Assignment1

April 7, 2022

1 Assignment 1: Two layer neural network

1.1 Boyuan Yao 19307110202

In this assignment, we need to implement a neural network with two layers. This report is written by jupyter notebook. So for complete version of training and testing process, please read the readme file of my github page carefully!

[52]:
```
# Some common setups
import numpy as np
from urllib import request
import gzip
import pickle
from data_utils import * # Data processing
from solver import * # Solver for updating the model
from twolayernet import * # The two layer net model
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (20.0, 16.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[2]:
```
# Load the MNIST data

data = load_mnist()
for k, v in list(data.items()):
    print(('%s: ' % k, v.shape))
```

```
('X_train: ', (55000, 784))
('y_train: ', (55000,))
('X_val: ', (5000, 784))
('y_val: ', (5000,))
('X_test: ', (10000, 784))
('y_test: ', (10000,))
```

I randomly sample 5000 images from the original MNIST dataset to validation set to validate the goodness of trained model during trainning

1.2 Part 1: Training

I have constructed the two layer network in the file ***twolayernet.py***. The network use the structure of one affine_ReLU layer (i.e. fully connected layer plus ReLU non-linear layer), and one fully connected layer to compute the final scores for each class. I use softmax loss for this network. The functions for each layer, including forward and backward propagation are in files ***layers.py*** and ***layers_utils.py***. The above three files also provide the option for different L_2 regularization strength. The model training process is included in file ***solver.py***. In ***solver.py***, there is a Solver class provide lots of options for training, including verbose, learning rate and learning rate decay, store the best parameters, batchsize, etc. You could check the annotation in the file for further information. In this part, I will show some of the experiment I've done for training part, including L_2 regularization and learning rate decay.

1.2.1 Vanilla SGD optimizer

The vanilla SGD optimizer takes the small portion of the training set as a minibatch for gradient decent process, which could largely decrease the cost on each update. If the sample quality is good enough, it will be a good estimator for the whole training set so that the noise of the gradient could be controlled, therefore, we could expect a good quality of decent direction. The *TwoLayerNet* is a class of two layer neural network. The input_size indicates the number of entries of each sample, the hidden_size indicates the number of hidden units of the single hidden layer, while the num_classes indicates how many classes are there for the training data. It also provide three other inputs, which are listed below

[60]:
```
################################################################################
# This cell train the network with default vanilla SGD optimizer  #
# Other parameters of TwoLayerNet are listed here:                #
# - weight_scale: Scalar giving the standard deviation for random #
#   initialization of the weights. Default 1e-3                   #
# - reg: Scalar giving L2 regularization strength. Default 0.0    #
# - params: The parameters of the net, if the input is none,      #
#   the nets will initialize the parameters using gaussian        #
#   distribution. Defalut None                                    #
################################################################################
input_size = 784
hidden_size = 25
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = Solver(model, data, print_every=100)
# For more parameters information of Solver, please check README
# or python file solver.py.
solver.train()
```

