This is the 2-layer neural network notebook for ECE C147/C247 Homework #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the notebook entirely when completed.

The goal of this notebook is to give you experience with training a two layer neural network.

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

%matplotlib inline
%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))))
```

Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass. Make sure to read the description of TwoLayerNet class in neural_net.py file , understand the architecture and initializations

In [ ]:
```
from nndl.neural_net import TwoLayerNet
```

In [ ]:
```
# 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()
```

Compute forward pass scores

In [ ]:
```
## Implement the forward pass of the neural network.
## See the loss() method in TwoLayerNet class for the same

# 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)))

# mine
# A D G
# J M B
# E H K
# N C F
# I L O

# correct
# A B C
# D E F
# G H I
# J K L
# M N O
```

```
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:
3.381231233889892e-08
```

Forward pass loss

In [ ]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

```
Loss: 1.071696123862817
Difference between your loss and correct loss:
0.0
```

Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

In [ ]:
```
from utils.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pa
# 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
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_r
```

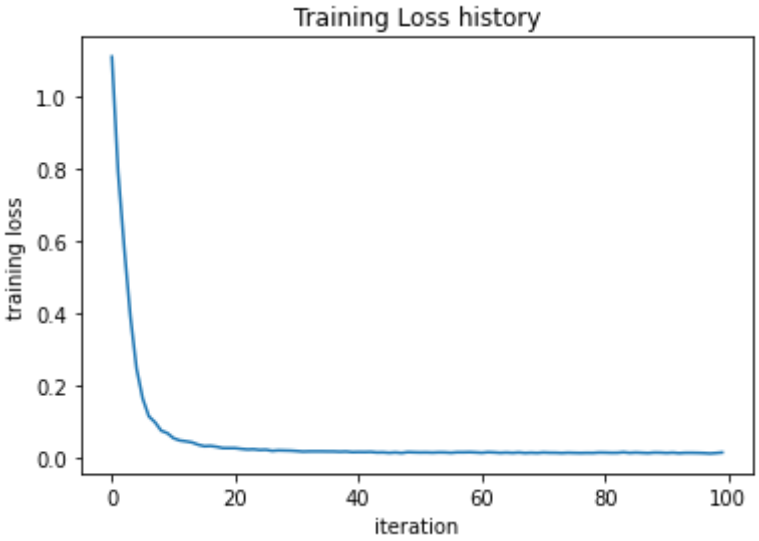
```
b2 max relative error: 1.248270530283678e-09
W2 max relative error: 2.9632227682005116e-10
b1 max relative error: 3.172680092703762e-09
W1 max relative error: 1.2832823337649917e-09
```

Training the network

Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax and SVM.

In [ ]:
```
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()
```

```
Final training loss:  0.014497864587765886
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [ ]:
```
from utils.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.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '/Users/mcapetz/Downloads/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,)
```

Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

In [ ]:
```
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-4, learning_rate_decay=0.95,
            reg=0.25, 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 2.302769713193006
iteration 100 / 1000: loss 2.3024454719405805
iteration 200 / 1000: loss 2.2973223064347863
iteration 300 / 1000: loss 2.2714527896770402
iteration 400 / 1000: loss 2.1676144295711666
iteration 500 / 1000: loss 2.1187039249761
iteration 600 / 1000: loss 2.0425279262316556
iteration 700 / 1000: loss 2.0410322587782623
iteration 800 / 1000: loss 1.9946566892338784
iteration 900 / 1000: loss 1.896382745210702
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

(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.

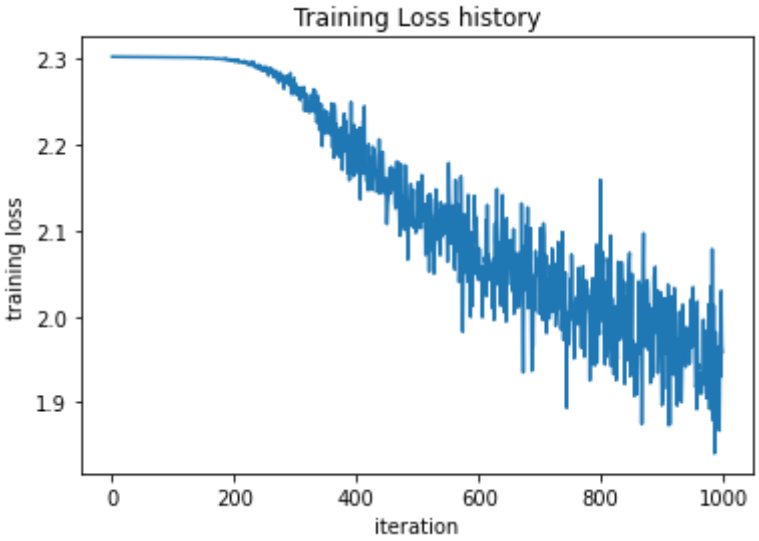
(2) How should you fix the problems you identified in (1)?

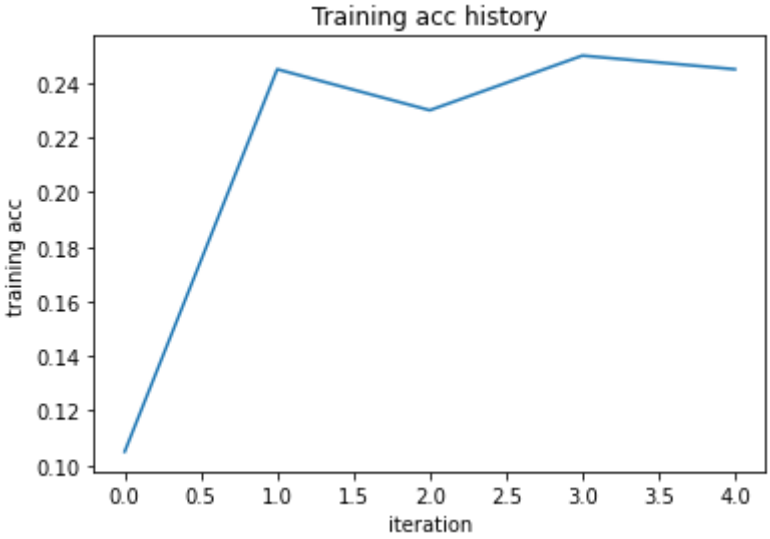
In [ ]:
```
stats['train_acc_history']
```

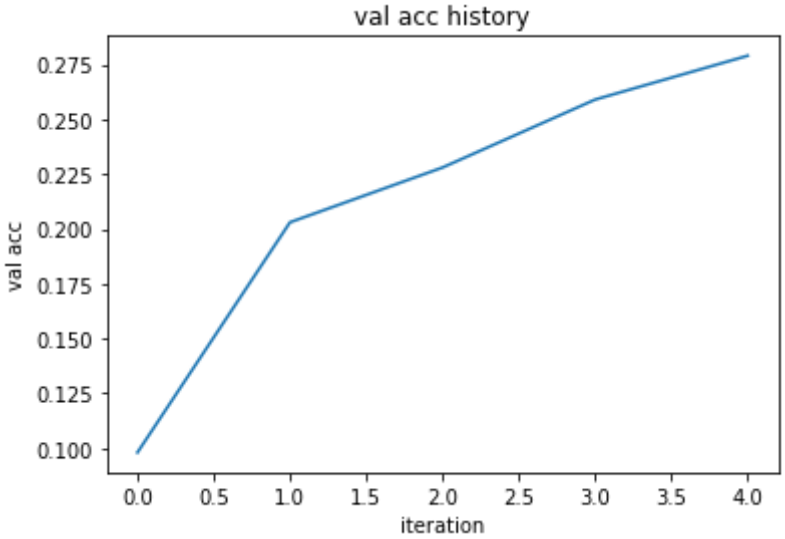
Out[ ]:
```
[0.105, 0.245, 0.23, 0.25, 0.245]
```

In [ ]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies

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()


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

# plot the loss history
plt.plot(stats['val_acc_history'])
plt.xlabel('iteration')
plt.ylabel('val acc')
plt.title('val acc history')
plt.show()


# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Final training loss:  1.9585869601827879
```







Answers:

(1) The hyperparameters including learning rate, iteration count, batch size, etc were not tuned. In particular, the iteration count was not as high as it could have been.

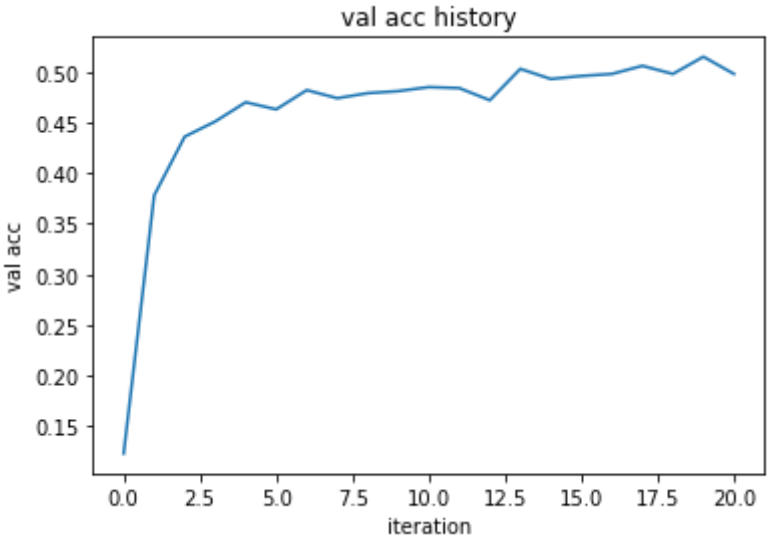
(2) I fixed these problems by experimenting with different numbers and plotting the different validation accuracies to see at what point they peaked. This led me to increase the number of iterations and also alter the learning rate and regularization rate.

