-loops )

input_size = 32 * 32 * 3 # do not change
hidden_size = 50 # do not change
num_classes = 10 # do not change
best_valacc = 0 # do not change

# Train the network and find best parameter:
net = TwoLayerNet(input_size, hidden_size, num_classes)
for i in range(100):

    num_iters= int(random.randint(20,50) * random.randint(12,15))
    batch_size= int(random.randint(15,20) * random.randint(10,15))
    learning_rate= int(random.randint(1,10)) / int((np.power(10, random.randint(4,7))
    learning_rate_decay= int(random.randint(1,100))  /(np.power(10, random.randint(2,
    reg= random.randint(1,100) /(np.power(10, random.randint(2,3)))

    stats = net.train(X_train, y_train, X_val, y_val,
            learning_rate, learning_rate_decay,
            reg, num_iters,
            batch_size, verbose=False)

    val_acc = (net.predict(X_val) == y_val).mean()

    if(best_valacc < val_acc):
        best_valacc = val_acc
        bestIter = num_iters
        bestBatch = batch_size
        bestLrate = learning_rate
        bestLdecay = learning_rate_decay
        bestReg = reg
        best_net = net
```

```
        # Output your results
        print("== Best parameter settings ==")
        # print your best parameter setting here!
        print("num_iters={}, batch_size={}, learning_rate={}, learning_rate_decay={}, reg={}"
        print("Best accuracy on validation set: {}".format(best_valacc))
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
```

```
== Best parameter settings ==
num_iters=600, batch_size=165, learning_rate=0.0008, learning_rate_decay=0.0066, reg=0.55
Best accuracy on validation set: 0.5
```

1.3.1 Quesions

(1) What is your best parameter settings? (Output from the previous cell)
(2) What parameters did you tune? How are they changing the performance of nerural network? You can discuss any observations from the optimization.

1.3.2 Answers

1. first time i got num_iters=602, batch_size=247, learning_rate=0.0004, learning_rate_decay=0.22, reg=0.16 Best accuracy on validation set: 0.5 second time i got == Best parameter settings == num_iters=600, batch_size=165, learning_rate=0.0008, learning_rate_decay=0.0066, reg=0.55 Best accuracy on validation set: 0.5 both times are different. different combination of parameters result can produce the same accuracy rate.

2. i tuned all parameters. I noticed that iteration number does not matter much. also if batch size is too small the program failed. I also learned that learning rate around e^-5 to e^-6 is a good number. if the learning rate was not small enough along with other values the program failed. learning decay rate and reg performed bad if they were above 1 or too small. I found out that properly small values for rate_decay and reg produce better outcome. When performing the batch iteration, this reg makes backpropagation update the variable by small increment, and with small learning decay, I was able to get a good result.

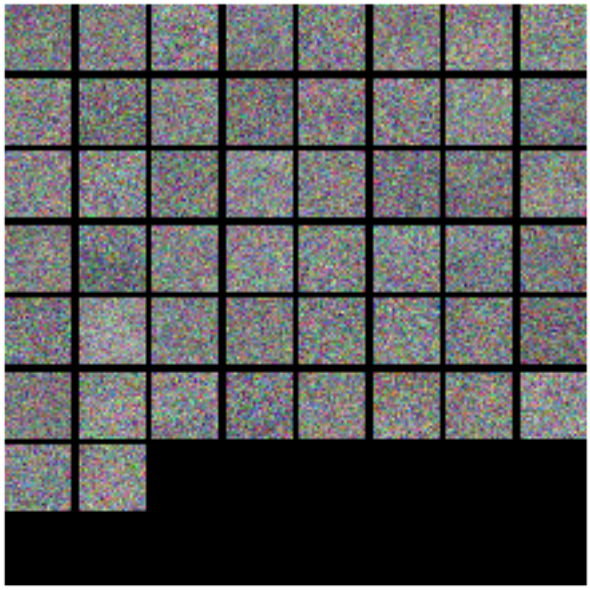
1.4 Visualize the weights of your neural networks

```
In [56]: from cs145.vis_utils import visualize_grid

        # Visualize the weights of the network

        def show_net_weights(net):
            W1 = net.params['W1']
            W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
            plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
            plt.gca().axis('off')
            plt.show()
```

```
show_net_weights(subopt_net)
show_net_weights(best_net)
```



1.4.1 Questions:

What differences do you see in the weights between the suboptimal net and the best net you arrived at? What do the weights in neural networks probably learn after training?

1.4.2 Answer:

I notice that images are more clear than the sub-optimal network. The weigths learn to minimize the errors better with optimized hyper-parameters and backpropagation. As a result, weights can better classify and analyze the inputs with more accuracy.

1.5 Evaluate on test set

```
In [57]: test_acc = (best_net.predict(X_test) == y_test).mean()
         #test_acc = (subopt_net.predict(X_test) == y_test).mean()
         print('Test accuracy: ', test_acc)

Test accuracy:  0.473
```

1.5.1 Questions:

(1) What is your test accuracy by using the best NN you have got? How much does the performance increase compared with kNN? Why can neural networks perform better than kNN?

(2) Do you have any other ideas or suggestions to further improve the performance of neural networks other than the parameters you have tried in the homework?

1.5.2 Answers:

1. the best error rate was 0.718 with K. That means KNN was 28.2% accurate. The test accuracy for Neural Network is 47.3%. That is 19.1 % increase in performance compared to KNN. NN works better beacuse KNN has limitations of having to choose K neighbors. However, NN compares all the inputs with various weights, optimize weights and bias to classify them into the class where they have the highest probability of fitting in. That is why NN works better than KNN

2. One of the ways we can use to improve data is feed more training data into NN. This can help us improve output accuracy. Also, we can try adding more hidden layers. This can substantially improve the result. Another method we can try is repeating the test with different set of initial weights. lastly, we can try using other method to calculate the error rate than MSE. This might be able to help us find the error more accurately and therefore, enable us to output better result.

1.6 Bonus Question: Change MSE Loss to Cross Entropy Loss

This is a bonus question. If you finish this (cross entropy loss) correctly, you will get **up to 20 points** (add up to your HW3 score).

Note: From grading policy of this course, your maximum points from homework are still 25 out of 100, but you can use the bonus question to make up other deduction of other assignments.

Pass output scores in networks from forward pass into softmax function. The softmax function is defined as,

$$p_j = \sigma(z_j) = \frac{e^{z_j}}{\sum_{c=1}^{C} e^{z_c}}$$

After softmax, the scores can be considered as probability of j-th class.

The cross entropy loss is defined as,

$$L = L_{\text{CE}} + L_{reg} = \frac{1}{N}\sum_{i=1}^{N} \log\left(p_{i,j}\right) + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

To take derivative of this loss, you will get the gradient as,

$$\frac{\partial L_{\text{CE}}}{\partial o_i} = p_i - y_i$$

More details about multi-class cross entropy loss, please check http://cs231n.github.io/linear-classify/ and more explanation about the derivative of cross entropy.

Change the loss from MSE to cross entropy, you only need to change you `MSE_loss(x,y)` in `TwoLayerNet.loss()` function to `softmax_loss(x,y)`.

Now you are free to use any code to show your results of the two-layer networks with newly-implemented cross entropy loss. You can use code from previous cells.

```
In [79]: best_net = None
         # Start training your networks and show your results
         input_size = 32 * 32 * 3 # do not change
         hidden_size = 50 # do not change
         num_classes = 10 # do not change
         best_valacc = 0 # do not change

         net = TwoLayerNet(input_size, hidden_size, num_classes)

         for i in range(100):

             num_iters= int(random.randint(20,50) * random.randint(12,15))
             batch_size= int(random.randint(15,20) * random.randint(10,15))
             learning_rate= int(random.randint(1,10)) / int((np.power(10, random.randint(4,7))
             learning_rate_decay= int(random.randint(1,100))  /(np.power(10, random.randint(2,
             reg= random.randint(1,100) /(np.power(10, random.randint(2,3)))

             stats = net.train(X_train, y_train, X_val, y_val,
                     learning_rate, learning_rate_decay,
                     reg, num_iters,
                     batch_size, verbose=False)
```

```
            val_acc = (net.predict(X_val) == y_val).mean()

            if(best_valacc < val_acc):
                best_valacc = val_acc
                bestIter = num_iters
                bestBatch = batch_size
                bestLrate = learning_rate
                bestLdecay = learning_rate_decay
                bestReg = reg
                best_net = net


        print('Validation accuracy: ', best_valacc)

        test_acc = (best_net.predict(X_test) == y_test).mean()
        #test_acc = (subopt_net.predict(X_test) == y_test).mean()
        print('Test accuracy: ', test_acc)
```

```
Validation accuracy:  0.503
Test accuracy:  0.486
```

In [ ]: *# the test shows a slight improvement in validation and test accuracy.*