```
(Iteration 1 / 2750) loss: 2.302404
(Epoch 0 / 10) train acc: 0.132109; val_acc: 0.131600
(Iteration 101 / 2750) loss: 0.542671
```

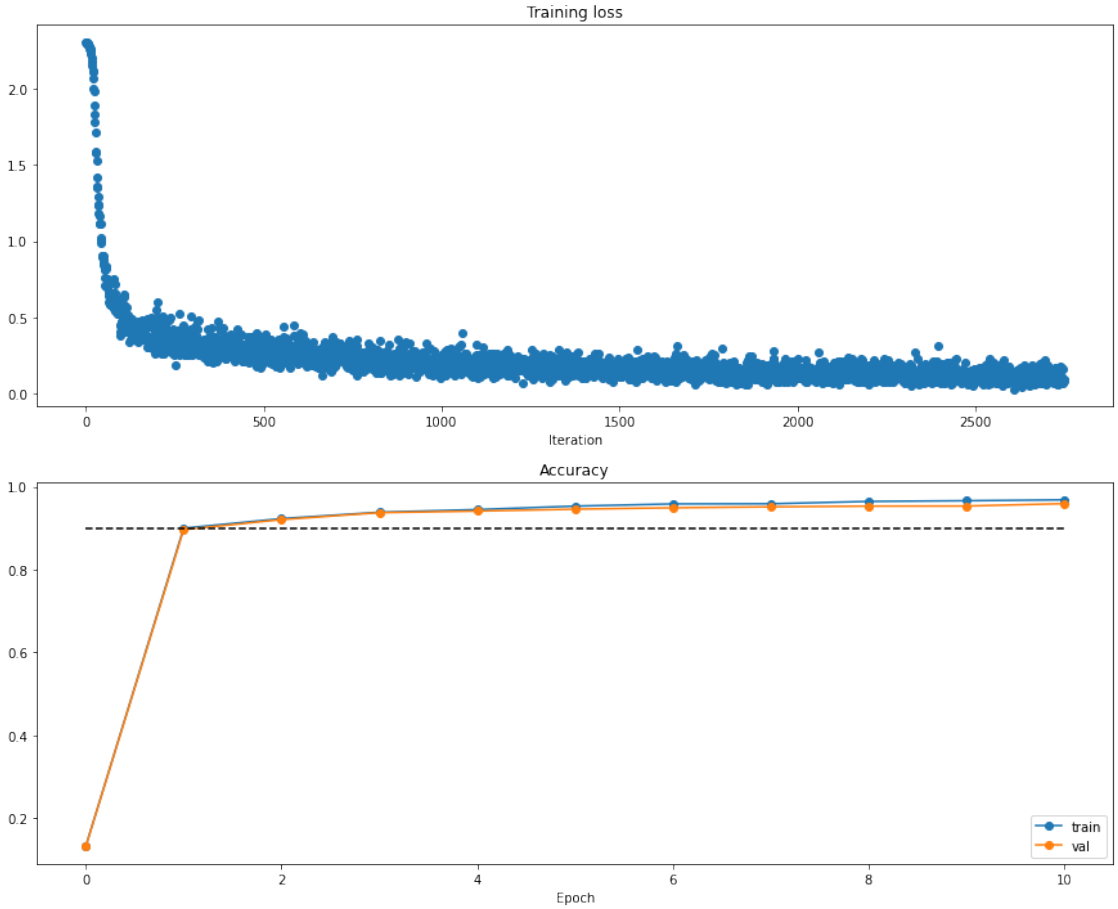
```
(Iteration 201 / 2750) loss: 0.384264
(Epoch 1 / 10) train acc: 0.899818; val_acc: 0.896400
(Iteration 301 / 2750) loss: 0.397509
(Iteration 401 / 2750) loss: 0.327322
(Iteration 501 / 2750) loss: 0.255939
(Epoch 2 / 10) train acc: 0.923073; val_acc: 0.920600
(Iteration 601 / 2750) loss: 0.351975
(Iteration 701 / 2750) loss: 0.194160
(Iteration 801 / 2750) loss: 0.254867
(Epoch 3 / 10) train acc: 0.938418; val_acc: 0.937400
(Iteration 901 / 2750) loss: 0.252108
(Iteration 1001 / 2750) loss: 0.165598
(Epoch 4 / 10) train acc: 0.944855; val_acc: 0.941200
(Iteration 1101 / 2750) loss: 0.224771
(Iteration 1201 / 2750) loss: 0.156349
(Iteration 1301 / 2750) loss: 0.140684
(Epoch 5 / 10) train acc: 0.953218; val_acc: 0.945800
(Iteration 1401 / 2750) loss: 0.213323
(Iteration 1501 / 2750) loss: 0.166477
(Iteration 1601 / 2750) loss: 0.105702
(Epoch 6 / 10) train acc: 0.958491; val_acc: 0.949200
(Iteration 1701 / 2750) loss: 0.107208
(Iteration 1801 / 2750) loss: 0.147283
(Iteration 1901 / 2750) loss: 0.192860
(Epoch 7 / 10) train acc: 0.958855; val_acc: 0.951400
(Iteration 2001 / 2750) loss: 0.175677
(Iteration 2101 / 2750) loss: 0.206384
(Epoch 8 / 10) train acc: 0.964491; val_acc: 0.953200
(Iteration 2201 / 2750) loss: 0.084298
(Iteration 2301 / 2750) loss: 0.112378
(Iteration 2401 / 2750) loss: 0.096437
(Epoch 9 / 10) train acc: 0.966218; val_acc: 0.953400
(Iteration 2501 / 2750) loss: 0.074729
(Iteration 2601 / 2750) loss: 0.078579
(Iteration 2701 / 2750) loss: 0.152481
(Epoch 10 / 10) train acc: 0.968291; val_acc: 0.959200
```

[61]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
```

```
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.9] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



[62]:
```
y_test_pred = np.argmax(solver.model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.9497
```