Optimize the neural network

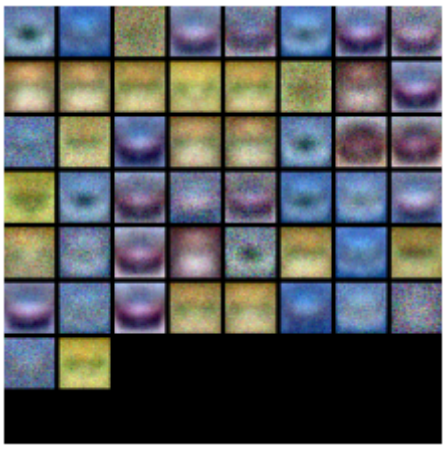
Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

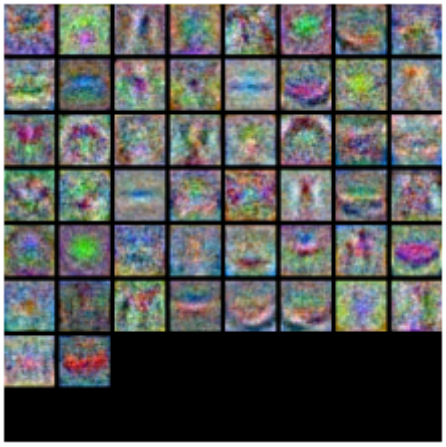
In [ ]:
```
best_net = TwoLayerNet(input_size, hidden_size, num_classes) # store the best m

# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 50% 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 - 28%)) / %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)!
# ================================================================ #
# Train the network
stats = best_net.train(X_train, y_train, X_val, y_val,
            num_iters=5000, batch_size=200,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=0.5, verbose=True)

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

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)

# plot the loss history
plt.plot(stats['val_acc_history'])
plt.xlabel('iteration')
plt.ylabel('val acc')
plt.title('val acc history')
plt.show()
```

```
iteration 0 / 5000: loss 2.3029597225435428
iteration 100 / 5000: loss 1.9325933178332602
iteration 200 / 5000: loss 1.7716759550027896
iteration 300 / 5000: loss 1.7248081531071433
iteration 400 / 5000: loss 1.8443332386092726
iteration 500 / 5000: loss 1.5302909563702052
iteration 600 / 5000: loss 1.6526188579465462
iteration 700 / 5000: loss 1.5906508141462397
iteration 800 / 5000: loss 1.4855353596856218
iteration 900 / 5000: loss 1.5159556839929458
iteration 1000 / 5000: loss 1.488270908981075
iteration 1100 / 5000: loss 1.4475743724154462
iteration 1200 / 5000: loss 1.4828636891972566
iteration 1300 / 5000: loss 1.4527629724343858
iteration 1400 / 5000: loss 1.4842048768754978
iteration 1500 / 5000: loss 1.5642384638909517
iteration 1600 / 5000: loss 1.5077882504893556
iteration 1700 / 5000: loss 1.5485629034664101
iteration 1800 / 5000: loss 1.4868146985257522
iteration 1900 / 5000: loss 1.4441929644630527
iteration 2000 / 5000: loss 1.5144998528901483
iteration 2100 / 5000: loss 1.393662849428555
iteration 2200 / 5000: loss 1.5304174989897708
iteration 2300 / 5000: loss 1.445368445922107
iteration 2400 / 5000: loss 1.455080286765323
iteration 2500 / 5000: loss 1.4549381905657
iteration 2600 / 5000: loss 1.2709510874375622
iteration 2700 / 5000: loss 1.5267038798950487
iteration 2800 / 5000: loss 1.4386154710730088
iteration 2900 / 5000: loss 1.4934165464617921
iteration 3000 / 5000: loss 1.3580106333238915
iteration 3100 / 5000: loss 1.3466971155558127
iteration 3200 / 5000: loss 1.3696032551646022
iteration 3300 / 5000: loss 1.3616061593985698
iteration 3400 / 5000: loss 1.4596148696103697
iteration 3500 / 5000: loss 1.3419087222617698
iteration 3600 / 5000: loss 1.5033050723689223
iteration 3700 / 5000: loss 1.3034197628298878
iteration 3800 / 5000: loss 1.587226669889954
iteration 3900 / 5000: loss 1.3827370758432151
iteration 4000 / 5000: loss 1.5225687814985926
iteration 4100 / 5000: loss 1.4189289335169293
iteration 4200 / 5000: loss 1.353386840778016
iteration 4300 / 5000: loss 1.4960344832766483
iteration 4400 / 5000: loss 1.3407818549293629
iteration 4500 / 5000: loss 1.3933265081570054
iteration 4600 / 5000: loss 1.2662590757250827
iteration 4700 / 5000: loss 1.507514944399104
iteration 4800 / 5000: loss 1.4202320018537
iteration 4900 / 5000: loss 1.3732155789389926
Validation accuracy:  0.507
Validation accuracy:  0.507
```



In [ ]:
```
from utils.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)
```





Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

In [ ]:

Answer:

(1) The best net I arrived at is more colorful and has more contrast. There are more complex shapes and colors compared to the suboptimal net. The shapes of colors in the best net I arrived at have more distinguishable shapes and features.

Evaluate on test set

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

```
Test accuracy:  0.515
```

In [ ]: