Problem 1

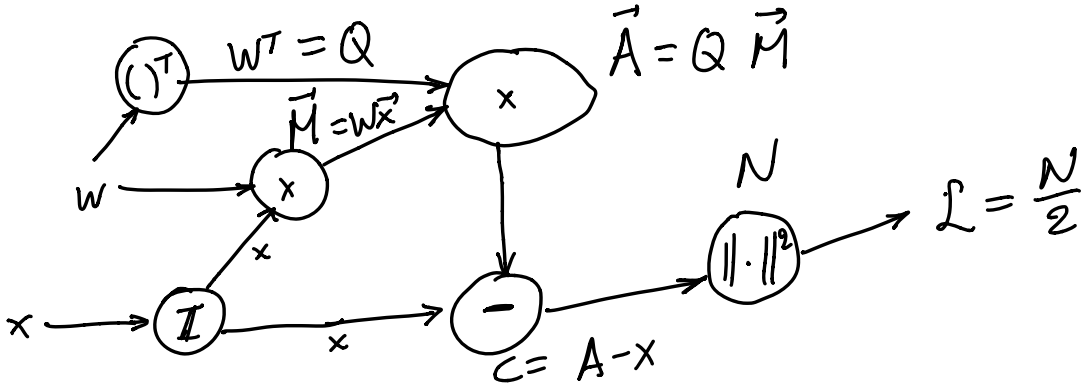
$x \in \mathbb{R}^n$

$W \in \mathbb{R}^{m \times n} \qquad m < n$

$$\mathcal{L} = \frac{1}{2} \| W^T W x - x \|^2$$

a) If we were able to drive $\mathcal{L} \geq 0$ all the way to zero, then $W^T W = \mathbb{I}_n$, which means that components of $W^T W x$ and x are the same. Moreover the image $y = Wx$ can be said to preserve some information of x when $W^T W = \mathbb{I}$ because Orthogonal matrices preserve some information about vectors since they are isometries and isometries preserve some structure of the vector space.

b) $\| W\vec{x} - \vec{x} \|^2$



c) At the top of the computational graph, we see the two paths to W.
These converging paths come from the fact that there is dependence on W at two places
If we call $Q = W^T$
we see that $\mathcal{L} = \mathcal{L}(Q(W), W, x)$

$$\frac{d\mathcal{L}}{dW} = \frac{\partial Q}{\partial W}\frac{\partial \mathcal{L}}{\partial Q} + \frac{\partial \mathcal{L}}{\partial W}$$

d) $\frac{\partial \mathcal{L}}{\partial N} = \frac{1}{2}$

$$\frac{\partial \mathcal{L}}{\partial \vec{C}} = \frac{\partial N}{\partial \vec{C}}\frac{\partial \mathcal{L}}{\partial N} = 2\vec{C}\,\frac{1}{2} = \vec{C}$$

$$\frac{\partial \mathcal{L}}{\partial \vec{A}} = \frac{\partial \vec{C}}{\partial \vec{A}}\frac{\partial \mathcal{L}}{\partial \vec{C}} = I\,\vec{C}$$

$$\frac{\partial \mathcal{L}}{\partial \vec{M}} = \frac{\partial \vec{A}}{\partial \vec{M}}\frac{\partial \mathcal{L}}{\partial \vec{A}} = Q^T\vec{C}$$

$$\frac{\partial \mathcal{L}}{\partial Q} = \frac{\partial \vec{A}}{\partial Q}\frac{\partial \mathcal{L}}{\partial \vec{A}} = \frac{\partial \mathcal{L}}{\partial \vec{A}}M^T = \vec{C}M^T$$

$$\frac{dL}{dW} = \frac{\partial Q}{\partial W}\frac{\partial L}{\partial Q} + \frac{\partial \vec{\mu}}{\partial W}\frac{\partial L}{\partial \vec{\mu}}$$

$$= Tr\left(\vec{c}\,M^T\right) + Q^T\vec{c}\,\vec{x}^T$$

$$= \mu\,\vec{c}^T + W\left(W^TWx - x\right)x^T$$

$$= Wx\left(W^TWx - x\right)^T + W\left(W^TWx - x\right)x^T$$

$$= -Wxx^T + Wxx^TW^TW + WW^TWxx^T - Wxx^T$$

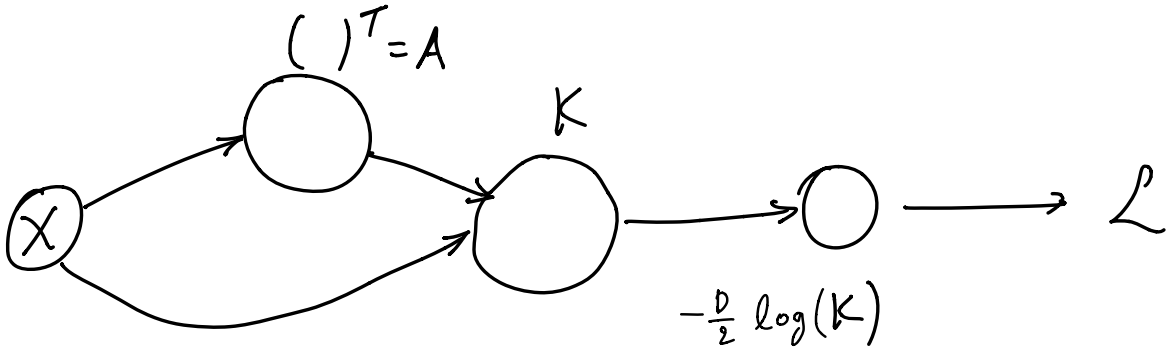
$$= \boxed{Wxx^TW^TW + WW^TWxx^T - 2Wxx^T}$$

Problem 2

$$\mathcal{L} = -c - \frac{D}{2}\log|K| - \frac{1}{2}\,tr(K^{-1}YY^T)$$

$$\mathcal{L}_1 = -\frac{D}{2}\log|\alpha XX^T + \beta^{-1}I|$$

a)



$$\frac{\partial \mathcal{L}_1}{\partial x} = \frac{\partial K}{\partial X}\,\frac{\partial \mathcal{L}_1}{\partial K}$$

$$= -\frac{D}{2}\,\frac{dK}{dx}\,\frac{\partial \log|K|}{\partial K}$$

$$= -\frac{D}{2}\,\frac{1}{K}\,(\alpha\, 2X)$$

$$= -\frac{\alpha D}{2}\,2X\left(\alpha XX^T + \beta^{-1}I\right)^{-1}$$

$$= -\alpha D X\left(\alpha XX^T + \beta^{-1}I\right)^{-1}$$

here have used

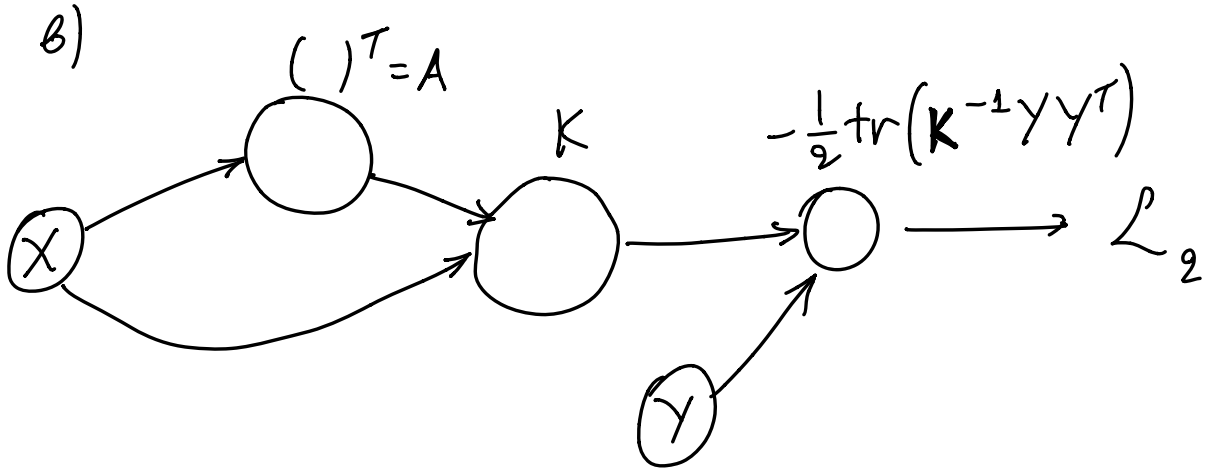
$$\frac{dK}{dX} = \frac{\partial A}{\partial X}\frac{\partial K}{\partial A} + \frac{\partial X}{\partial X}\frac{\partial K}{\partial X}$$

$$= \frac{\partial A}{\partial X}\alpha X^T + \mathbb{I}\alpha A^T$$

$$= \alpha \frac{\partial (X^T)}{\partial X} X^T + \alpha (X^T)^T$$

$$= \alpha 2 X$$

B)



$$\frac{\partial \mathcal{L}_2}{\partial X} = \frac{dK}{\partial X}\frac{\partial \mathcal{L}_2}{\partial K}$$

$$= \left(\frac{\partial A}{\partial X} \frac{\partial K}{\partial A} + \frac{\partial X}{\partial X} \frac{\partial K}{\partial X} \right) \frac{\partial \mathcal{L}_2}{\partial K}$$

$$= \left(-K^{-T} \frac{\partial \mathcal{L}_2}{\partial (K^{-1})} K^{-T} \right) \alpha \mathcal{L} X$$

$$= -\frac{1}{2} \alpha \mathcal{L} \left(-K^{-T} \frac{\partial \, Tr(BYY^T)}{\partial B} K^{-T} \right) X$$

$$= \alpha \; K^{-T} \, YY^T \, K^{-T} X$$

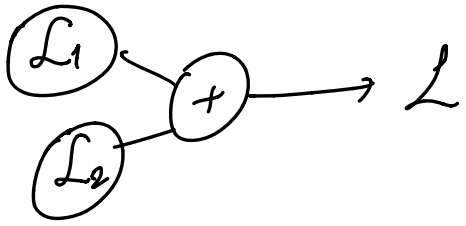
where we used $\frac{\partial}{\partial X} Tr(XA) = A^T$

However $K^{-T} = (K^T)^{-1} = K^{-1}$

hence

$$\frac{\partial \mathcal{L}_2}{\partial X} = \alpha \, K^{-1} YY^T K^{-1} X$$

e) Therefore from



we get

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}_1}{\partial x} + \frac{\partial \mathcal{L}_2}{\partial x}$$

$$= \alpha K^{-1} Y Y^T K^{-1} X$$

$$- \alpha D K^{-1} X$$

two_layer_nn

February 3, 2021

0.1 This is the 2-layer neural network workbook for ECE 247 Assignment #3

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

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

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

[1]:
```
import random
import numpy as np
from cs231n.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))))
```

0.2 Toy example

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

[2]:
```
from nndl.neural_net import TwoLayerNet
```

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

input_size = 4
hidden_size = 10
num_classes = 3
```

```
num_inputs = 5

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

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

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

0.2.1 Compute forward pass scores

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

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

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

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

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

Difference between your scores and correct scores:
3.381231233889892e-08
```

0.2.2 Forward pass loss

[5]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

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

[6]:
```
print(loss)
```

```
1.071696123862817
```

0.2.3 Backward pass

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

[7]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
 ↪verbose=False)
```

```
    print('{} max relative error: {}'.format(param_name,␣
↪rel_error(param_grad_num, grads[param_name])))
```

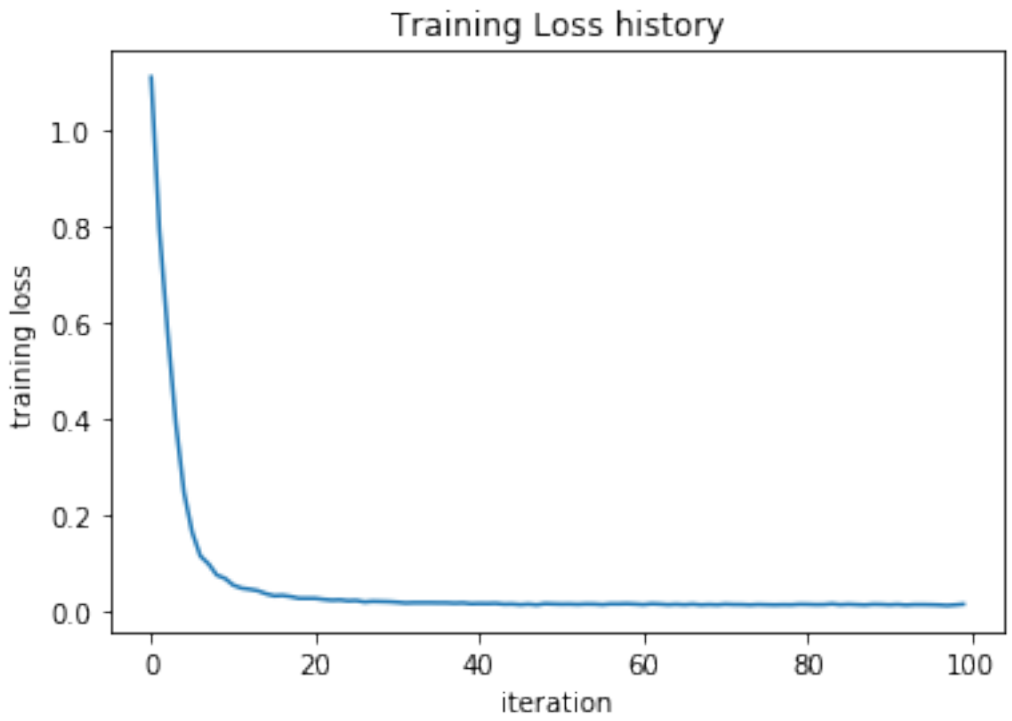
```
W2 max relative error: 2.9632245016399034e-10
b2 max relative error: 1.8392106647421603e-10
W1 max relative error: 1.2832892417669998e-09
b1 max relative error: 3.172680285697327e-09
```

0.2.4 Training the network

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

[8]:
```
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:  0.014497865475252903
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[9]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '/Users/stathismegas/Documents/ECE_247/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,)
```

0.3.1 Running SGD

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

[10]:
```
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 2.302120160170799
iteration 200 / 1000: loss 2.2956136025577862
iteration 300 / 1000: loss 2.2518259124925293
iteration 400 / 1000: loss 2.1889952557361934
iteration 500 / 1000: loss 2.1162528247006445
iteration 600 / 1000: loss 2.0646709106774503
iteration 700 / 1000: loss 1.9901688837884741
iteration 800 / 1000: loss 2.0028277559728718
iteration 900 / 1000: loss 1.9465178635514782
Validation accuracy:  0.283
```

0.4 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.

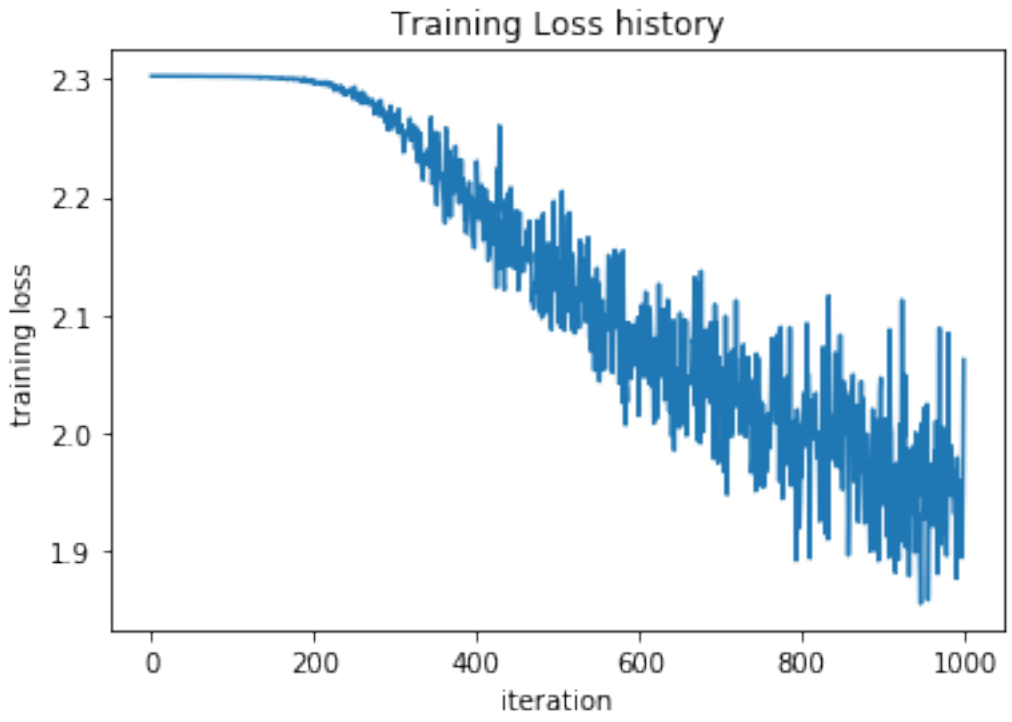
(2) How should you fix the problems you identified in (1)?

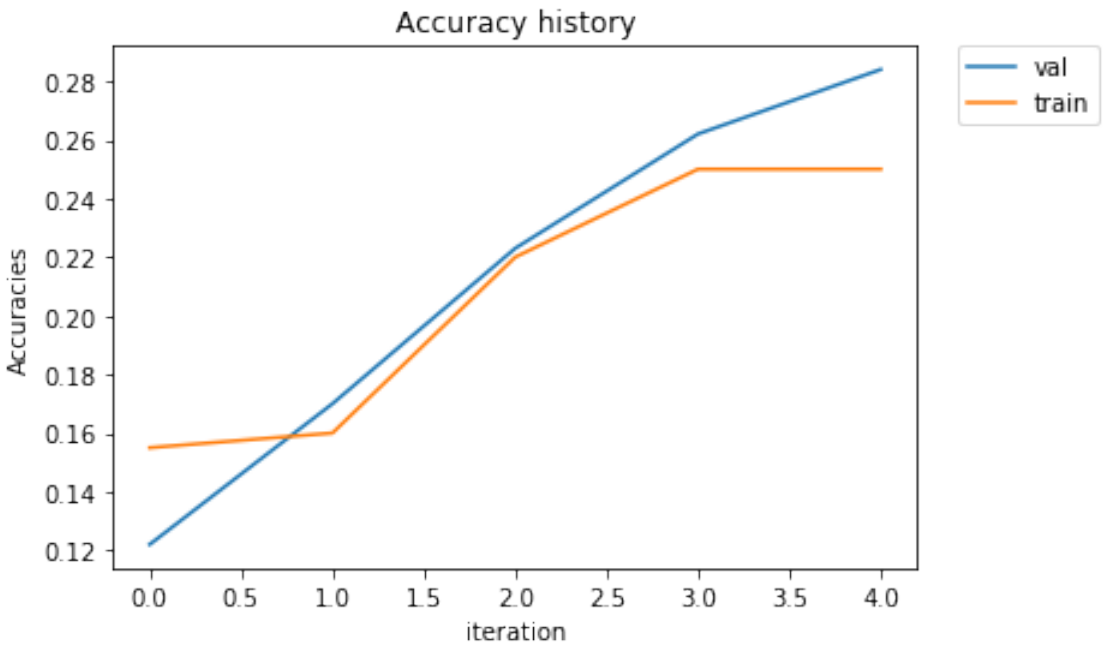
[11]:
```
stats['train_acc_history']
```

[11]:
```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

[12]:
```
# ================================================================ #
# 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 history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
plt.plot( stats['val_acc_history'], label="val")
plt.plot(stats['train_acc_history'], label="train")
plt.xlabel('iteration')
plt.ylabel('Accuracies')
plt.title('Accuracy history')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()

# f = plt.figure()
# ax = f.gca()
# ax.plot(stats['train_acc_history'], stats['val_acc_history'], '-o')
# ax.set_xlabel('$iteration$')
# ax.set_ylabel('$accuracy$')

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





0.5 Answers:

(1) We see that the loss function is very noisy with a mean which is roughly a straight line after iteration 300. These two features are a hint that the learning rate is small! We would want the initial exponential decay of the loss to persist longer so we need to increase the learning rate. Likely it also means we should adjust the number of iterations and the learning_rate_decay. Moreover, we see that the accuracies keep increasing and they haven't saturated. Yet both accuracies are so low that it is clear that the learning rate, number of iterations and learning rate decay need adjustment.

(2) The way to perform hyperparameter tuning is to train the model of the train data for different specifications of the hyperparameters, then calculate the validation accuracy, and finally pick the hyperparamters that had the maximum validation accuracy. We do this below.

0.6 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.

[30]:
```
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)!
# ================================================================ #
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

rates = [ 9e-4, 1e-3, 1.1e-3, 1.2e-3, 1.4e-3, 1.5e-3, 1.6e-3, 1.8e-3, 2e-3␣
↪,1e-4]
decays = [0.91, 0.92, 0.93]
best=0
best_rate=0
best_decay=0

for decay in decays:
    for rate in rates:
        # Train the network
        net = TwoLayerNet(input_size, hidden_size, num_classes)
        stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=rate, learning_rate_decay=decay,
            reg=0.25, verbose=True)

        # Predict on the validation set
        val_acc = (net.predict(X_val) == y_val).mean()
        print('Validation accuracy: ', val_acc, 'for learning rate=', rate,'and␣
↪decay rate=', decay)

        if val_acc>best:
            best = val_acc
            best_net = net
            best_rate = rate
            best_decay = decay


print('best validation accuracy ', best, 'achieved for learning rate',␣
↪best_rate, 'and decay', best_decay)


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

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

```
iteration 0 / 1000: loss 2.3027526263458826
iteration 100 / 1000: loss 2.037422584023342
iteration 200 / 1000: loss 1.8063172829654737
iteration 300 / 1000: loss 1.7266531403359007
iteration 400 / 1000: loss 1.7117307632715548
iteration 500 / 1000: loss 1.6523430829766477
iteration 600 / 1000: loss 1.6176053001213357
iteration 700 / 1000: loss 1.5520675504820995
iteration 800 / 1000: loss 1.473717380809065
iteration 900 / 1000: loss 1.4987916655599491
Validation accuracy:  0.455 for learning rate= 0.0009 and decay rate= 0.91
iteration 0 / 1000: loss 2.302764747712296
iteration 100 / 1000: loss 2.0551035766260437
iteration 200 / 1000: loss 1.805449265167068
iteration 300 / 1000: loss 1.7158471167141658
iteration 400 / 1000: loss 1.668332818026018
iteration 500 / 1000: loss 1.5329187156511352
iteration 600 / 1000: loss 1.5529101232342561
iteration 700 / 1000: loss 1.4913610314197687
iteration 800 / 1000: loss 1.5817636881214792
iteration 900 / 1000: loss 1.5279816895403542
Validation accuracy:  0.461 for learning rate= 0.001 and decay rate= 0.91
iteration 0 / 1000: loss 2.302766852272655
iteration 100 / 1000: loss 1.855235840033207
iteration 200 / 1000: loss 1.8675844283360492
iteration 300 / 1000: loss 1.7178357306705292
iteration 400 / 1000: loss 1.6437015113766227
iteration 500 / 1000: loss 1.4671104184796173
iteration 600 / 1000: loss 1.6267537139398216
iteration 700 / 1000: loss 1.5753304871589
iteration 800 / 1000: loss 1.5612237028478564
iteration 900 / 1000: loss 1.5764734165392968
Validation accuracy:  0.465 for learning rate= 0.0011 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027764598997176
iteration 100 / 1000: loss 1.9626617441087344
iteration 200 / 1000: loss 1.7642765066552624
iteration 300 / 1000: loss 1.690622408728896
iteration 400 / 1000: loss 1.6502610745165731
iteration 500 / 1000: loss 1.4860774448938399
iteration 600 / 1000: loss 1.4888684224217603
iteration 700 / 1000: loss 1.5057732642153925
iteration 800 / 1000: loss 1.6079413630666022
iteration 900 / 1000: loss 1.5482142684063374
Validation accuracy:  0.483 for learning rate= 0.0012 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027811931269064
```

```
iteration 100 / 1000: loss 1.9082413493381
iteration 200 / 1000: loss 1.6902834536098321
iteration 300 / 1000: loss 1.4995754285038871
iteration 400 / 1000: loss 1.6765854657150168
iteration 500 / 1000: loss 1.5754334182647751
iteration 600 / 1000: loss 1.6382714697352592
iteration 700 / 1000: loss 1.5389399558512489
iteration 800 / 1000: loss 1.4386463455227754
iteration 900 / 1000: loss 1.3965520887781604
Validation accuracy:  0.476 for learning rate= 0.0014 and decay rate= 0.91
iteration 0 / 1000: loss 2.302779184717079
iteration 100 / 1000: loss 1.8741982028033704
iteration 200 / 1000: loss 1.7245724788786345
iteration 300 / 1000: loss 1.5620967493973503
iteration 400 / 1000: loss 1.6136589336123084
iteration 500 / 1000: loss 1.6406302083322952
iteration 600 / 1000: loss 1.507580058353166
iteration 700 / 1000: loss 1.604260452416418
iteration 800 / 1000: loss 1.5319333393445522
iteration 900 / 1000: loss 1.564120763222298
Validation accuracy:  0.464 for learning rate= 0.0015 and decay rate= 0.91
iteration 0 / 1000: loss 2.30280608437208
iteration 100 / 1000: loss 1.844347668160187
iteration 200 / 1000: loss 1.6693091159577698
iteration 300 / 1000: loss 1.6417388097343981
iteration 400 / 1000: loss 1.6450387854298085
iteration 500 / 1000: loss 1.4311272798917978
iteration 600 / 1000: loss 1.446066499037622
iteration 700 / 1000: loss 1.4438185708296873
iteration 800 / 1000: loss 1.473262465633535
iteration 900 / 1000: loss 1.3780585344469705
Validation accuracy:  0.489 for learning rate= 0.0016 and decay rate= 0.91
iteration 0 / 1000: loss 2.302750861249898
iteration 100 / 1000: loss 1.819096005456171
iteration 200 / 1000: loss 1.6986503091863674
iteration 300 / 1000: loss 1.6575175319812596
iteration 400 / 1000: loss 1.7330169264919373
iteration 500 / 1000: loss 1.407538264143662
iteration 600 / 1000: loss 1.449062675963313
iteration 700 / 1000: loss 1.4881309966281373
iteration 800 / 1000: loss 1.4852235479224032
iteration 900 / 1000: loss 1.5832232839697862
Validation accuracy:  0.472 for learning rate= 0.0018 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027764803930637
iteration 100 / 1000: loss 1.7666360596126112
iteration 200 / 1000: loss 1.6661962752947854
iteration 300 / 1000: loss 1.6999386351179402
iteration 400 / 1000: loss 1.5768473862582923
```

```
iteration 500 / 1000: loss 1.5625661897692444
iteration 600 / 1000: loss 1.731500669467699
iteration 700 / 1000: loss 1.5758523547277488
iteration 800 / 1000: loss 1.5080668710421075
iteration 900 / 1000: loss 1.3670808491281907
Validation accuracy:  0.475 for learning rate= 0.002 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027886031861735
iteration 100 / 1000: loss 2.302533871368175
iteration 200 / 1000: loss 2.2995890510324717
iteration 300 / 1000: loss 2.274271170412578
iteration 400 / 1000: loss 2.25020365022876
iteration 500 / 1000: loss 2.2002144505112735
iteration 600 / 1000: loss 2.134826036920963
iteration 700 / 1000: loss 2.089515609506605
iteration 800 / 1000: loss 1.9707954594611419
iteration 900 / 1000: loss 2.030086299887209
Validation accuracy:  0.274 for learning rate= 0.0001 and decay rate= 0.91
iteration 0 / 1000: loss 2.302765225758387
iteration 100 / 1000: loss 2.0456681259091605
iteration 200 / 1000: loss 1.786443653581439
iteration 300 / 1000: loss 1.6945349592896
iteration 400 / 1000: loss 1.6538275805435
iteration 500 / 1000: loss 1.632368840740618
iteration 600 / 1000: loss 1.5050096224608955
iteration 700 / 1000: loss 1.5423219198786622
iteration 800 / 1000: loss 1.556037673841548
iteration 900 / 1000: loss 1.4199593816582998
Validation accuracy:  0.457 for learning rate= 0.0009 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027875440096817
iteration 100 / 1000: loss 1.893192465425405
iteration 200 / 1000: loss 1.7166711303251012
iteration 300 / 1000: loss 1.7479828400383337
iteration 400 / 1000: loss 1.5818909354310782
iteration 500 / 1000: loss 1.533252403371238
iteration 600 / 1000: loss 1.6675537965822993
iteration 700 / 1000: loss 1.4628574994586232
iteration 800 / 1000: loss 1.5234503808913165
iteration 900 / 1000: loss 1.4923782729519934
Validation accuracy:  0.488 for learning rate= 0.001 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027879282421972
iteration 100 / 1000: loss 1.8347423406391759
iteration 200 / 1000: loss 1.8311958227659326
iteration 300 / 1000: loss 1.572280395078183
iteration 400 / 1000: loss 1.7540775583742172
iteration 500 / 1000: loss 1.475113698667047
iteration 600 / 1000: loss 1.7005921772815111
iteration 700 / 1000: loss 1.568302438100576
iteration 800 / 1000: loss 1.4255591861318369
```

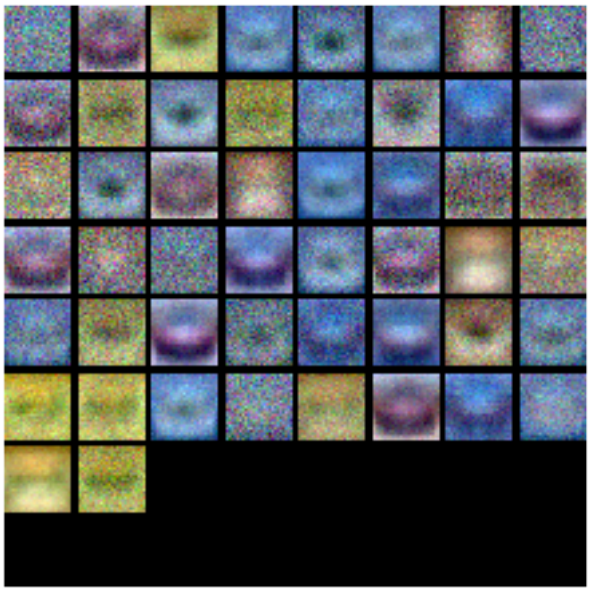
```
iteration 900 / 1000: loss 1.4214717842026463
Validation accuracy:  0.481 for learning rate= 0.0011 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027877966236256
iteration 100 / 1000: loss 1.9025250585142668
iteration 200 / 1000: loss 1.7845374628555084
iteration 300 / 1000: loss 1.7891133134763677
iteration 400 / 1000: loss 1.6250452612311737
iteration 500 / 1000: loss 1.5456332798448067
iteration 600 / 1000: loss 1.5794498909274866
iteration 700 / 1000: loss 1.5152706977299364
iteration 800 / 1000: loss 1.525523299029692
iteration 900 / 1000: loss 1.3821067009338521
Validation accuracy:  0.48 for learning rate= 0.0012 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027821195137346
iteration 100 / 1000: loss 1.815597285691698
iteration 200 / 1000: loss 1.7991268879709077
iteration 300 / 1000: loss 1.666103974818494
iteration 400 / 1000: loss 1.6905994971634697
iteration 500 / 1000: loss 1.4756411125111968
iteration 600 / 1000: loss 1.5687947634261603
iteration 700 / 1000: loss 1.4782461627557029
iteration 800 / 1000: loss 1.547258877718856
iteration 900 / 1000: loss 1.5621352272609677
Validation accuracy:  0.472 for learning rate= 0.0014 and decay rate= 0.92
iteration 0 / 1000: loss 2.302782568036341
iteration 100 / 1000: loss 1.835301814385985
iteration 200 / 1000: loss 1.6799214174151353
iteration 300 / 1000: loss 1.6160132102638458
iteration 400 / 1000: loss 1.7358092682569097
iteration 500 / 1000: loss 1.552820431337074
iteration 600 / 1000: loss 1.6005060273751806
iteration 700 / 1000: loss 1.5376882804793943
iteration 800 / 1000: loss 1.5762845037532875
iteration 900 / 1000: loss 1.4910685469870995
Validation accuracy:  0.461 for learning rate= 0.0015 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027620957625845
iteration 100 / 1000: loss 1.8185433142811709
iteration 200 / 1000: loss 1.6663917008525475
iteration 300 / 1000: loss 1.5054221375292076
iteration 400 / 1000: loss 1.5822511273494242
iteration 500 / 1000: loss 1.5361872019104705
iteration 600 / 1000: loss 1.5984936787144204
iteration 700 / 1000: loss 1.5001788008846235
iteration 800 / 1000: loss 1.5750661777506691
iteration 900 / 1000: loss 1.5304911021803427
Validation accuracy:  0.488 for learning rate= 0.0016 and decay rate= 0.92
iteration 0 / 1000: loss 2.30279132805707
iteration 100 / 1000: loss 1.8022569025215653
```

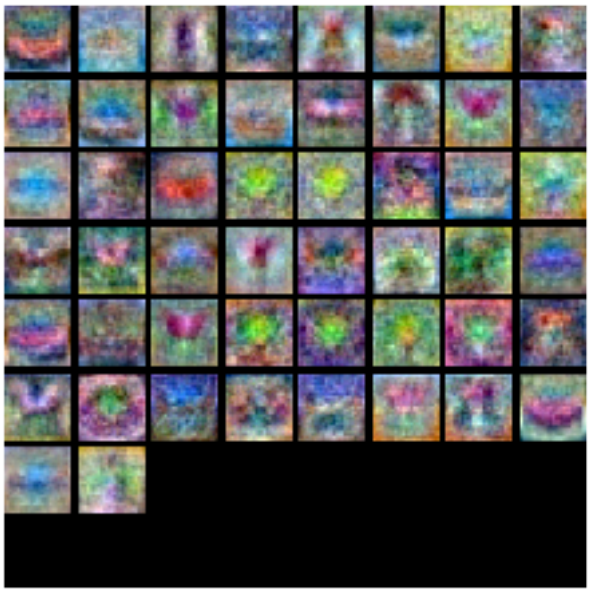
```
iteration 200 / 1000: loss 1.7573959921805067
iteration 300 / 1000: loss 1.6555398440790572
iteration 400 / 1000: loss 1.685691605374468
iteration 500 / 1000: loss 1.551463633368692
iteration 600 / 1000: loss 1.471125429922446
iteration 700 / 1000: loss 1.5437237629860001
iteration 800 / 1000: loss 1.4709821741433275
iteration 900 / 1000: loss 1.6465870701008538
Validation accuracy:  0.46 for learning rate= 0.0018 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027653782483903
iteration 100 / 1000: loss 1.7354278700991306
iteration 200 / 1000: loss 1.692773437371705
iteration 300 / 1000: loss 1.6308499124637141
iteration 400 / 1000: loss 1.6799614056389043
iteration 500 / 1000: loss 1.4751466940373366
iteration 600 / 1000: loss 1.7528690158745446
iteration 700 / 1000: loss 1.571636255433799
iteration 800 / 1000: loss 1.4913193076122098
iteration 900 / 1000: loss 1.505855628654246
Validation accuracy:  0.449 for learning rate= 0.002 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027608591890325
iteration 100 / 1000: loss 2.302305779563855
iteration 200 / 1000: loss 2.2954228136909407
iteration 300 / 1000: loss 2.248648408026622
iteration 400 / 1000: loss 2.223361274912419
iteration 500 / 1000: loss 2.169656954955473
iteration 600 / 1000: loss 2.0937973509531065
iteration 700 / 1000: loss 2.0661813765817767
iteration 800 / 1000: loss 2.0725727880403833
iteration 900 / 1000: loss 1.9881720834217709
Validation accuracy:  0.273 for learning rate= 0.0001 and decay rate= 0.92
iteration 0 / 1000: loss 2.302799524865559
iteration 100 / 1000: loss 1.9940750813606323
iteration 200 / 1000: loss 1.7211028884783817
iteration 300 / 1000: loss 1.699737404568344
iteration 400 / 1000: loss 1.7850021763742054
iteration 500 / 1000: loss 1.6608943321713947
iteration 600 / 1000: loss 1.7556723024018612
iteration 700 / 1000: loss 1.6797959089158714
iteration 800 / 1000: loss 1.6125549391236447
iteration 900 / 1000: loss 1.4418816636118525
Validation accuracy:  0.471 for learning rate= 0.0009 and decay rate= 0.93
iteration 0 / 1000: loss 2.3028089084282795
iteration 100 / 1000: loss 1.9578561389940095
iteration 200 / 1000: loss 1.7532891891711948
iteration 300 / 1000: loss 1.6283618901230952
iteration 400 / 1000: loss 1.6670149341486005
iteration 500 / 1000: loss 1.531887080385329
```

```
iteration 600 / 1000: loss 1.5401651930568854
iteration 700 / 1000: loss 1.447107094577251
iteration 800 / 1000: loss 1.6555877622214972
iteration 900 / 1000: loss 1.6356135852796474
Validation accuracy:  0.471 for learning rate= 0.001 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027914774734963
iteration 100 / 1000: loss 1.9134514974450185
iteration 200 / 1000: loss 1.8190576140026622
iteration 300 / 1000: loss 1.6278760988062977
iteration 400 / 1000: loss 1.4848310488246041
iteration 500 / 1000: loss 1.6039628321171158
iteration 600 / 1000: loss 1.3854227668578198
iteration 700 / 1000: loss 1.3951224054104614
iteration 800 / 1000: loss 1.6405236444994489
iteration 900 / 1000: loss 1.3930023110969063
Validation accuracy:  0.478 for learning rate= 0.0011 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027834683215183
iteration 100 / 1000: loss 1.9976130905594969
iteration 200 / 1000: loss 1.6972729286003418
iteration 300 / 1000: loss 1.7222532972312177
iteration 400 / 1000: loss 1.5929367696983607
iteration 500 / 1000: loss 1.5402579996768127
iteration 600 / 1000: loss 1.5275985210170882
iteration 700 / 1000: loss 1.5595503840206457
iteration 800 / 1000: loss 1.5384125115052179
iteration 900 / 1000: loss 1.4632347069188691
Validation accuracy:  0.479 for learning rate= 0.0012 and decay rate= 0.93
iteration 0 / 1000: loss 2.302774112771274
iteration 100 / 1000: loss 1.913518883359934
iteration 200 / 1000: loss 1.7662738696946416
iteration 300 / 1000: loss 1.6143280430411133
iteration 400 / 1000: loss 1.5172801735235812
iteration 500 / 1000: loss 1.5814246969408219
iteration 600 / 1000: loss 1.630279773696954
iteration 700 / 1000: loss 1.4597148063165972
iteration 800 / 1000: loss 1.4185925677272049
iteration 900 / 1000: loss 1.5220000988029048
Validation accuracy:  0.458 for learning rate= 0.0014 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027759664938667
iteration 100 / 1000: loss 1.8502196236758943
iteration 200 / 1000: loss 1.6561324366184307
iteration 300 / 1000: loss 1.633037661366044
iteration 400 / 1000: loss 1.5621801150459207
iteration 500 / 1000: loss 1.555079668167577
iteration 600 / 1000: loss 1.442662989002445
iteration 700 / 1000: loss 1.5436268160716338
iteration 800 / 1000: loss 1.6140923532723623
iteration 900 / 1000: loss 1.6485802536415806
```

```
Validation accuracy:  0.486 for learning rate= 0.0015 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027831931343314
iteration 100 / 1000: loss 1.8799633369913613
iteration 200 / 1000: loss 1.7437969641831665
iteration 300 / 1000: loss 1.5934710322973362
iteration 400 / 1000: loss 1.643125191684595
iteration 500 / 1000: loss 1.4741327505926718
iteration 600 / 1000: loss 1.6807849502063543
iteration 700 / 1000: loss 1.5080744714033936
iteration 800 / 1000: loss 1.4904408322730713
iteration 900 / 1000: loss 1.5331369427690242
Validation accuracy:  0.502 for learning rate= 0.0016 and decay rate= 0.93
iteration 0 / 1000: loss 2.302776308975928
iteration 100 / 1000: loss 1.8386854484027673
iteration 200 / 1000: loss 1.7380583349812813
iteration 300 / 1000: loss 1.6575012337082617
iteration 400 / 1000: loss 1.5826972138729605
iteration 500 / 1000: loss 1.5097910113000492
iteration 600 / 1000: loss 1.422820614007086
iteration 700 / 1000: loss 1.6510415941271193
iteration 800 / 1000: loss 1.5023397592744636
iteration 900 / 1000: loss 1.4820162337293827
Validation accuracy:  0.466 for learning rate= 0.0018 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027580897821105
iteration 100 / 1000: loss 1.8134832652077677
iteration 200 / 1000: loss 1.673572618171472
iteration 300 / 1000: loss 1.514317595263018
iteration 400 / 1000: loss 1.6859639083136124
iteration 500 / 1000: loss 1.4915345097951742
iteration 600 / 1000: loss 1.4577402104140278
iteration 700 / 1000: loss 1.680291860404149
iteration 800 / 1000: loss 1.7381266018848855
iteration 900 / 1000: loss 1.547767205001409
Validation accuracy:  0.451 for learning rate= 0.002 and decay rate= 0.93
iteration 0 / 1000: loss 2.3028006912047427
iteration 100 / 1000: loss 2.3024373819498543
iteration 200 / 1000: loss 2.3004167287282278
iteration 300 / 1000: loss 2.280025715281056
iteration 400 / 1000: loss 2.207597840718256
iteration 500 / 1000: loss 2.1105841798573826
iteration 600 / 1000: loss 2.1408763274717715
iteration 700 / 1000: loss 2.0620071802265523
iteration 800 / 1000: loss 2.057689566109094
iteration 900 / 1000: loss 2.0125096320131033
Validation accuracy:  0.277 for learning rate= 0.0001 and decay rate= 0.93
best validation accuracy  0.502 achieved for learning rate 0.0016 and decay 0.93
```

[31]:
```
from cs231n.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)
```





0.7 Question:

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

0.8 Answer:

(1) It is obvious that the subopt net has figured out some rough template features of classes. Eg we roughly see the outline of a car in some entries of the grip plot. Whoever, the features detectd by teh subopt are so rough that they could only lead to a bad performance. On the contrary, in the best network achieved, we see that the template features detected are much more detailed, and naturally best_net is able to classify photos much more accurately.

0.9 Evaluate on test set

[34]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.469
```

[ ]:

[ ]:

layers.py

Untitled1

February 3, 2021

[ ]:
```
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""


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.
  # ================================================================ #

  N = np.shape(x)[0]
  x_flat = np.reshape(x, (N,-1))
  out = np.dot( x_flat , w ) + b    # N,M = N,D * D,M

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

  cache = (x, w, b)
  return out, cache


def affine_backward(dout, cache):
  """
  Computes the backward pass for an affine layer.

  Inputs:
  - dout: Upstream derivative, of shape (N, M)
  - cache: Tuple of:
    - x: Input data, of shape (N, d_1, ... d_k)
    - w: Weights, of shape (D, M)

  Returns a tuple of:
  - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
  - dw: Gradient with respect to w, of shape (D, M)
  - db: Gradient with respect to b, of shape (M,)
  """
  x, w, b = cache
  dx, dw, db = None, None, None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Calculate the gradients for the backward pass.
  # ================================================================ #

  # dout is N x M
  # dx should be N x d1 x ... x dk; it relates to dout through multiplication␣
↪with w, which is D x M
  # dw should be D x M; it relates to dout through multiplication with x, which␣
↪is N x D after reshaping
  # db should be M; it is just the sum over dout examples

  N = np.shape(x)[0]
```

```
  shaping = np.shape(x)
  x_flat = np.reshape(x, (N,-1))

  dx = np.reshape(np.dot( dout , np.transpose(w) ),   shaping )
  dw = np.dot(x_flat.T, dout)
  db = np.sum(dout, axis=0, keepdims=True)


  # ================================================================ #
  # 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(0 , x)

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

  cache = x
  return out, cache


def relu_backward(dout, cache):
  """
  Computes the backward pass for a layer of rectified linear units (ReLUs).

  Input:
  - dout: Upstream derivatives, of any shape
  - cache: Input x, of same shape as dout
```

```
  Returns:
  - dx: Gradient with respect to x
  """
  x = cache

  # ============================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU backward pass
  # ============================================================ #

  # ReLU directs linearly to those > 0
  dx = dout
  #print('from inside the relu_back, the input dim is', np.shape(dout),'and the output dim is', np.shape(dx))
  dx[x <= 0] = 0

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

  return dx

def svm_loss(x, y):
  """
  Computes the loss and gradient using for multiclass SVM classification.

  Inputs:
  - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """
  N = x.shape[0]
  correct_class_scores = x[np.arange(N), y]
  margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
  margins[np.arange(N), y] = 0
  loss = np.sum(margins) / N
  num_pos = np.sum(margins > 0, axis=1)
  dx = np.zeros_like(x)
  dx[margins > 0] = 1
  dx[np.arange(N), y] -= num_pos
  dx /= N
```

```
  return loss, dx


def softmax_loss(x, y):
  """
  Computes the loss and gradient for softmax classification.

  Inputs:
  - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """

  probs = np.exp(x - np.max(x, axis=1, keepdims=True))
  probs /= np.sum(probs, axis=1, keepdims=True)
  N = x.shape[0]
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
  dx = probs.copy()
  dx[np.arange(N), y] -= 1
  dx /= N
  return loss, dx
```

neural_net

Untitled1

February 3, 2021

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, 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.
    """
    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

    # 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.
    # ================================================================ #

    hidden1 = np.dot( X , np.transpose(W1) ) + b1
    relu_scores = np.maximum(0, hidden1 )
    hidden2 = np.dot( relu_scores , np.transpose(W2) ) + b2
    scores = hidden2


    # ================================================================ #
    # 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 the variable loss.  Multiply the regularization
        #         loss by 0.5 (in addition to the factor reg).
        # ================================================================ #

    # scores is num_examples by num_classes
    #pass
    expon = np.exp( scores )

    vec1 = expon.sum(axis=1)
    vec1_log = np.log(vec1)
    vec2 = scores[  np.arange( np.shape(X)[0] ), y]
    vec_final = vec1_log - vec2
    loss = np.sum(vec_final)
    loss /= N

    reg_loss = 0.5 * reg * ( np.sum(W1 * W1) + np.sum(W2 * W2) )
```

```
    loss += reg_loss




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

    grads = {}

    # ============================================================= #
    # YOUR CODE HERE:
        #         Implement the backward pass.  Compute the derivatives of the
        #         weights and the biases.  Store the results in the grads
        #        dictionary.  e.g., grads['W1'] should store the gradient for
        #         W1, and be of the same size as W1.
        # ============================================================= #


    prefac =  np.transpose( np.transpose(expon) / vec1)   # has dimension N,C

    prefac[ np.arange(np.shape(X)[0]), y] -= 1
    grads['W2'] = np.dot(np.transpose(prefac), relu_scores)
    grads['W2'] /= np.shape(X)[0]
    grads['W2'] += reg* W2

    grads['b2'] = np.dot(np.transpose(prefac), np.ones( N )  )
    grads['b2'] /= np.shape(X)[0]
    grads['b2'] += reg* b2

    hidden_prefac = np.dot( prefac , W2 )    #should have dim N,H
    hidden_prefac[hidden1 <= 0] = 0


    grads['W1'] = np.dot(np.transpose(hidden_prefac), X)   #should have dim H,D
    grads['W1'] /= np.shape(X)[0]
    grads['W1'] += reg* W1

    grads['b1'] = np.dot(np.transpose(hidden_prefac), np.ones( N )  )
    grads['b1'] /= np.shape(X)[0]
    grads['b1'] += reg* b1



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

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

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in np.arange(num_iters):
        X_batch = None
        y_batch = None
        rand_ind = np.random.choice( np.shape(X)[0] , batch_size )


      # ================================================================ #
      # YOUR CODE HERE:
            #         Create a minibatch by sampling batch_size samples␣
↪randomly.
            # ================================================================ #
        X_batch = X[rand_ind, :]
        y_batch = y[rand_ind]
```

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

       # Compute loss and gradients using the current minibatch
        loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
        loss_history.append(loss)

      # ================================================================ #
      # YOUR CODE HERE:
            #         Perform a gradient descent step using the minibatch to␣
↪update
           #         all parameters (i.e., W1, W2, b1, and b2).
             # ================================================================ #

        self.params['W1'] += -learning_rate * grads['W1']
        self.params['b1'] += -learning_rate * grads['b1']
        self.params['W2'] += -learning_rate * grads['W2']
        self.params['b2'] += -learning_rate * grads['b2']

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

        if verbose and it % 100 == 0:
            print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

      # Every epoch, check train and val accuracy and decay learning rate.
        if it % iterations_per_epoch == 0:
            # Check accuracy
            train_acc = (self.predict(X_batch) == y_batch).mean()
            val_acc = (self.predict(X_val) == y_val).mean()
            train_acc_history.append(train_acc)
            val_acc_history.append(val_acc)

            # Decay learning rate
            learning_rate *= learning_rate_decay

    return {
      'loss_history': loss_history,
      'train_acc_history': train_acc_history,
      'val_acc_history': val_acc_history,
    }

  def predict(self, X):
    """
```

```
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
      to have class c, where 0 <= c < C.
    """
    y_pred = None

    # ========================================================== #
    # YOUR CODE HERE:
    #         Predict the class given the input data.
    # ========================================================== #
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']

    hidden1 = np.dot( X , np.transpose(W1) ) + b1
    relu_scores = np.maximum(0, hidden1 )
    hidden2 = np.dot( relu_scores , np.transpose(W2) ) + b2
    scores = hidden2   #has dimension N,C
    y_pred = np.argmax( scores , axis=1)




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

    return y_pred
```

FC_nets

January 31, 2021

1 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).

