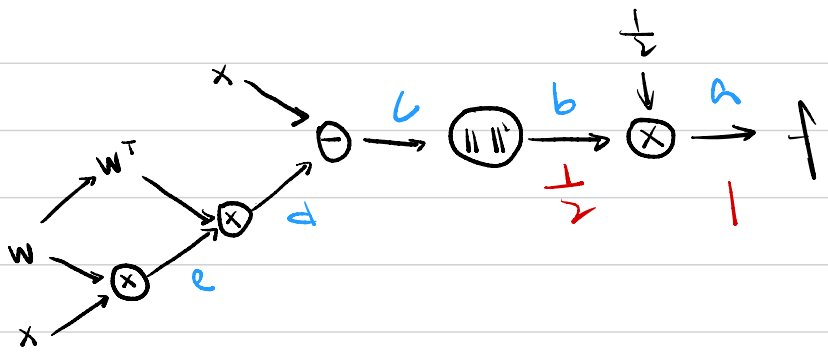
HW3 Yuchen Wang

1.

(a) Minimizing this L by finding W is to minimize the error between the predicted output and the actual output. So, the model will best capture the patterns from the input data x. Therefore, this minimization will ought to preserve information about x.

(b)



(c) Suppose there are two paths $W \to a \to L$ and $W \to b \to L$, when we calculate $\nabla_W L$, we can expree this by the summation of the derivative of two paths of:

$$\nabla_W L = \frac{\partial a}{\partial W} \cdot \frac{\partial L}{\partial a} + \frac{\partial b}{\partial W} \cdot \frac{\partial L}{\partial b}$$

(d) Suppose we have $\frac{\partial L}{\partial a} = 1$

$a = \frac{1}{2} b$

$$\frac{\partial L}{\partial b} = \frac{\partial a}{\partial b} \cdot \frac{\partial L}{\partial a} = \frac{1}{2} \cdot 1 = \frac{1}{2}$$

$b = \|v\|^2$

$$\frac{\partial L}{\partial v} = \frac{\partial b}{\partial v} \cdot \frac{\partial L}{\partial b} = 2v \cdot \frac{1}{2} = v$$ (In Matrixwok βv)

$v = d - x$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial d} \cdot \frac{\partial L}{\partial v} = 1 \cdot v = v$$

$d = W^T \cdot e$

$$\frac{\partial L}{\partial e} = \frac{\partial d}{\partial e} \cdot \frac{\partial L}{\partial d} = W^T \cdot v$$

$$\frac{\partial L}{\partial W^T} = \frac{\partial d}{\partial W^T} \cdot \frac{\partial L}{\partial d} = e \cdot v$$

$e = Wx$

$$\frac{\partial L}{\partial W} = \frac{\partial e}{\partial W} \cdot \frac{\partial L}{\partial e} = x \cdot W^T \cdot v$$

When $L = W^T W x - x$, we will have

$$\frac{\partial L}{\partial W^T} = W x (W^T W x - x)$$

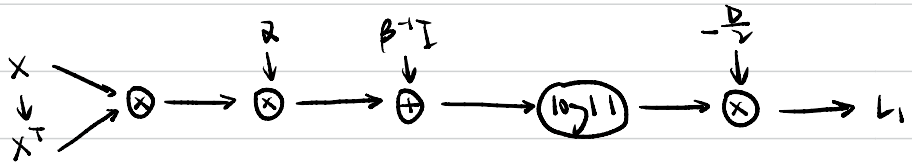
$$\frac{\partial L}{\partial W} = x W^T (W^T W x - x)$$

$$\nabla_W L = W (W^T W x - x) x^T + W x (W^T W x - x)^T$$

$m \times n \quad n \quad 1 \times n \quad m \times n \quad n \quad 1 \times n$

3.

(a)



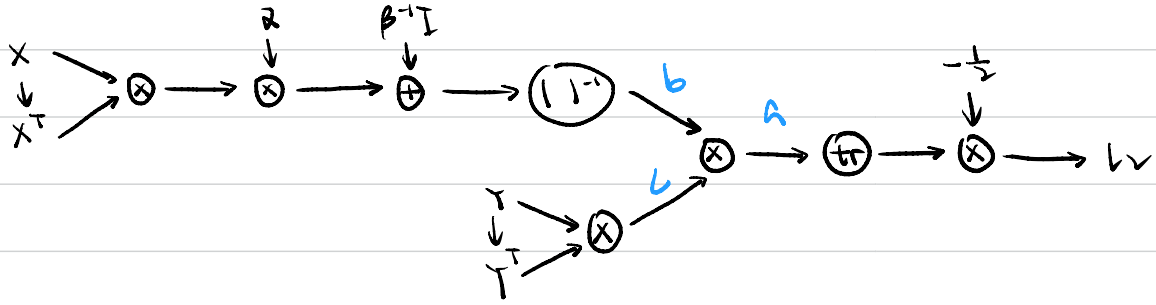
(b) $\frac{\partial L_1}{\partial K} = -\frac{D}{2}(K^{-1})^T$ (In Matrix Book 57))

$\frac{\partial L_1}{\partial X} = \frac{\partial K}{\partial X} \cdot \frac{\partial L_1}{\partial K} = -\frac{D}{2}(K^{-1})^T \cdot 2X$

$\frac{\partial L_1}{\partial X^T} = \frac{\partial K}{\partial X^T} \cdot \frac{\partial L_1}{\partial K} = 2X^T \cdot \left(-\frac{D}{2}(K^{-1})^T\right)$

$\frac{\partial L_1}{\partial X} = -\frac{D}{2}(K^{-1})^T \cdot 2X + 2X^T \cdot \left(-\frac{D}{2}(K^{-1})^T\right) = -2D(K^{-1})^T X$, where $K = 2XX^T + \beta^{-1}I$

(c)



(d) $\frac{\partial L_2}{\partial K^{-1}YY^T} = -\frac{1}{2}I$ (In Matrix Book 65))

$A = K^{-1}YY^T$

$\frac{\partial L_2}{\partial K^{-1}} = \frac{\partial A}{\partial K^{-1}} \cdot \frac{\partial L_2}{\partial A} = YY^T \cdot \left(-\frac{1}{2}I\right) = -\frac{1}{2}YY^T$

$\frac{\partial K^{-1}}{\partial X} = -K^{-1}\frac{\partial K}{\partial X}K^{-1}$ (In Matrix Book 59))

$\frac{\partial K}{\partial X} = 2X$ (From (b))

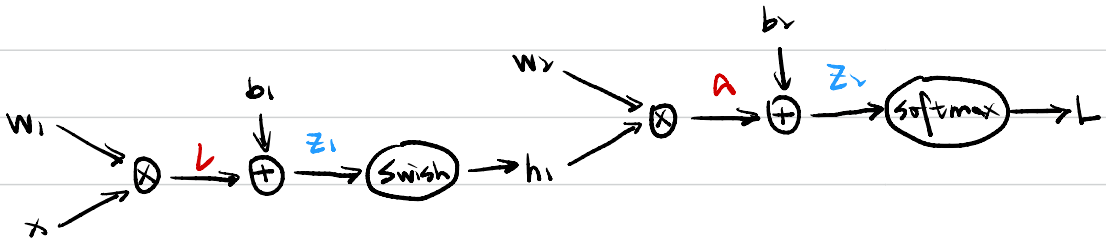
$\frac{\partial L_2}{\partial X} = \frac{\partial K^{-1}}{\partial X} \cdot \frac{\partial L_2}{\partial K^{-1}} = -K^{-1}(2X)K^{-1} \cdot \left(-\frac{1}{2}YY^T\right)$

$= 2K^{-1}YY^TK^{-1}X$, where $K = 2XX^T + \beta^{-1}I$

(e) $\frac{\partial L}{\partial X} = -2D(K^{-1})^T X + 2K^{-1}YY^TK^{-1}X$, where $K = 2XX^T + \beta^{-1}I$

3.

(a)



(b) Since we know $\frac{\partial L}{\partial z_2}$,

$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}$ (based on what we learned in the lecture)

$a = W_2 h_1$

$\frac{\partial L}{\partial W_2} = \frac{\partial a}{\partial W_2} \cdot \frac{\partial L}{\partial a} = \frac{\partial L}{\partial a} \cdot h_1^T = \frac{\partial L}{\partial z_2} \cdot h_1^T$

So, $\nabla_{W_2} L = \frac{\partial L}{\partial z_2} \cdot h_1^T$, $\nabla_{b_2} L = \frac{\partial L}{\partial z_2}$

(c) $\frac{\partial L}{\partial h_1} = \frac{\partial a}{\partial h_1} \cdot \frac{\partial L}{\partial a} = W_2^T \frac{\partial L}{\partial z_2}$

Now, we want to calculate $\frac{\partial L}{\partial z_1} = \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial L}{\partial h_1}$

$h_1 = \text{swish}(z_1) = z_1 \sigma(z_1)$, and swish is an elementwise swish operation

$\frac{\partial h_1}{\partial z_1} = \frac{\partial}{\partial z_1} \text{swish}(z_1)$

product rule

$= z_1 \sigma(z_1)[1 - \sigma(z_1)] + \sigma(z_1)$

$= z_1 \sigma(z_1) - z_1(\sigma(z_1))^2 + \sigma(z_1)$

$= z_1 \sigma(z_1) + \sigma(z_1)(1 - z_1 \sigma(z_1))$

$= \text{swish}(z_1) + \sigma(z_1)(1 - \text{swish}(z_1))$

$= h_1 + \sigma(z_1)(1 - h_1)$

$\frac{\partial h_1}{\partial z_1} = \begin{bmatrix} h_{1,1} \sigma(z_{1,1})(1 - h_{1,1}) & \cdots \\ \vdots & \ddots \\ & h_{1,H} \sigma(z_{1,H})(1 - h_{1,H}) \end{bmatrix} \in \mathbb{R}^{H \times H}$

This is a diagonal matrix where the entries besides the diagonal are all 0's

Therefore, $\frac{\partial L}{\partial z_1} = \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial L}{\partial h_1} = (h_1 + \sigma(z_1) \odot (1 - h_1)) \odot \frac{\partial L}{\partial h_1}$

$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial U} = \frac{\partial L}{\partial z_1} = (h_1 + \sigma(z_1) \odot (1 - h_1)) \odot \left(W_2^T \cdot \frac{\partial L}{\partial z_2}\right)$

$U = W_1 x$

$$\frac{\partial L}{\partial W_1} = \frac{\partial z_1}{\partial W_1} \cdot \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial z_1} \cdot X^T = \left[\left(h_1 + \sigma(z_1) \odot (1 - h_1) \right) \odot \left(W_2^T \cdot \frac{\partial L}{\partial z_2} \right) \right] \cdot X^T$$

So, $\nabla_{W_1} L = \left[\left(h_1 + \sigma(z_1) \odot (1 - h_1) \right) \odot \left(W_2^T \cdot \frac{\partial L}{\partial z_2} \right) \right] \cdot X^T$ and

$\nabla_{b_1} L = \left(h_1 + \sigma(z_1) \odot (1 - h_1) \right) \odot \left(W_2^T \cdot \frac{\partial L}{\partial z_2} \right)$

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 [110]:

```
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))))
```

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

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 [111]:

```
from nndl.neural_net import TwoLayerNet
```

In [112]:

```
# 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 [113]:

```
## 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)))
```

```
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:
3.3812311957259755e-08
```

Forward pass loss

In [114]:

```
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 [115]:

```
from utils.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=Fals
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, g
```

```
W2 max relative error: 2.9632233460136427e-10
b2 max relative error: 1.8392017135950213e-10
W1 max relative error: 1.28328951808708e-09
b1 max relative error: 3.172680285697327e-09
```

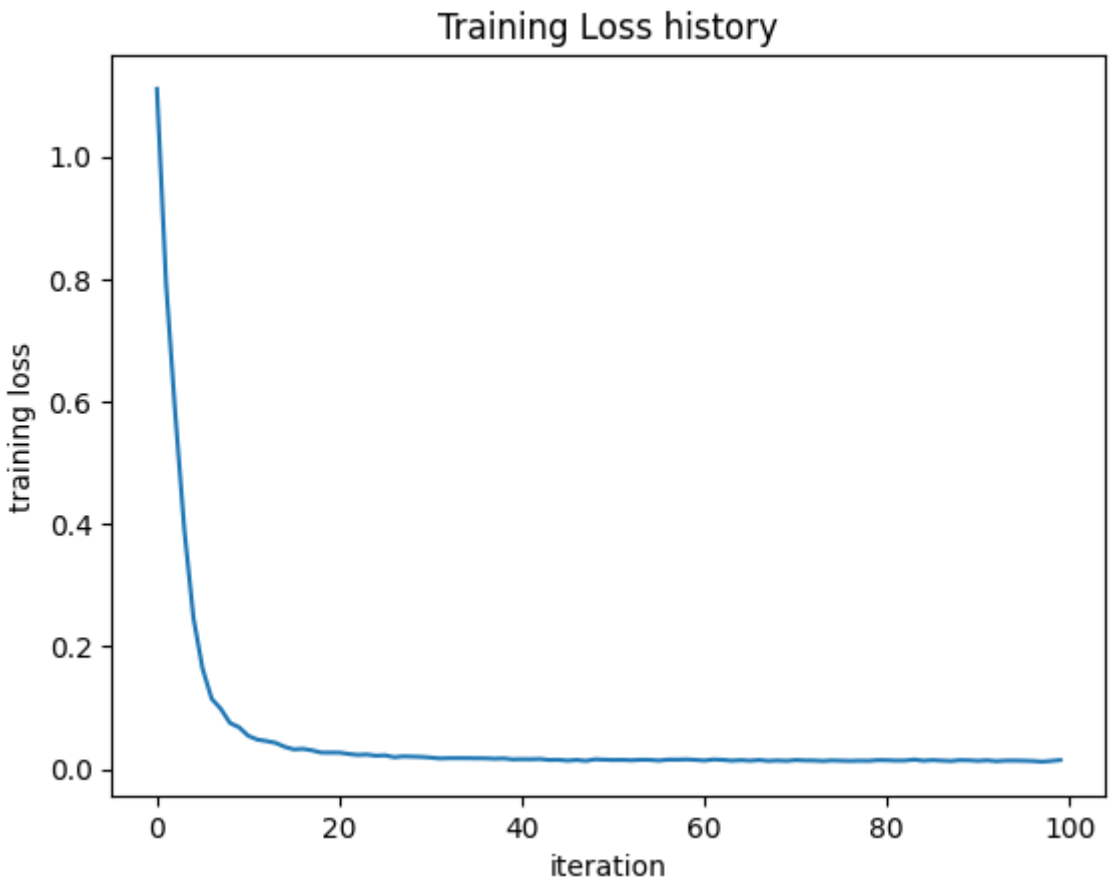
Training the network

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

In [116]:

```
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:  0.014498902952971729
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [117]:

```
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/wangyuchen/desktop/COM SCI 247/HW/HW2/hw2_Questions/code/c
    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 [118]:

```
from nndl.neural_net import TwoLayerNet
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 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.2518259043164135
iteration 400 / 1000: loss 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897743
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.9901688623083942
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
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 [119]:

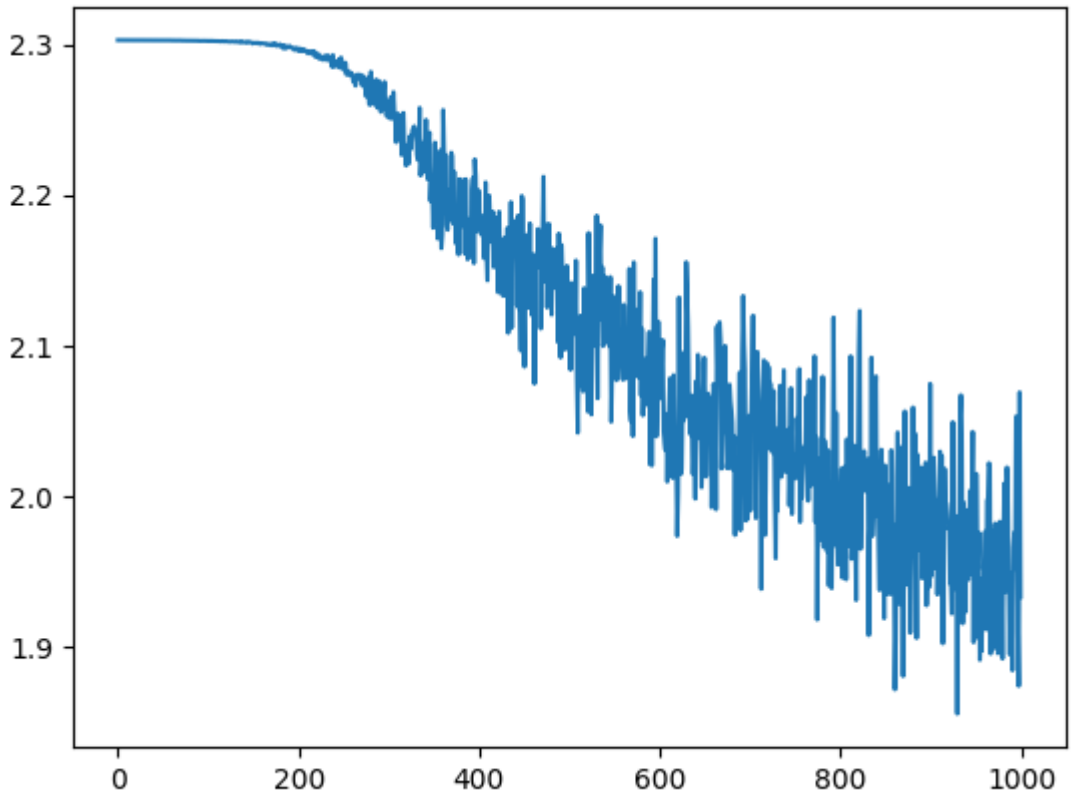
```
stats['train_acc_history']
```

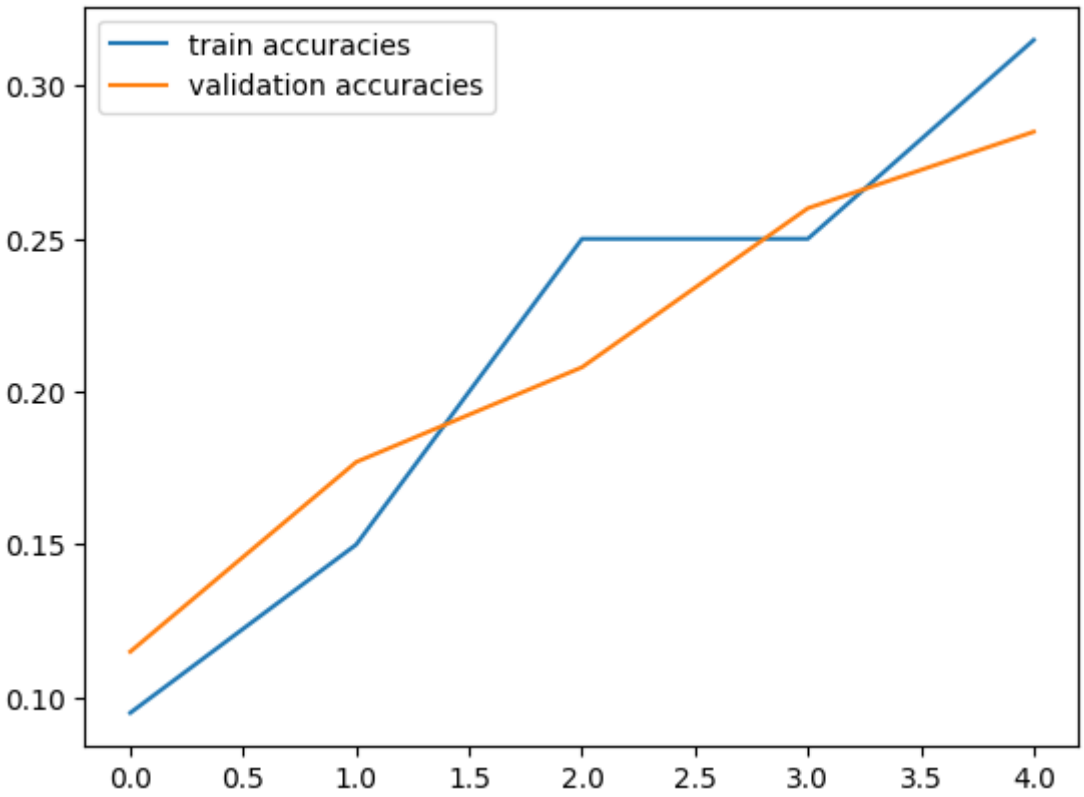
Out[119]:

```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

In [120]:

```
# ================================================================ #
# 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

#plot the loss function
plt.plot(stats['loss_history'])
plt.show()

#plot the train / validation accuracies
#plot the train accuracies
plt.plot(stats['train_acc_history'], label = "train accuracies")
#plot the validation accuracies
plt.plot(stats['val_acc_history'], label = "validation accuracies")
plt.legend() #show the labels
plt.show()


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





Answers:

(1) First of all, it seems like the learning rate is not large enough since the loss in the first graph barely changes during the first 200 iterations. Also, the changes of loss is still unpredictable and the loss function does not go flatten. So, the number of iterations is not enough. On the other hand, the patters of train accuracy and validation accuracy do not match.

(2) Based on my answer to the first question, I will try to increase the learning rate and number of iterations.

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 [121]:

```
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 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 by changing the learning rate

input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10

learningrates = [1e-5, 1e-4, 1e-3] #checking different learning rate

currentbestacc = 0

for rate in learningrates:
    currentnet = TwoLayerNet(input_size, hidden_size, num_classes)
    print("Current learning rate: ", rate)
    currentstats = currentnet.train(X_train, y_train, X_val, y_val,
                num_iters=6000, batch_size=200,
                #increase the number of iterations from 1000 to 6000
                learning_rate=rate, learning_rate_decay=0.95,
                reg=0.25, verbose=True)

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

    #when we detect a better accuracy, update it
    if currentval_acc > currentbestacc:
        currentbestacc = currentval_acc
        best_net = currentnet #update the best net

#As we can see that when we have learning rate = 1e-3 and 6000 iterations,
#the validation accuracy is 0.525, which is higher than 0.5

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

```
Current learning rate:  1e-05
iteration 0 / 6000: loss 2.3027667167979295
iteration 100 / 6000: loss 2.3027299318867556
iteration 200 / 6000: loss 2.3027045062357443
iteration 300 / 6000: loss 2.3027105833127006
iteration 400 / 6000: loss 2.302681593506126
iteration 500 / 6000: loss 2.302660027738356
```

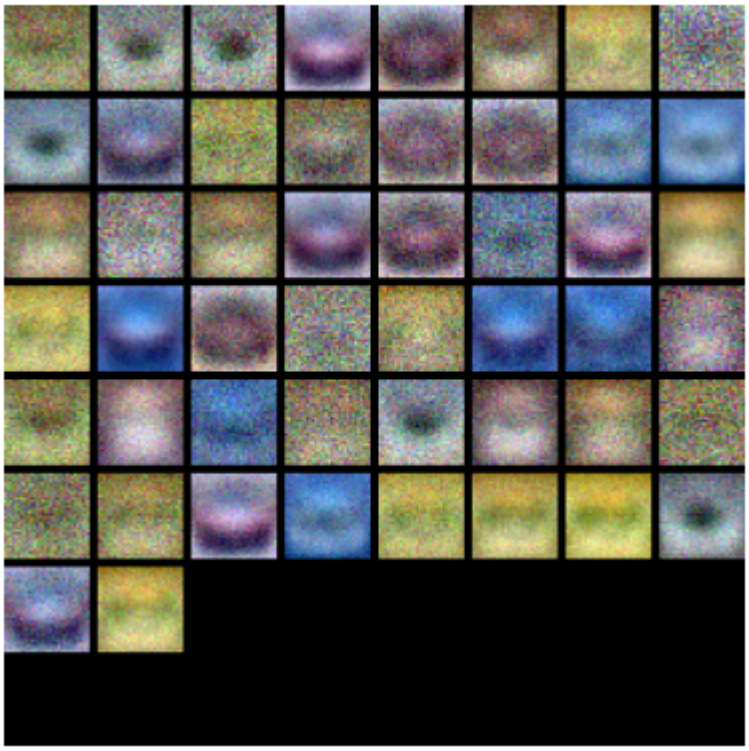
```
iteration 600 / 6000: loss 2.3026070348816576
iteration 700 / 6000: loss 2.3025672488781286
iteration 800 / 6000: loss 2.302511043748264
iteration 900 / 6000: loss 2.3025020673744923
iteration 1000 / 6000: loss 2.302416327991673
iteration 1100 / 6000: loss 2.3022083751559155
iteration 1200 / 6000: loss 2.3022747409308537
iteration 1300 / 6000: loss 2.30219325838239
iteration 1400 / 6000: loss 2.3018614049182693
iteration 1500 / 6000: loss 2.3015019270228256
iteration 1600 / 6000: loss 2.301536483683131
iteration 1700 / 6000: loss 2.301767713274723
iteration 1800 / 6000: loss 2.300709678050533
iteration 1900 / 6000: loss 2.301290650219527
iteration 2000 / 6000: loss 2.3010291716275875
iteration 2100 / 6000: loss 2.299650098794119
iteration 2200 / 6000: loss 2.299737857512051
iteration 2300 / 6000: loss 2.2984253983447562
iteration 2400 / 6000: loss 2.2980365011404196
iteration 2500 / 6000: loss 2.2992992291319685
iteration 2600 / 6000: loss 2.299627689999791
iteration 2700 / 6000: loss 2.297543354053057
iteration 2800 / 6000: loss 2.2952355551610513
iteration 2900 / 6000: loss 2.2927049332611946
iteration 3000 / 6000: loss 2.293678763848068
iteration 3100 / 6000: loss 2.293973688735582
iteration 3200 / 6000: loss 2.293091890357758
iteration 3300 / 6000: loss 2.2846736621635326
iteration 3400 / 6000: loss 2.2871805688936333
iteration 3500 / 6000: loss 2.2855967169100486
iteration 3600 / 6000: loss 2.284357044393651
iteration 3700 / 6000: loss 2.2782899805748733
iteration 3800 / 6000: loss 2.2745005389183905
iteration 3900 / 6000: loss 2.275556564698514
iteration 4000 / 6000: loss 2.267487014201895
iteration 4100 / 6000: loss 2.2738931178358768
iteration 4200 / 6000: loss 2.2680568931107175
iteration 4300 / 6000: loss 2.271036059034352
iteration 4400 / 6000: loss 2.250629004050632
iteration 4500 / 6000: loss 2.2535301108529686
iteration 4600 / 6000: loss 2.259513023649877
iteration 4700 / 6000: loss 2.252526427701579
iteration 4800 / 6000: loss 2.254006278577032
iteration 4900 / 6000: loss 2.241832354679842
iteration 5000 / 6000: loss 2.2465206320433695
iteration 5100 / 6000: loss 2.2488472393825907
iteration 5200 / 6000: loss 2.2281221888271623
iteration 5300 / 6000: loss 2.2382524787755127
iteration 5400 / 6000: loss 2.2408751294044436
iteration 5500 / 6000: loss 2.2363632236741484
iteration 5600 / 6000: loss 2.2471540583821232
iteration 5700 / 6000: loss 2.2540115695449665
iteration 5800 / 6000: loss 2.249283057621046
iteration 5900 / 6000: loss 2.2368603802236895
Validation accuracy:  0.198
Current learning rate:  0.0001
iteration 0 / 6000: loss 2.3027523109667447
iteration 100 / 6000: loss 2.3021854434913176
iteration 200 / 6000: loss 2.298166294976898
iteration 300 / 6000: loss 2.262526100713673
iteration 400 / 6000: loss 2.2058669247935776
```

```
iteration 500 / 6000: loss 2.1381464515777897
iteration 600 / 6000: loss 2.1036514155918535
iteration 700 / 6000: loss 2.0594896655549246
iteration 800 / 6000: loss 2.0366079150168708
iteration 900 / 6000: loss 1.9146904866884107
iteration 1000 / 6000: loss 1.9312527048617063
iteration 1100 / 6000: loss 1.9074859034861307
iteration 1200 / 6000: loss 1.997314819472955
iteration 1300 / 6000: loss 1.8867853328379658
iteration 1400 / 6000: loss 1.9830802909845027
iteration 1500 / 6000: loss 1.6580392809598494
iteration 1600 / 6000: loss 1.8514605985629005
iteration 1700 / 6000: loss 1.8313330786671587
iteration 1800 / 6000: loss 1.9347685017299059
iteration 1900 / 6000: loss 1.8975711187583773
iteration 2000 / 6000: loss 1.8135942868523232
iteration 2100 / 6000: loss 1.739037408726185
iteration 2200 / 6000: loss 1.7635052637566293
iteration 2300 / 6000: loss 1.7778587734499245
iteration 2400 / 6000: loss 1.860647488254527
iteration 2500 / 6000: loss 1.7451665341659566
iteration 2600 / 6000: loss 1.756881437270907
iteration 2700 / 6000: loss 1.7143128795162645
iteration 2800 / 6000: loss 1.7118844235931565
iteration 2900 / 6000: loss 1.7051128038606658
iteration 3000 / 6000: loss 1.7787943983776386
iteration 3100 / 6000: loss 1.4481164494113663
iteration 3200 / 6000: loss 1.643941722674297
iteration 3300 / 6000: loss 1.6508049684979538
iteration 3400 / 6000: loss 1.6349524882725375
iteration 3500 / 6000: loss 1.644801761334519
iteration 3600 / 6000: loss 1.717269085028796
iteration 3700 / 6000: loss 1.647661081257863
iteration 3800 / 6000: loss 1.7586038246408173
iteration 3900 / 6000: loss 1.7101576079559133
iteration 4000 / 6000: loss 1.5828600355424058
iteration 4100 / 6000: loss 1.710850224891623
iteration 4200 / 6000: loss 1.5818850994640035
iteration 4300 / 6000: loss 1.6226822742466365
iteration 4400 / 6000: loss 1.5794530722703617
iteration 4500 / 6000: loss 1.6285364651060898
iteration 4600 / 6000: loss 1.8288009831424652
iteration 4700 / 6000: loss 1.6988006269575717
iteration 4800 / 6000: loss 1.6787469755955113
iteration 4900 / 6000: loss 1.5980073772341683
iteration 5000 / 6000: loss 1.6611313639245888
iteration 5100 / 6000: loss 1.5625045265489585
iteration 5200 / 6000: loss 1.7147658090536722
iteration 5300 / 6000: loss 1.6859105439323467
iteration 5400 / 6000: loss 1.670128636683942
iteration 5500 / 6000: loss 1.6193826026862599
iteration 5600 / 6000: loss 1.5478055605738732
iteration 5700 / 6000: loss 1.6041835965683873
iteration 5800 / 6000: loss 1.5014616244770773
iteration 5900 / 6000: loss 1.6162831191704066
Validation accuracy:  0.43
Current learning rate:  0.001
iteration 0 / 6000: loss 2.302756328481318
iteration 100 / 6000: loss 1.9461212550121298
iteration 200 / 6000: loss 1.697576110687341
iteration 300 / 6000: loss 1.7250125483915062
```

```
iteration 400 / 6000: loss 1.6177731688015784
iteration 500 / 6000: loss 1.6003398032960405
iteration 600 / 6000: loss 1.559859892759535
iteration 700 / 6000: loss 1.5254590667177301
iteration 800 / 6000: loss 1.4492035114383475
iteration 900 / 6000: loss 1.539087944583191
iteration 1000 / 6000: loss 1.5171220280124238
iteration 1100 / 6000: loss 1.4627193363712245
iteration 1200 / 6000: loss 1.4477505521330176
iteration 1300 / 6000: loss 1.447764015412449
iteration 1400 / 6000: loss 1.4758072304504524
iteration 1500 / 6000: loss 1.4798457346593645
iteration 1600 / 6000: loss 1.4782316735081764
iteration 1700 / 6000: loss 1.2768191143111087
iteration 1800 / 6000: loss 1.3666078374309203
iteration 1900 / 6000: loss 1.3253896674436156
iteration 2000 / 6000: loss 1.2664653914628123
iteration 2100 / 6000: loss 1.5482195832951884
iteration 2200 / 6000: loss 1.3824775029766827
iteration 2300 / 6000: loss 1.3073928698978514
iteration 2400 / 6000: loss 1.3598722034997857
iteration 2500 / 6000: loss 1.3358141709657865
iteration 2600 / 6000: loss 1.3852410401895743
iteration 2700 / 6000: loss 1.318688949273191
iteration 2800 / 6000: loss 1.2126419671181978
iteration 2900 / 6000: loss 1.3147983613189003
iteration 3000 / 6000: loss 1.4467148185156133
iteration 3100 / 6000: loss 1.266353099644433
iteration 3200 / 6000: loss 1.2949378635506428
iteration 3300 / 6000: loss 1.4232845460423398
iteration 3400 / 6000: loss 1.4118032734833303
iteration 3500 / 6000: loss 1.2636198708361313
iteration 3600 / 6000: loss 1.2497784054513508
iteration 3700 / 6000: loss 1.3293752985734295
iteration 3800 / 6000: loss 1.2768651372227982
iteration 3900 / 6000: loss 1.3048190870093974
iteration 4000 / 6000: loss 1.2838069300615127
iteration 4100 / 6000: loss 1.3858450343085562
iteration 4200 / 6000: loss 1.2041472834348543
iteration 4300 / 6000: loss 1.3360812125533137
iteration 4400 / 6000: loss 1.2518531298151794
iteration 4500 / 6000: loss 1.3050447983553255
iteration 4600 / 6000: loss 1.338364659252226
iteration 4700 / 6000: loss 1.3949491541618557
iteration 4800 / 6000: loss 1.348662163217858
iteration 4900 / 6000: loss 1.3289859340305277
iteration 5000 / 6000: loss 1.3183767059812155
iteration 5100 / 6000: loss 1.3434362914038196
iteration 5200 / 6000: loss 1.183282612926703
iteration 5300 / 6000: loss 1.3351485989775211
iteration 5400 / 6000: loss 1.339260275949583
iteration 5500 / 6000: loss 1.421995477196877
iteration 5600 / 6000: loss 1.3917825384468079
iteration 5700 / 6000: loss 1.2260269411811262
iteration 5800 / 6000: loss 1.2792045580344877
iteration 5900 / 6000: loss 1.3274046520061757
Validation accuracy:  0.508
Validation accuracy:  0.508
```

In [122]:

```
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?

Answer:

(1) In the best net, it is easier to recognize the specific shapes or feature, where the suboptimal net has much more noises.

Evaluate on test set

In [123]:

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

```
Test accuracy:  0.507
```

In [ ]:

```

```

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```python
import numpy as np
import matplotlib.pyplot as plt


class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  D, a hidden layer dimension of H, and performs classification over C classes.
  We train the network with a softmax loss function and L2 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

  The outputs of the second fully-connected layer are the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable self.params, which is a dictionary with the following keys:

    W1: First layer weights; has shape (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    """
```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```
    self.params = {}
    self.params['W1'] = std * np.random.randn(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_size)
    self.params['b2'] = np.zeros(output_size)


  def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optional; if it
      is not passed then we only return scores, and if it is passed then we
      instead return the loss and gradients.
    - reg: Regularization strength.

    Returns:
    If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those parameters
      with respect to the loss function; has the same keys as self.params.
    """
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```
    # Compute the forward pass
    scores = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the output scores of the neural network.  The result
    #   should be (N, C). As stated in the description for this class,
    #   there should not be a ReLU layer after the second FC layer.
    #   The output of the second FC layer is the output scores. Do not
    #   use a for loop in your implementation.
    # ================================================================ #

    relu = lambda x: x * (x > 0) #define the relu function
    h1 = X.dot(W1.transpose()) + b1 #h1 for the first layer
    reluout = relu(h1) #relu activation on h1
    scores = reluout.dot(W2.transpose()) + b2 #h2 for the second layer

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


    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the loss of the neural network.  This includes the
    #   softmax loss and the L2 regularization for W1 and W2. Store the
    #   total loss in teh variable loss.  Multiply the regularization
    #   loss by 0.5 (in addition to the factor reg).
```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```
106     # ================================================================ #
107
108     # scores is num_examples by num_classes
109
110     #calculate the softmax loss
111     softmaxloss = np.sum(np.log( np.sum( np.exp(scores), axis = 1 ) ) - scores[np.arange(scores.shape[0]), y] ) / N
112     loss = softmaxloss + reg * ((np.sum(W1 ** 2)) + np.sum(W2 ** 2)) * 0.5 #L2 regularization by multiplying the
113                                                                            #regularization loss by 0.5(with factor reg)
114
115     # ================================================================ #
116     # END YOUR CODE HERE
117     # ================================================================ #
118
119     grads = {}
120
121     # ================================================================ #
122     # YOUR CODE HERE:
123     #   Implement the backward pass.  Compute the derivatives of the
124     #   weights and the biases.  Store the results in the grads
125     #   dictionary.  e.g., grads['W1'] should store the gradient for
126     #   W1, and be of the same size as W1.
127     # ================================================================ #
128
129     probabilities = np.exp(scores) / np.sum(np.exp(scores), axis = 1, keepdims = True) #calculate the probability
130     probabilities[np.arange(N), y] -= 1
131
132     #store the results in the grads dictionary
133     #for the second layer
134     derivative2 = np.maximum(0, W1.dot(X.transpose()) + b1.reshape([W1.shape[0], 1]))
135     grads['W2'] = (probabilities / N).transpose().dot(derivative2.transpose()) + reg * W2
136     grads['b2'] = np.sum(probabilities / N, axis = 0, keepdims = True)
137
138     #for the first layer
139     derivative1 = (W1.dot(X.transpose()) > 0) * W2.transpose().dot((probabilities / N).transpose())
140     grads['W1'] = derivative1.dot(X) + reg * W1
```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

```
141     grads['b1'] = np.sum(derivative1, axis = 1, keepdims = True).transpose()
142
143     # ============================================================ #
144     # END YOUR CODE HERE
145     # ============================================================ #
146
147     return loss, grads
148
149   def train(self, X, y, X_val, y_val,
150             learning_rate=1e-3, learning_rate_decay=0.95,
151             reg=1e-5, num_iters=100,
152             batch_size=200, verbose=False):
153     """
154     Train this neural network using stochastic gradient descent.
155
156     Inputs:
157     - X: A numpy array of shape (N, D) giving training data.
158     - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
159       X[i] has label c, where 0 <= c < C.
160     - X_val: A numpy array of shape (N_val, D) giving validation data.
161     - y_val: A numpy array of shape (N_val,) giving validation labels.
162     - learning_rate: Scalar giving learning rate for optimization.
163     - learning_rate_decay: Scalar giving factor used to decay the learning rate
164       after each epoch.
165     - reg: Scalar giving regularization strength.
166     - num_iters: Number of steps to take when optimizing.
167     - batch_size: Number of training examples to use per step.
168     - verbose: boolean; if true print progress during optimization.
169     """
170     num_train = X.shape[0]
171     iterations_per_epoch = max(num_train / batch_size, 1)
172
173     # Use SGD to optimize the parameters in self.model
174     loss_history = []
175     train_acc_history = []
```

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```
176     val_acc_history = []
177
178     for it in np.arange(num_iters):
179       X_batch = None
180       y_batch = None
181
182       # ========================================================================== #
183       # YOUR CODE HERE:
184       #   Create a minibatch by sampling batch_size samples randomly.
185       # ========================================================================== #
186
187       minisample = np.random.choice(num_train, batch_size) #generates a random minibatch sample
188       X_batch = X[minisample]
189       y_batch = y[minisample]
190
191       # ========================================================================== #
192       # END YOUR CODE HERE
193       # ========================================================================== #
194
195        # Compute loss and gradients using the current minibatch
196       loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
197       loss_history.append(loss)
198
199       # ========================================================================== #
200       # YOUR CODE HERE:
201       #   Perform a gradient descent step using the minibatch to update
202       #   all parameters (i.e., W1, W2, b1, and b2).
203       # ========================================================================== #
204
205       self.params['W1'] = self.params['W1'] - grads['W1'] * learning_rate #gradient descent on W1
206       self.params['W2'] = self.params['W2'] - grads['W2'] * learning_rate #gradient descent on W2
207       self.params['b1'] = self.params['b1'] - grads['b1'] * learning_rate #gradient descent on b1
208       self.params['b2'] = self.params['b2'] - grads['b2'] * learning_rate #gradient descent on b2
209
210
```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```
211       # ================================================================ #
212       # END YOUR CODE HERE
213       # ================================================================ #
214
215       if verbose and it % 100 == 0:
216         print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
217
218       # Every epoch, check train and val accuracy and decay learning rate.
219       if it % iterations_per_epoch == 0:
220         # Check accuracy
221         train_acc = (self.predict(X_batch) == y_batch).mean()
222         val_acc = (self.predict(X_val) == y_val).mean()
223         train_acc_history.append(train_acc)
224         val_acc_history.append(val_acc)
225
226         # Decay learning rate
227         learning_rate *= learning_rate_decay
228
229     return {
230       'loss_history': loss_history,
231       'train_acc_history': train_acc_history,
232       'val_acc_history': val_acc_history,
233     }
234
235   def predict(self, X):
236     """
237     Use the trained weights of this two-layer network to predict labels for
238     data points. For each data point we predict scores for each of the C
239     classes, and assign each data point to the class with the highest score.
240
241     Inputs:
242     - X: A numpy array of shape (N, D) giving N D-dimensional data points to
243       classify.
244
245     Returns:
```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | solver.py | layer_utils.py | layers.py

```
234
235   def predict(self, X):
236     """
237     Use the trained weights of this two-layer network to predict labels for
238     data points. For each data point we predict scores for each of the C
239     classes, and assign each data point to the class with the highest score.
240
241     Inputs:
242     - X: A numpy array of shape (N, D) giving N D-dimensional data points to
243       classify.
244
245     Returns:
246     - y_pred: A numpy array of shape (N,) giving predicted labels for each of
247       the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
248       to have class c, where 0 <= c < C.
249     """
250     y_pred = None
251
252     # ================================================================ #
253     # YOUR CODE HERE:
254     #   Predict the class given the input data.
255     # ================================================================ #
256
257     f = lambda x: x * (x > 0) #define the relu function
258     h1 = f(X.dot(self.params['W1'].transpose()) + self.params['b1']) #the first layer after relu
259     h2 = h1.dot(self.params['W2'].transpose()) + self.params['b2'] #the second layer(result)
260     y_pred = np.argmax(h2, axis = 1) #get the highest score for each elements of X
261
262
263     # ================================================================ #
264     # END YOUR CODE HERE
265     # ================================================================ #
266
267     return y_pred
268
```

COM SCI 247/HW/HW3/HW3_code/nndl/neural_net.py 1:1 LF UTF-8 Python GitHub Git (0)

Fully connected networks

In the previous notebook, you implemented a simple two-layer neural network class. However, this class is not modular. If you wanted to change the number of layers, you would need to write a new loss and gradient function. If you wanted to optimize the network with different optimizers, you'd need to write new training functions. If you wanted to incorporate regularizations, you'd have to modify the loss and gradient function.

Instead of having to modify functions each time, for the rest of the class, we'll work in a more modular framework where we define forward and backward layers that calculate losses and gradients respectively. Since the forward and backward layers share intermediate values that are useful for calculating both the loss and the gradient, we'll also have these function return "caches" which store useful intermediate values.

The goal is that through this modular design, we can build different sized neural networks for various applications.

In this HW #3, we'll define the basic architecture, and in HW #4, we'll build on this framework to implement different optimizers and regularizations (like BatchNorm and Dropout).

Modular layers

This notebook will build modular layers in the following manner. First, there will be a forward pass for a given layer with inputs (`x`) and return the output of that layer (`out`) as well as cached variables (`cache`) that will be used to calculate the gradient in the backward pass.

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive derivative of loss with respect to outputs and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

In [66]:

```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient, eval_numerical_gradient_ar
from utils.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%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))))
```

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

In [67]:

```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

Linear layers

In this section, we'll implement the forward and backward pass for the linear layers.

The linear layer forward pass is the function `affine_forward` in `nndl/layers.py` and the backward pass is `affine_backward`.

After you have implemented these, test your implementation by running the cell below.

Affine layer forward pass

Implement `affine_forward` and then test your code by running the following cell.

In [68]:

```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print('Testing affine_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing affine_forward function:
difference: 9.7698500479884e-10
```

Affine layer backward pass

Implement `affine_backward` and then test your code by running the following cell.

In [69]:

```
# Test the affine_backward function

x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dout
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around 1e-10
print('Testing affine_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_backward function:
dx error: 2.457463990122334e-10
dw error: 1.193556602763492e-09
db error: 3.643641422261432e-10
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

Implement the `relu_forward` function in `nndl/layers.py` and then test your code by running the following cell.

In [70]:

```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be around 1e-8
print('Testing relu_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing relu_forward function:
difference: 4.999999798022158e-08
```

ReLU backward pass

Implement the `relu_backward` function in `nndl/layers.py` and then test your code by running the following cell.

In [71]:

```
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 1e-12
print('Testing relu_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing relu_backward function:
dx error: 3.2756117919377313e-12
```

Combining the affine and ReLU layers

Often times, an affine layer will be followed by a ReLU layer. So let's make one that puts them together. Layers that are combined are stored in `nndl/layer_utils.py` .

Affine-ReLU layers

We've implemented `affine_relu_forward()` and `affine_relu_backward` in `nndl/layer_utils.py`. Take a look at them to make sure you understand what's going on. Then run the following cell to ensure its implemented correctly.

In [72]:

```
from nndl.layer_utils import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x,
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w,
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b,

print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error: 2.0556850132323285e-10
dw error: 1.4955311691194157e-09
db error: 1.8928846578151878e-11
```

Softmax loss

You've already implemented it, so we have written it in `layers.py`. The following code will ensure they are working correctly.

In [73]:

```
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)



dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing softmax_loss:
loss: 2.3028293071217116
dx error: 9.871461667174707e-09
```

Implementation of a two-layer NN

In `nndl/fc_net.py`, implement the class `TwoLayerNet` which uses the layers you made here. When you have finished, the following cell will test your implementation.

In [74]:

```
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-2
model = TwoLayerNet(input_dim=D, hidden_dims=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = {}'.format(reg))
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.2236151215593397e-08
W2 relative error: 3.342953960692366 5e-10
b1 relative error: 4.7288944058018464e-09
```

```
b2 relative error: 4.3291285233961314e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.527915286171985e-07
W2 relative error: 1.3678335722105113e-07
b1 relative error: 1.5646801749611563e-08
b2 relative error: 9.089621155678095e-10
```

Solver

We will now use the utils Solver class to train these networks. Familiarize yourself with the API in `utils/solver.py` . After you have done so, declare an instance of a TwoLayerNet with 200 units and then train it with the Solver. Choose parameters so that your validation accuracy is at least 50%.

In [75]:

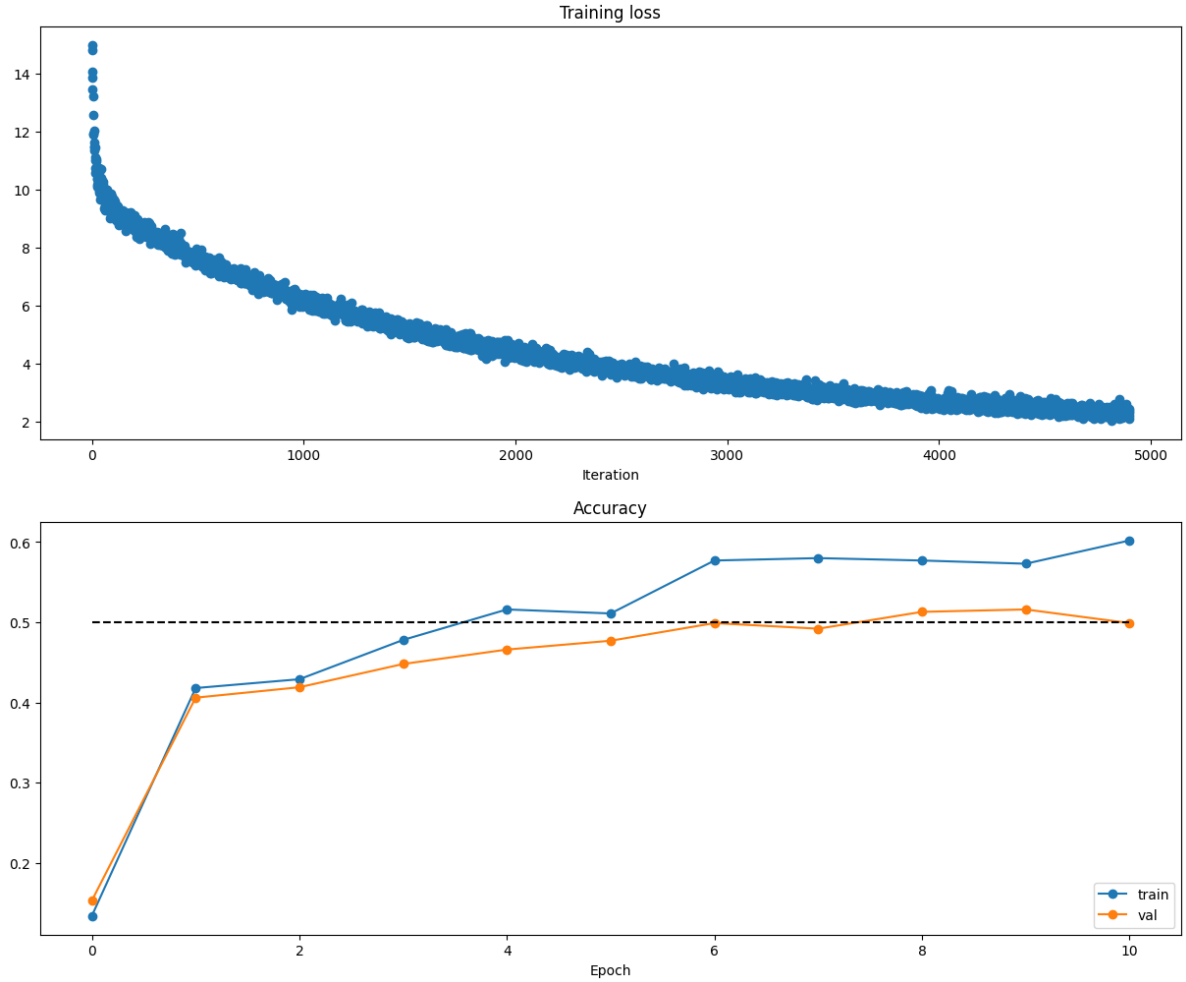
```
model = TwoLayerNet()
solver = None

# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 50%.  We won't have you optimize this further
#   since you did it in the previous notebook.
#
# ================================================================ #

#initialize the input size, hidden size, and the number of classes
input_size = 3072
hidden_size = 200
numclass = 10
model = TwoLayerNet(input_dim = input_size, hidden_dims = hidden_size,
                    num_classes = numclass, weight_scale = std, reg = 0.25)
                                            #this reg is from two_layer_nn

solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-3,
                        #this learning rate is the best from two_layer_nn
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
solver.train()

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

```
(Iteration 1 / 4900) loss: 13.473065
(Epoch 0 / 10) train acc: 0.134000; val_acc: 0.153000
(Iteration 101 / 4900) loss: 9.363778
(Iteration 201 / 4900) loss: 8.840233
(Iteration 301 / 4900) loss: 8.443530
(Iteration 401 / 4900) loss: 7.948557
(Epoch 1 / 10) train acc: 0.418000; val_acc: 0.406000
(Iteration 501 / 4900) loss: 7.680017
(Iteration 601 / 4900) loss: 7.018495
(Iteration 701 / 4900) loss: 6.795612
(Iteration 801 / 4900) loss: 6.632900
(Iteration 901 / 4900) loss: 6.685198
(Epoch 2 / 10) train acc: 0.429000; val_acc: 0.419000
(Iteration 1001 / 4900) loss: 6.055642
(Iteration 1101 / 4900) loss: 5.720069
(Iteration 1201 / 4900) loss: 5.818843
(Iteration 1301 / 4900) loss: 5.494520
(Iteration 1401 / 4900) loss: 5.261642
(Epoch 3 / 10) train acc: 0.478000; val_acc: 0.448000
(Iteration 1501 / 4900) loss: 5.407405
(Iteration 1601 / 4900) loss: 4.941966
(Iteration 1701 / 4900) loss: 4.750067
(Iteration 1801 / 4900) loss: 4.586770
(Iteration 1901 / 4900) loss: 4.572251
```

```
(Epoch 4 / 10) train acc: 0.516000; val_acc: 0.466000
(Iteration 2001 / 4900) loss: 4.494158
(Iteration 2101 / 4900) loss: 4.371851
(Iteration 2201 / 4900) loss: 4.032794
(Iteration 2301 / 4900) loss: 4.195725
(Iteration 2401 / 4900) loss: 4.024897
(Epoch 5 / 10) train acc: 0.511000; val_acc: 0.477000
(Iteration 2501 / 4900) loss: 3.970599
(Iteration 2601 / 4900) loss: 3.617921
(Iteration 2701 / 4900) loss: 3.573335
(Iteration 2801 / 4900) loss: 3.679093
(Iteration 2901 / 4900) loss: 3.344997
(Epoch 6 / 10) train acc: 0.577000; val_acc: 0.499000
(Iteration 3001 / 4900) loss: 3.229398
(Iteration 3101 / 4900) loss: 3.307908
(Iteration 3201 / 4900) loss: 3.221081
(Iteration 3301 / 4900) loss: 3.081173
(Iteration 3401 / 4900) loss: 3.019923
(Epoch 7 / 10) train acc: 0.580000; val_acc: 0.492000
(Iteration 3501 / 4900) loss: 2.813302
(Iteration 3601 / 4900) loss: 2.817736
(Iteration 3701 / 4900) loss: 2.973856
(Iteration 3801 / 4900) loss: 2.663095
(Iteration 3901 / 4900) loss: 2.751516
(Epoch 8 / 10) train acc: 0.577000; val_acc: 0.513000
(Iteration 4001 / 4900) loss: 2.558269
(Iteration 4101 / 4900) loss: 2.638850
(Iteration 4201 / 4900) loss: 2.572045
(Iteration 4301 / 4900) loss: 2.473713
(Iteration 4401 / 4900) loss: 2.682496
(Epoch 9 / 10) train acc: 0.573000; val_acc: 0.516000
(Iteration 4501 / 4900) loss: 2.375608
(Iteration 4601 / 4900) loss: 2.579251
(Iteration 4701 / 4900) loss: 2.416563
(Iteration 4801 / 4900) loss: 2.408897
(Epoch 10 / 10) train acc: 0.602000; val_acc: 0.499000
```

In [76]:

```
# 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([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



Multilayer Neural Network

Now, we implement a multi-layer neural network.

Read through the `FullyConnectedNet` class in the file `nndl/fc_net.py` .

Implement the initialization, the forward pass, and the backward pass. There will be lines for batchnorm and dropout layers and caches; ignore these all for now. That'll be in HW #4.

In [77]:

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = {}'.format(reg))
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: {}'.format(loss))

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads[name])))
```

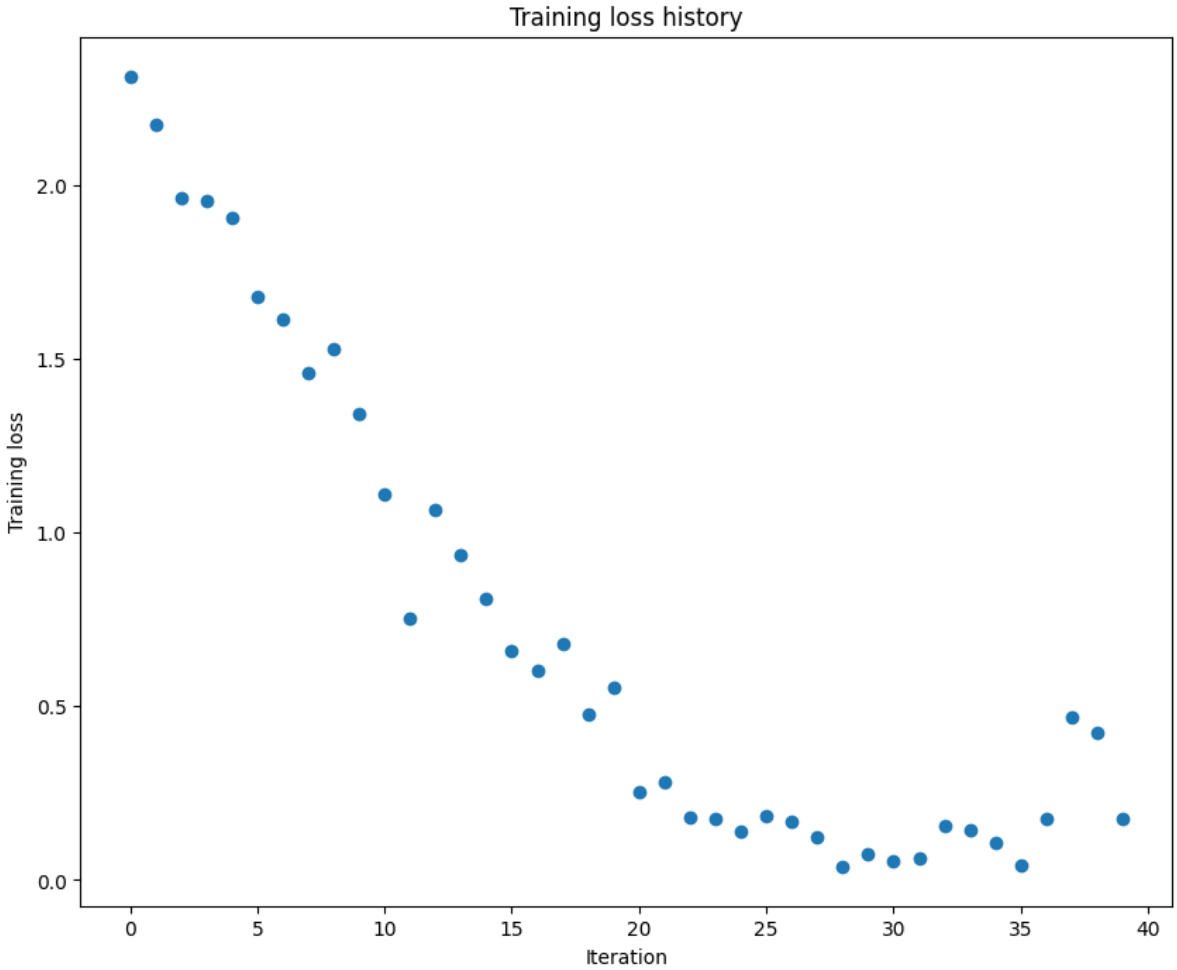
```
Running check with reg = 0
Initial loss: 2.3024754666449287
W1 relative error: 1.6781556810519585e-05
W2 relative error: 9.291991164949132e-07
W3 relative error: 4.622234571345717e-08
b1 relative error: 5.004219688120511e-08
b2 relative error: 4.94979589225779e-09
b3 relative error: 1.253765500153131e-10
Running check with reg = 3.14
Initial loss: 6.9465939534832435
W1 relative error: 5.809813796582568e-08
W2 relative error: 2.174329219567837e-08
W3 relative error: 1.4242724122271282e-08
b1 relative error: 7.304729433506879e-08
b2 relative error: 4.8929082312357747e-08
b3 relative error: 1.7624436790913813e-10
```

In [78]:

```
# Use the three layer neural network to overfit a small dataset.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}


#### !!!!!!
# Play around with the weight_scale and learning_rate so that you can overfit a smal
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 1e-2
learning_rate = 1e-2

model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 2.314006
(Epoch 0 / 20) train acc: 0.300000; val_acc: 0.144000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.139000
(Epoch 2 / 20) train acc: 0.480000; val_acc: 0.159000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.146000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.100000
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.152000
(Iteration 11 / 40) loss: 1.111340
(Epoch 6 / 20) train acc: 0.760000; val_acc: 0.179000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.167000
(Epoch 8 / 20) train acc: 0.920000; val_acc: 0.177000
(Epoch 9 / 20) train acc: 0.880000; val_acc: 0.157000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.169000
(Iteration 21 / 40) loss: 0.254163
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.176000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.204000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.184000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.173000
(Iteration 31 / 40) loss: 0.052834
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.160000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.159000
(Epoch 18 / 20) train acc: 0.960000; val_acc: 0.186000
```

```
(Epoch 19 / 20) train acc: 0.960000; val_acc: 0.149000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.189000
```



In [ ]:

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```python
import numpy as np
import pdb


def affine_forward(x, w, b):
  """
  Computes the forward pass for an affine (fully-connected) layer.

  The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
  examples, where each example x[i] has shape (d_1, ..., d_k). We will
  reshape each input into a vector of dimension D = d_1 * ... * d_k, and
  then transform it to an output vector of dimension M.

  Inputs:
  - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
  - w: A numpy array of weights, of shape (D, M)
  - b: A numpy array of biases, of shape (M,)

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ============================================================ #

  xreshape = x.reshape(x.shape[0], -1) #reshape the input into a vector
  out = xreshape.dot(w) + b #the transition between each layer will be Wx + b

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

COM SCI 247/HW/HW3/HW3_code/nndl/layers.py 55:50 (2, 6) LF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```
38    cache = (x, w, b)
39    return out, cache
40
41
42  def affine_backward(dout, cache):
43    """
44    Computes the backward pass for an affine layer.
45
46    Inputs:
47    - dout: Upstream derivative, of shape (N, M)
48    - cache: Tuple of:
49      - x: Input data, of shape (N, d_1, ... d_k)
50      - w: Weights, of shape (D, M)
51
52    Returns a tuple of:
53    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
54    - dw: Gradient with respect to w, of shape (D, M)
55    - db: Gradient with respect to b, of shape (M,)
56    """
57    x, w, b = cache
58    dx, dw, db = None, None, None
59
60    # ============================================================ #
61    # YOUR CODE HERE:
62    #   Calculate the gradients for the backward pass.
63    # ============================================================ #
64
65    # dout is N x M
66    # dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x M
67    # dw should be D x M; it relates to dout through multiplication with x, which is N x D after reshaping
68    # db should be M; it is just the sum over dout examples
69
70    xreshape = x.reshape(x.shape[0], -1) #reshape the input
71    dx = np.reshape(dout.dot(w.transpose()), x.shape) #dout through multiplication with w
72                                                      #its size should be the same as x's
```

COM SCI 247/HW/HW3/HW3_code/nndl/layers.py 39:20 LF UTF-8 Python GitHub Git (0)

```
    xreshape = x.reshape(x.shape[0], -1) #reshape the input
    dx = np.reshape(dout.dot(w.transpose()), x.shape) #dout through multiplication with w
                                                      #its size should be the same as x's
    dw = (xreshape.transpose()).dot(dout) #dout through multiplication with x
    db = np.sum(dout, axis = 0) #sum over dout examples

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

    return dx, dw, db

def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ============================================================ #

    out = np.maximum(x, 0) #based on the definition of the relu function

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

    cache = x
```

```
105   return out, cache
106
107
108 def relu_backward(dout, cache):
109   """
110   Computes the backward pass for a layer of rectified linear units (ReLUs).
111
112   Input:
113   - dout: Upstream derivatives, of any shape
114   - cache: Input x, of same shape as dout
115
116   Returns:
117   - dx: Gradient with respect to x
118   """
119   x = cache
120
121   # ============================================================ #
122   # YOUR CODE HERE:
123   #   Implement the ReLU backward pass
124   # ============================================================ #
125
126   # ReLU directs linearly to those > 0
127
128   dx = (x > 0) * (dout) #based on the derivative of the relu function
129
130   # ============================================================ #
131   # END YOUR CODE HERE
132   # ============================================================ #
133
134   return dx
135
136
137 def softmax_loss(x, y):
138   """
139   Computes the loss and gradient for softmax classification.
```

```
126   # ReLU directs linearly to those > 0
127
128   dx = (x > 0) * (dout) #based on the derivative of the relu function
129
130   # ============================================================ #
131   # END YOUR CODE HERE
132   # ============================================================ #
133
134   return dx
135
136
137 def softmax_loss(x, y):
138   """
139   Computes the loss and gradient for softmax classification.
140
141   Inputs:
142   - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
143     for the ith input.
144   - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
145     0 <= y[i] < C
146
147   Returns a tuple of:
148   - loss: Scalar giving the loss
149   - dx: Gradient of the loss with respect to x
150   """
151
152   probs = np.exp(x - np.max(x, axis=1, keepdims=True))
153   probs /= np.sum(probs, axis=1, keepdims=True)
154   N = x.shape[0]
155   loss = -np.sum(np.log(probs[np.arange(N), y])) / N
156   dx = probs.copy()
157   dx[np.arange(N), y] -= 1
158   dx /= N
159   return loss, dx
160
```

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```python
import numpy as np

from .layers import *
from .layer_utils import *


class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network with ReLU nonlinearity and
  softmax loss that uses a modular layer design. We assume an input dimension
  of D, a hidden dimension of H, and perform classification over C classes.

  The architecure should be affine - relu - affine - softmax.

  Note that this class does not implement gradient descent; instead, it
  will interact with a separate Solver object that is responsible for running
  optimization.

  The learnable parameters of the model are stored in the dictionary
  self.params that maps parameter names to numpy arrays.
  """

  def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
               dropout=0, weight_scale=1e-3, reg=0.0):
    """
    Initialize a new network.

    Inputs:
    - input_dim: An integer giving the size of the input
    - hidden_dims: An integer giving the size of the hidden layer
    - num_classes: An integer giving the number of classes to classify
    - dropout: Scalar between 0 and 1 giving dropout strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - reg: Scalar giving L2 regularization strength.
```

COM SCI 247/HW/HW3/HW3_code/nndl/fc_net.py 30:66 CRLF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```
37      self.params = {}
38      self.reg = reg
39
40      # ============================================================================ #
41      # YOUR CODE HERE:
42      #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
43      #   self.params['W2'], self.params['b1'] and self.params['b2']. The
44      #   biases are initialized to zero and the weights are initialized
45      #   so that each parameter has mean 0 and standard deviation weight_scale.
46      #   The dimensions of W1 should be (input_dim, hidden_dim) and the
47      #   dimensions of W2 should be (hidden_dims, num_classes)
48      # ============================================================================ #
49
50      self.params['b1'] = np.zeros(hidden_dims) #based on the size of W1
51      self.params['b2'] = np.zeros(num_classes) #based on the size of W2
52
53      #the weights will have mean 0 and standard deviation weight_scale
54      self.params['W1'] = np.random.normal(0, weight_scale, (input_dim, hidden_dims))
55      self.params['W2'] = np.random.normal(0, weight_scale, (hidden_dims, num_classes))
56
57      # ============================================================================ #
58      # END YOUR CODE HERE
59      # ============================================================================ #
60
61    def loss(self, X, y=None):
62      """
63      Compute loss and gradient for a minibatch of data.
64
65      Inputs:
66      - X: Array of input data of shape (N, d_1, ..., d_k)
67      - y: Array of labels, of shape (N,). y[i] gives the label for X[i].
68
69      Returns:
70      If y is None, then run a test-time forward pass of the model and return:
71      - scores: Array of shape (N, C) giving classification scores, where
```

```
    Returns:
    If y is None, then run a test-time forward pass of the model and return:
    - scores: Array of shape (N, C) giving classification scores, where
      scores[i, c] is the classification score for X[i] and class c.

    If y is not None, then run a training-time forward and backward pass and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping parameter
      names to gradients of the loss with respect to those parameters.
    """
    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ============================================================ #

    #out1, cache1 = affine_forward(X, self.params['W1'], self.params['b1']) #do the first layer affine forward
    #out2, cache2 = relu_forward(out1) #relu activation
    out1, cache1 = affine_relu_forward(X, self.params['W1'], self.params['b1']) #do the first layer affine followed by relu
    scores, cachescores = affine_forward(out1, self.params['W2'], self.params['b2']) #do the second layer affine forward


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

    # If y is None then we are in test mode so just return scores
    if y is None:
      return scores

    loss, grads = 0, {}
```

```
103     loss, grads = 0, {}
104     # ============================================================ #
105     # YOUR CODE HERE:
106     #   Implement the backward pass of the two-layer neural net.  Store
107     #   the loss as the variable 'loss' and store the gradients in the
108     #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
109     #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
110     #   i.e., grads[k] holds the gradient for self.params[k].
111     #
112     #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
113     #   for each W.  Be sure to include the 0.5 multiplying factor to
114     #   match our implementation.
115     #
116     #   And be sure to use the layers you prior implemented.
117     # ============================================================ #
118
119     loss, dx = softmax_loss(scores, y) #calculate loss and dx by given softmax_loss
120     loss += (self.reg) * (np.sum(self.params['W1'] ** 2) + np.sum(self.params['W2'] ** 2)) * 0.5 #L2 regularization by multiplying the
121                                                             #regularization loss by 0.5(with factor reg)
122
123     dx2, dw2, db2 = affine_backward(dx, cachescores) #backward affine of the second layer
124     dx1, dw1, db1 = affine_relu_backward(dx2, cache1) #backward pass for the affine-relu convenience layer
125
126     #store the results in the grads dictionary
127     grads['W1'] = dw1 + (self.reg) * self.params['W1']
128     grads['W2'] = dw2 + (self.reg) * self.params['W2']
129     grads['b1'] = db1
130     grads['b2'] = db2
131
132
133     # ============================================================ #
134     # END YOUR CODE HERE
135     # ============================================================ #
136
137     return loss, grads
```

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```
class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden layers,
  ReLU nonlinearities, and a softmax loss function. This will also implement
  dropout and batch normalization as options. For a network with L layers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

  where batch normalization and dropout are optional, and the {...} block is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
    """
    Initialize a new FullyConnectedNet.

    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden layer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify.
    - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
      the network should not use dropout at all.
    - use_batchnorm: Whether or not the network should use batch normalization.
    - reg: Scalar giving L2 regularization strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - dtype: A numpy datatype object; all computations will be performed using
      this datatype. float32 is faster but less accurate, so you should use
      float64 for numeric gradient checking.
```

COM SCI 247/HW/HW3/HW3_code/nndl/fc_net.py 30:66 CRLF UTF-8 Python GitHub Git (0)

```
    - seed: If not None, then pass this random seed to the dropout layers. This
      will make the dropout layers deteriminstic so we can gradient check the
      model.
    """
    self.use_batchnorm = use_batchnorm
    self.use_dropout = dropout > 0
    self.reg = reg
    self.num_layers = 1 + len(hidden_dims)
    self.dtype = dtype
    self.params = {}

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize all parameters of the network in the self.params dictionary.
    #   The weights and biases of layer 1 are W1 and b1; and in general the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    # ============================================================ #

    for i in np.arange (1, self.num_layers + 1):

        #we will consider three instances:

        if i == 1: #for the layer 1

            self.params['b1'] = np.zeros(hidden_dims[i - 1]) #based on the size of the first hidden layer
            #the weights will have mean 0 and standard deviation weight_scale
            self.params['W1'] = np.random.normal(0, weight_scale, (input_dim, hidden_dims[i - 1]))

        elif i == self.num_layers: #for the last layer

            self.params['b' + str(i)] = np.zeros(num_classes) #the size after the last layer will be the number of classes
            self.params['W' + str(i)] = np.random.normal(0, weight_scale, (hidden_dims[i - 2], num_classes))

```

```
210         else: #for the layer except the first and the last
211 
212             self.params['b' + str(i)] = np.zeros(hidden_dims[i - 1]) #based on the size of the ith hidden layer
213             self.params['W' + str(i)] = np.random.normal(0, weight_scale, (hidden_dims[i - 2], hidden_dims[i - 1]))
214 
215 
216 
217     # ============================================================ #
218     # END YOUR CODE HERE
219     # ============================================================ #
220 
221     # When using dropout we need to pass a dropout_param dictionary to each
222     # dropout layer so that the layer knows the dropout probability and the mode
223     # (train / test). You can pass the same dropout_param to each dropout layer.
224     self.dropout_param = {}
225     if self.use_dropout:
226       self.dropout_param = {'mode': 'train', 'p': dropout}
227       if seed is not None:
228         self.dropout_param['seed'] = seed
229 
230     # With batch normalization we need to keep track of running means and
231     # variances, so we need to pass a special bn_param object to each batch
232     # normalization layer. You should pass self.bn_params[0] to the forward pass
233     # of the first batch normalization layer, self.bn_params[1] to the forward
234     # pass of the second batch normalization layer, etc.
235     self.bn_params = []
236     if self.use_batchnorm:
237       self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]
238 
239     # Cast all parameters to the correct datatype
240     for k, v in self.params.items():
241       self.params[k] = v.astype(dtype)
242 
243 
244   def loss(self, X, y=None):
```

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```
244   def loss(self, X, y=None):
245     """
246     Compute loss and gradient for the fully-connected net.
247
248     Input / output: Same as TwoLayerNet above.
249     """
250     X = X.astype(self.dtype)
251     mode = 'test' if y is None else 'train'
252
253     # Set train/test mode for batchnorm params and dropout param since they
254     # behave differently during training and testing.
255     if self.dropout_param is not None:
256       self.dropout_param['mode'] = mode
257     if self.use_batchnorm:
258       for bn_param in self.bn_params:
259         bn_param[mode] = mode
260
261     scores = None
262
263     # ============================================================ #
264     # YOUR CODE HERE:
265     #   Implement the forward pass of the FC net and store the output
266     #   scores as the variable "scores".
267     # ============================================================ #
268
269     out_list = []
270     cache_list = []
271
272     for i in np.arange (1, self.num_layers + 1):
273
274         #we will consider three instances:
275
276         if i == 1: #for the layer 1
277
278             out1, cache1 = affine_relu_forward(X, self.params['W' + str(i)], self.params['b' + str(i)]) #do the first layer affine
```

COM SCI 247/HW/HW3/HW3_code/nndl/fc_net.py 30:66 CRLF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```
278             out1, cache1 = affine_relu_forward(X, self.params['W' + str(i)], self.params['b' + str(i)]) #do the first layer affine
279                                                                                                          #followed by relu
280             out_list.append(out1)
281             cache_list.append(cache1)
282 
283         elif i == self.num_layers: #for the last layer
284 
285             scores, cachescores = affine_forward(out_list[i - 2], self.params['W' + str(i)], self.params['b' + str(i)]) #do the last
286                                                                                                                          #layer affine forward
287             cache_list.append(cachescores)
288 
289         else: #for the layers except the first and the last
290 
291             outnow, cachenow = affine_relu_forward(out_list[i - 2], self.params['W' + str(i)], self.params['b' + str(i)]) #do the affine
292                                                                                                                          #followed by relu
293             out_list.append(outnow)
294             cache_list.append(cachenow)
295 
296 
297     # ============================================================ #
298     # END YOUR CODE HERE
299     # ============================================================ #
300 
301     # If test mode return early
302     if mode == 'test':
303       return scores
304 
305     loss, grads = 0.0, {}
306     # ============================================================ #
307     # YOUR CODE HERE:
308     #   Implement the backwards pass of the FC net and store the gradients
309     #   in the grads dict, so that grads[k] is the gradient of self.params[k]
310     #   Be sure your L2 regularization includes a 0.5 factor.
311     # ============================================================ #
312 
```

COM SCI 247/HW/HW3/HW3_code/nndl/fc_net.py 30:66 CRLF UTF-8 Python GitHub Git (0)

Welcome Guide | neural_net.py | fc_net.py | layer_utils.py | optim.py | layers.py

```
301     # If test mode return early
302     if mode == 'test':
303       return scores
304
305     loss, grads = 0.0, {}
306     # ============================================================ #
307     # YOUR CODE HERE:
308     #   Implement the backwards pass of the FC net and store the gradients
309     #   in the grads dict, so that grads[k] is the gradient of self.params[k]
310     #   Be sure your L2 regularization includes a 0.5 factor.
311     # ============================================================ #
312
313     loss, dx = softmax_loss(scores, y) #calculate loss and dx by given softmax_loss
314
315     for j in np.arange (self.num_layers, 0, -1): #loop from the last layer to the first layer
316         loss += (self.reg) * (np.sum(self.params['W' + str(j)] ** 2)) * 0.5 #L2 regularization by multiplying the
317                                                     #regularization loss by 0.5(with factor reg)
318
319         if j == self.num_layers: #from the last layer
320             #backward affine of the second layer
321             currentdx, grads['W' + str(j)], grads['b' + str(j)] = affine_backward(dx, cache_list[j - 1])
322
323         else: #for other layers:
324             #backward pass for the affine-relu convenience layer
325             currentdx, grads['W' + str(j)], grads['b' + str(j)] = affine_relu_backward(currentdx, cache_list[j - 1])
326
327         #store the results in the grads dictionary
328         grads['W' + str(j)] += (self.reg) * self.params['W' + str(j)]
329
330
331     # ============================================================ #
332     # END YOUR CODE HERE
333     # ============================================================ #
334     return loss, grads
335
```