We could see that, with vanilla SGD and a simple two layer network of 25 hidden units, the model could be trained to achieve more than 90% accuracy on both training and validation set, and achieve 95% percent of accuracy on the testing set! Moreover, we could see that there is just a tiny gap of performance between training set and validation set (even between testing set)

1.2.2 L_2 regularization

In this part we will add the L_2 regularization and see how much we could improve the performance of the same network structure

[6]:
```
input_size = 784
hidden_size = 25
num_classes = 10
# We apply a L2 regularization with strength 1e-3 here
model = TwoLayerNet(input_size, hidden_size, num_classes, reg=1e-3)
solver = Solver(model, data, print_every=100)
solver.train()
```

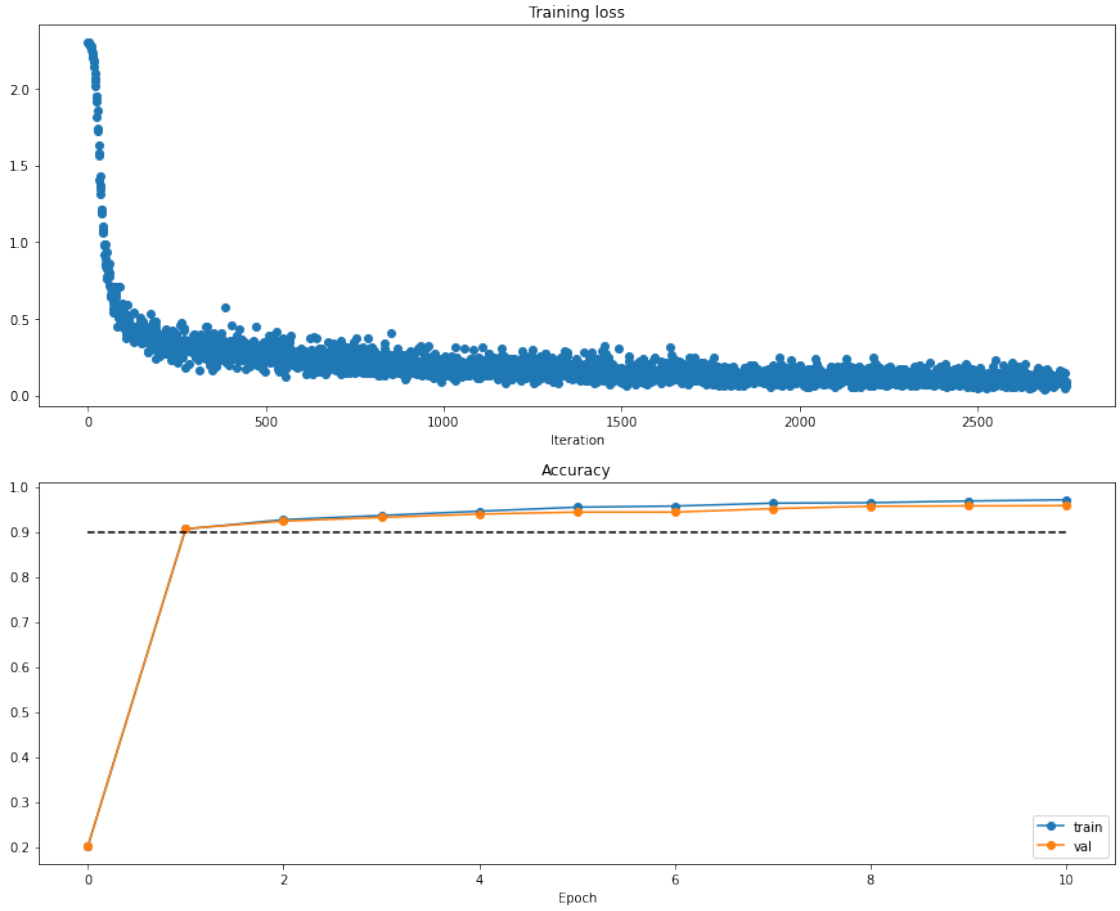
```
(Iteration 1 / 2750) loss: 2.302585
(Epoch 0 / 10) train acc: 0.201255; val_acc: 0.201400
(Iteration 101 / 2750) loss: 0.577068
(Iteration 201 / 2750) loss: 0.254298
(Epoch 1 / 10) train acc: 0.906018; val_acc: 0.906800
(Iteration 301 / 2750) loss: 0.283188
(Iteration 401 / 2750) loss: 0.353007
(Iteration 501 / 2750) loss: 0.293475
(Epoch 2 / 10) train acc: 0.926782; val_acc: 0.923600
(Iteration 601 / 2750) loss: 0.227676
(Iteration 701 / 2750) loss: 0.311082
(Iteration 801 / 2750) loss: 0.151273
(Epoch 3 / 10) train acc: 0.936127; val_acc: 0.931800
(Iteration 901 / 2750) loss: 0.139934
(Iteration 1001 / 2750) loss: 0.198695
(Epoch 4 / 10) train acc: 0.945564; val_acc: 0.939200
(Iteration 1101 / 2750) loss: 0.214539
(Iteration 1201 / 2750) loss: 0.214834
(Iteration 1301 / 2750) loss: 0.163541
(Epoch 5 / 10) train acc: 0.954455; val_acc: 0.943600
(Iteration 1401 / 2750) loss: 0.110215
(Iteration 1501 / 2750) loss: 0.145728
(Iteration 1601 / 2750) loss: 0.103231
(Epoch 6 / 10) train acc: 0.957073; val_acc: 0.943600
(Iteration 1701 / 2750) loss: 0.149281
(Iteration 1801 / 2750) loss: 0.131112
(Iteration 1901 / 2750) loss: 0.079353
(Epoch 7 / 10) train acc: 0.963400; val_acc: 0.951200
(Iteration 2001 / 2750) loss: 0.198145
(Iteration 2101 / 2750) loss: 0.136425
(Epoch 8 / 10) train acc: 0.964418; val_acc: 0.956600
(Iteration 2201 / 2750) loss: 0.147788
(Iteration 2301 / 2750) loss: 0.095600
(Iteration 2401 / 2750) loss: 0.125025
(Epoch 9 / 10) train acc: 0.968255; val_acc: 0.957400
(Iteration 2501 / 2750) loss: 0.137654
```

```
(Iteration 2601 / 2750) loss: 0.167207
(Iteration 2701 / 2750) loss: 0.093695
(Epoch 10 / 10) train acc: 0.970764; val_acc: 0.958000
```

[7]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.9] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



[8]:
```
y_test_pred = np.argmax(solver.model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.9547
```

We could see that there is no significant difference after we applying the L_2 regularization. Maybe it is because the whole dataset is just too simple.

1.2.3 Learning rate decay

In this part we deploy learning rate decay to shrink at the end of every epoch.

[9]:
```
input_size = 784
hidden_size = 25
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
# We apply learning rate decay here
solver = Solver(model, data, print_every=100, lr_decay=0.99)
solver.train()
```

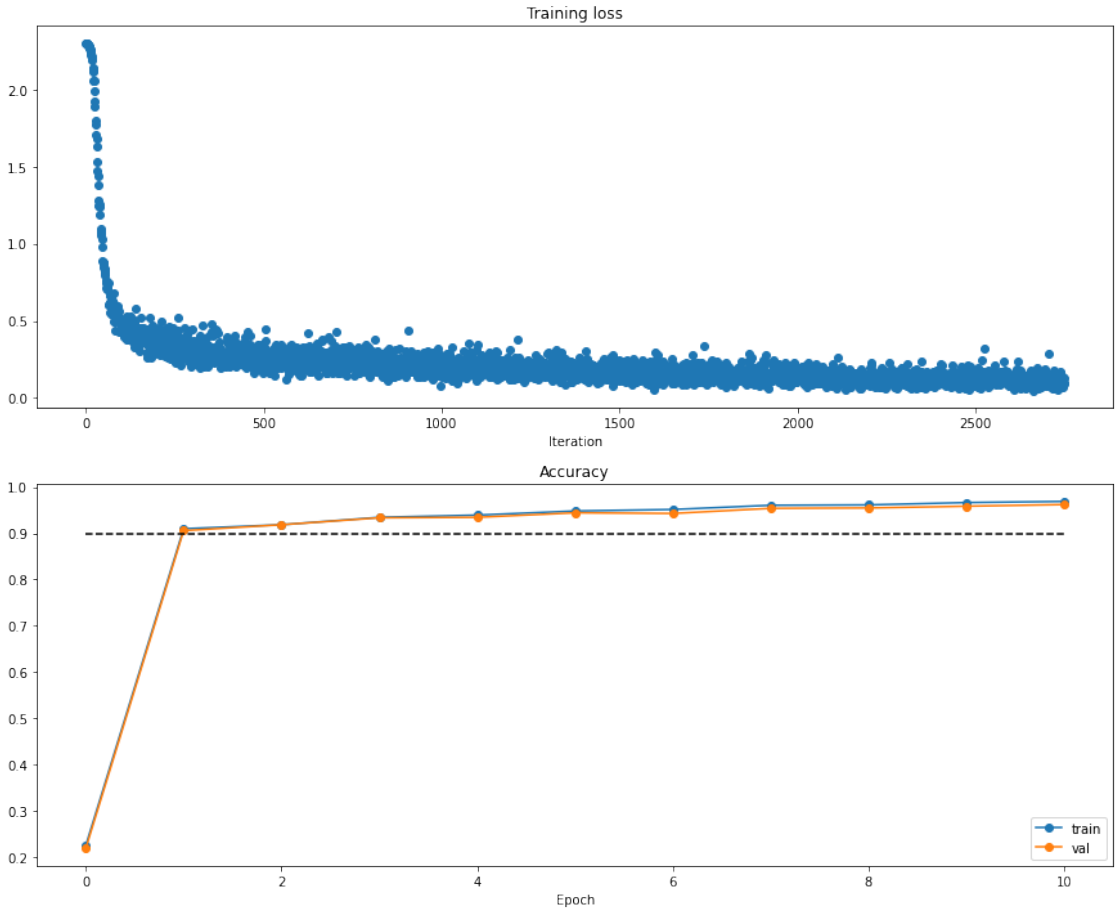
```
(Iteration 1 / 2750) loss: 2.301618
(Epoch 0 / 10) train acc: 0.225255; val_acc: 0.219200
(Iteration 101 / 2750) loss: 0.432779
(Iteration 201 / 2750) loss: 0.382305
(Epoch 1 / 10) train acc: 0.909673; val_acc: 0.905600
(Iteration 301 / 2750) loss: 0.448011
(Iteration 401 / 2750) loss: 0.353138
(Iteration 501 / 2750) loss: 0.194576
(Epoch 2 / 10) train acc: 0.918400; val_acc: 0.918400
(Iteration 601 / 2750) loss: 0.156914
(Iteration 701 / 2750) loss: 0.312384
(Iteration 801 / 2750) loss: 0.193965
(Epoch 3 / 10) train acc: 0.934400; val_acc: 0.933400
(Iteration 901 / 2750) loss: 0.304708
(Iteration 1001 / 2750) loss: 0.198908
(Epoch 4 / 10) train acc: 0.939182; val_acc: 0.934400
(Iteration 1101 / 2750) loss: 0.242500
(Iteration 1201 / 2750) loss: 0.172572
(Iteration 1301 / 2750) loss: 0.140060
(Epoch 5 / 10) train acc: 0.948073; val_acc: 0.944200
(Iteration 1401 / 2750) loss: 0.121991
(Iteration 1501 / 2750) loss: 0.177355
(Iteration 1601 / 2750) loss: 0.162331
(Epoch 6 / 10) train acc: 0.951382; val_acc: 0.942800
(Iteration 1701 / 2750) loss: 0.149697
(Iteration 1801 / 2750) loss: 0.151470
(Iteration 1901 / 2750) loss: 0.117613
(Epoch 7 / 10) train acc: 0.960455; val_acc: 0.953800
```

```
(Iteration 2001 / 2750) loss: 0.172190
(Iteration 2101 / 2750) loss: 0.171422
(Epoch 8 / 10) train acc: 0.961273; val_acc: 0.954600
(Iteration 2201 / 2750) loss: 0.127354
(Iteration 2301 / 2750) loss: 0.118901
(Iteration 2401 / 2750) loss: 0.219375
(Epoch 9 / 10) train acc: 0.966291; val_acc: 0.958400
(Iteration 2501 / 2750) loss: 0.115053
(Iteration 2601 / 2750) loss: 0.097593
(Iteration 2701 / 2750) loss: 0.111615
(Epoch 10 / 10) train acc: 0.968782; val_acc: 0.962000
```

[10]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.9] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



[11]:
```
y_test_pred = np.argmax(solver.model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.9532
```

We could see that compare to the vanilla one, there is also no significant boosting of the performance

1.3 Part 2: Hyperparameters Searching

In this section, I will show you some simple unpacked codes of hyperparameters searching. In my github page you could find a packed version of this code, doing exactly the same things but you could do it all in command line.

[33]:
```
input_size = 784
num_classes = 10

# Choice of hyperparameters
lr_rate = [1e-3, 1e-4]
hidden_units = [100, 200, 300]
regs = [1e-1, 1e-2, 1e-3]
```

```
best_acc = 0.0
best_model = None
best_params = []

for lr in lr_rate:
    for units in hidden_units:
        for reg in regs:
            model = TwoLayerNet(input_size,
                                units,
                                num_classes,
                                reg=reg)
            solver = Solver(model,
                            data,
                            optim_config={
                                'learning_rate': lr
                            },
                            verbose = False,
                            save_model = False,
                            num_epochs = 20)
            solver.train()
            print("================================================")
            print("learning rate = %f, hidden units = %d\nregularization␣
 ↪strength = %f, accuracy on val = %f"
                  % (lr, units, reg, solver.best_val_acc))
            print("================================================")
            if solver.best_val_acc > best_acc:
                best_model = solver.model
                best_acc = solver.best_val_acc
                best_params = [lr, units, reg]
```

```
================================================
learning rate = 0.001000, hidden units = 100
regularization strength = 0.100000, accuracy on val = 0.979800
================================================
================================================
learning rate = 0.001000, hidden units = 100
regularization strength = 0.010000, accuracy on val = 0.983000
================================================
================================================
learning rate = 0.001000, hidden units = 100
regularization strength = 0.001000, accuracy on val = 0.983800
================================================
================================================
learning rate = 0.001000, hidden units = 200
regularization strength = 0.100000, accuracy on val = 0.980600
================================================
================================================
learning rate = 0.001000, hidden units = 200
```

```
regularization strength = 0.010000, accuracy on val = 0.986000
================================================
================================================
learning rate = 0.001000, hidden units = 200
regularization strength = 0.001000, accuracy on val = 0.984200
================================================
================================================
learning rate = 0.001000, hidden units = 300
regularization strength = 0.100000, accuracy on val = 0.983200
================================================
================================================
learning rate = 0.001000, hidden units = 300
regularization strength = 0.010000, accuracy on val = 0.985200
================================================
================================================
learning rate = 0.001000, hidden units = 300
regularization strength = 0.001000, accuracy on val = 0.985800
================================================
================================================
learning rate = 0.000100, hidden units = 100
regularization strength = 0.100000, accuracy on val = 0.932600
================================================
================================================
learning rate = 0.000100, hidden units = 100
regularization strength = 0.010000, accuracy on val = 0.932400
================================================
================================================
learning rate = 0.000100, hidden units = 100
regularization strength = 0.001000, accuracy on val = 0.932000
================================================
================================================
learning rate = 0.000100, hidden units = 200
regularization strength = 0.100000, accuracy on val = 0.935400
================================================
================================================
learning rate = 0.000100, hidden units = 200
regularization strength = 0.010000, accuracy on val = 0.935600
================================================
================================================
learning rate = 0.000100, hidden units = 200
regularization strength = 0.001000, accuracy on val = 0.938400
================================================
================================================
learning rate = 0.000100, hidden units = 300
regularization strength = 0.100000, accuracy on val = 0.938200
================================================
================================================
learning rate = 0.000100, hidden units = 300
```

```
regularization strength = 0.010000, accuracy on val = 0.939200
==================================================
==================================================
learning rate = 0.000100, hidden units = 300
regularization strength = 0.001000, accuracy on val = 0.937600
==================================================
```

[56]:
```
best_model.reg = 0.0
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
print("Best parameters:")
print("Learning rate:", best_params[0])
print("Hidden units:", best_params[1])
print("Regularization strength:", best_params[2])
```

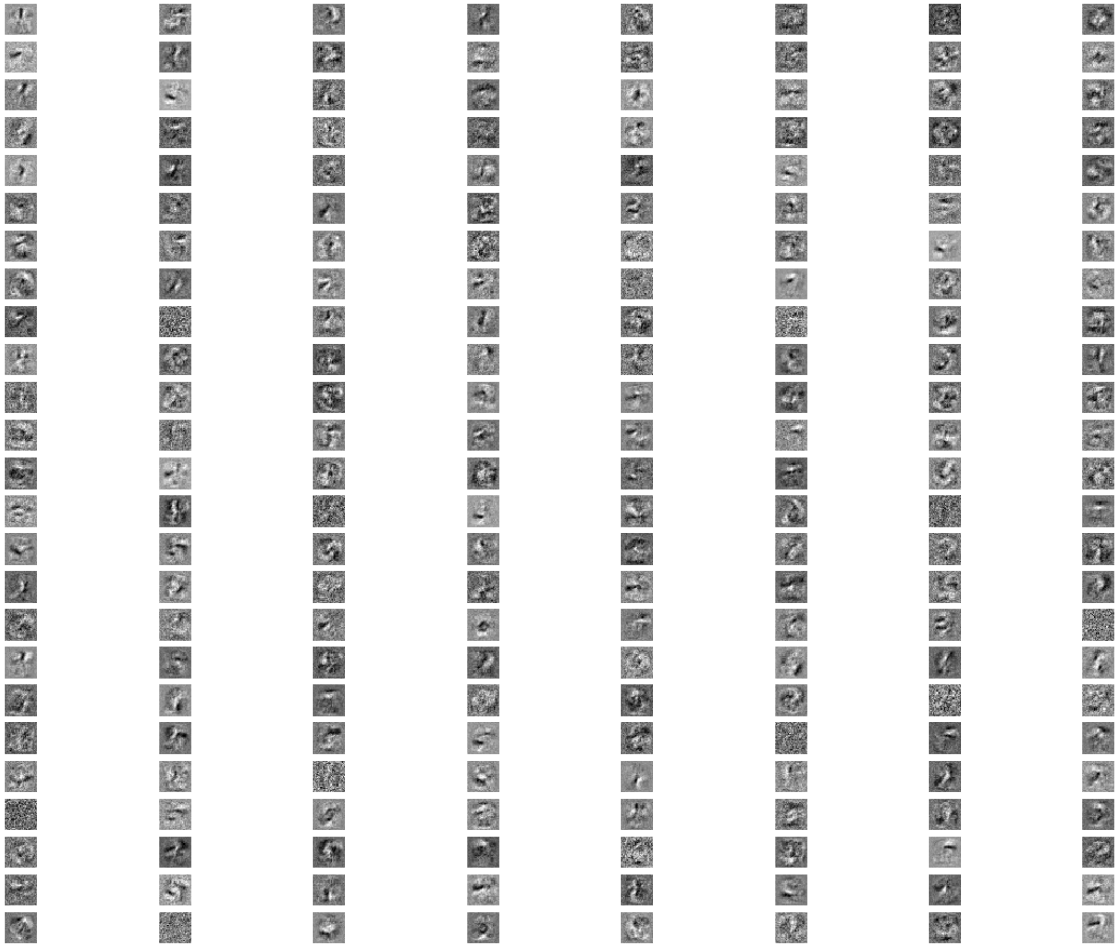
```
Test set accuracy:  0.976
Best parameters:
Learning rate: 0.001
Hidden units: 200
Regularization strength: 0.01
```

We could see that the with 200 hidden units, 1e-3 learning rate and 0.01 regularization strength, we get 97.6% accuracy on the test set. We could visualize the parameter with the following codes.

[54]:
```
W1 = best_model.params['W1'].reshape(28, 28, -1).transpose(2, 0, 1)
plt.rcParams['image.cmap'] = 'gray'
num_row = (W1.shape[0] + 7) // 8
count = 0

for i in range(num_row):
    for j in range(8):
        plt.subplot(num_row, 8, count + 1)
        img = W1[count, :, :]
        img =  255.0 * (img - np.min(img)) / (np.max(img) - np.min(img))
        plt.imshow(img.astype('uint8'))
        plt.gca().axis('off')
        count = count + 1

plt.show()
```



And we could save our parameters into the path we want using the following codes

[55]:
```
np.save("best_model.npy", best_model.params)
```

1.4 Part 3: Testing

In this part, we show that we could load the model we save, and test it on the testing set and output the accuracy

[59]:
```
params = np.load("best_model.npy", allow_pickle=True).item()
model = TwoLayerNet(784, params['W1'].shape[1], 10, params=params)
y_test_pred = np.argmax(model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.976
```

We could see that the test accuracy is exactly the same as the above accuracy we get. I've upload the above parameters on my github page and also my google drive, feel free to download it!