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, and their layer structure. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

1.1 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 [73]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_ar
from cs231n.solver import Solver

import os
#alias kk os._exit(0)

%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 [7]:
```
# 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,)
```

1.2 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.

1.2.1 Affine layer forward pass

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

In [15]:
```
# 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
```

1.2.2 Affine layer backward pass

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

In [18]:
```
# 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: 1.8652421687926941e-10
dw error: 1.1954598341802656e-10
db error: 5.469309841204346e-12
```

1.3 Activation layers

In this section you'll implement the ReLU activation.

1.3.1 ReLU forward pass

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

In [22]:
```
# 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
```

1.3.2 ReLU backward pass

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

```
In [23]: 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: 3.2756100968675378e-12
```

1.4 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`.

1.4.1 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 [24]: 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, d
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, d
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, d

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: 2.409121803285706e-11
dw error: 2.5588677013797947e-09
db error: 1.892888397293258e-11
```

1.5 Softmax and SVM losses

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

In [25]:
```
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: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print('Testing svm_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))

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 svm_loss:
loss: 9.000230416932016
dx error: 8.182894472887002e-10

Testing softmax_loss:
loss: 2.302608600826777
dx error: 7.260416088699731e-09
```

1.6 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 [53]:
```
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: 2.131611955458401e-08
W2 relative error: 3.310270199776237e-10
b1 relative error: 8.36819673247588e-09
b2 relative error: 2.530774050159566e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.5279153413239097e-07
W2 relative error: 2.8508696990815807e-08
b1 relative error: 1.5646802033932055e-08
b2 relative error: 9.089614638133234e-10
```

1.7 Solver

We will now use the cs231n Solver class to train these networks. Familiarize yourself with the API in `cs231n/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 40%.

```
In [70]: 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 40%.  We won't have you optimize this further
         #   since you did it in the previous notebook.
         #
         # ============================================================ #



         # data = {
         #       'X_train': X_train
         #       'y_train': y_train
         #       'X_val': X_val
```

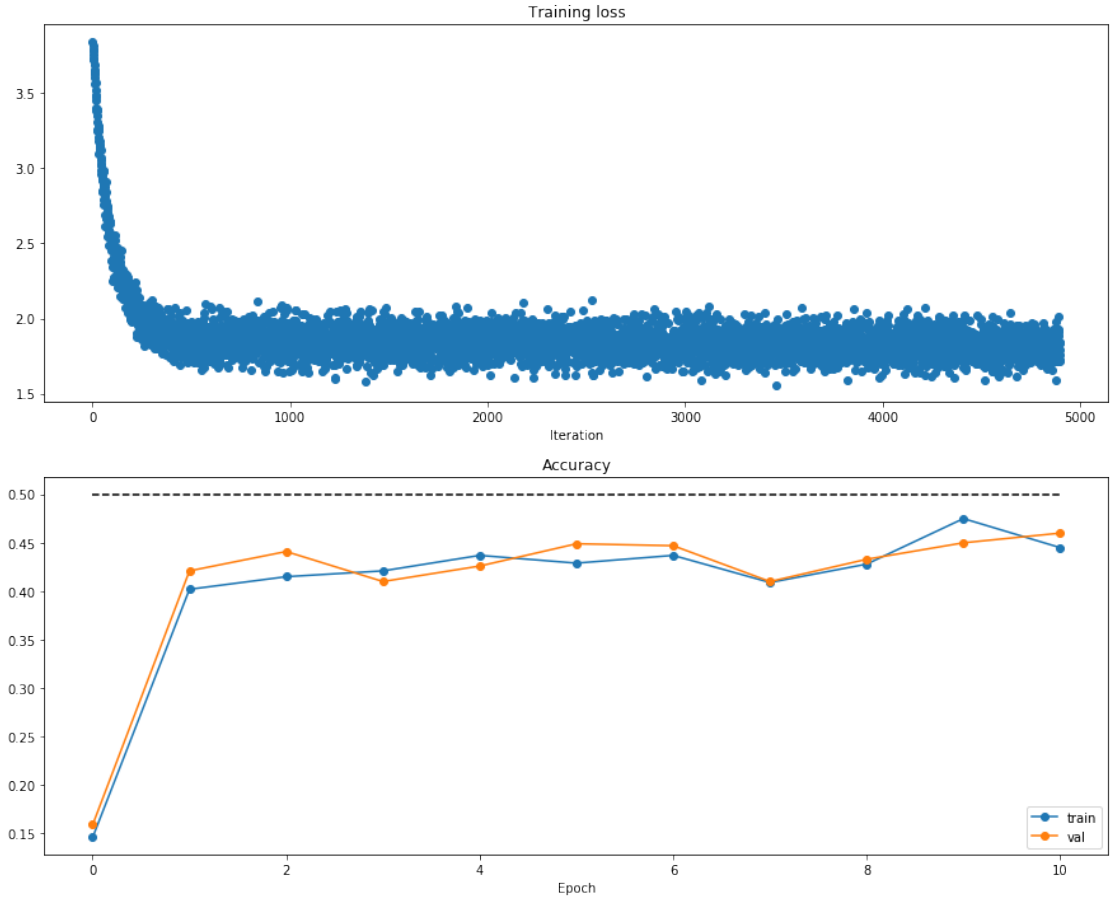
```
        #       'y_val': y_val
        #     }

        model = TwoLayerNet( hidden_dims=200, num_classes=10, reg=5)

        solver = Solver(model, data,
                        update_rule='sgd',
                        optim_config={
                          'learning_rate': 1e-3,
                        },
                        lr_decay=0.95,
                        num_epochs=10, batch_size=100,
                        print_every=100)

        solver.train()

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

```
(Iteration 1 / 4900) loss: 3.840819
(Epoch 0 / 10) train acc: 0.146000; val_acc: 0.159000
(Iteration 101 / 4900) loss: 2.533802
(Iteration 201 / 4900) loss: 2.156810
(Iteration 301 / 4900) loss: 1.982050
(Iteration 401 / 4900) loss: 1.735392
(Epoch 1 / 10) train acc: 0.402000; val_acc: 0.421000
(Iteration 501 / 4900) loss: 1.859601
(Iteration 601 / 4900) loss: 1.848238
(Iteration 701 / 4900) loss: 1.894866
(Iteration 801 / 4900) loss: 1.979157
(Iteration 901 / 4900) loss: 1.839227
(Epoch 2 / 10) train acc: 0.415000; val_acc: 0.441000
(Iteration 1001 / 4900) loss: 1.912778
(Iteration 1101 / 4900) loss: 1.863958
(Iteration 1201 / 4900) loss: 1.882777
(Iteration 1301 / 4900) loss: 1.810530
(Iteration 1401 / 4900) loss: 1.947189
(Epoch 3 / 10) train acc: 0.421000; val_acc: 0.410000
(Iteration 1501 / 4900) loss: 1.787672
(Iteration 1601 / 4900) loss: 1.930837
(Iteration 1701 / 4900) loss: 1.974351
(Iteration 1801 / 4900) loss: 1.727694
(Iteration 1901 / 4900) loss: 1.855581
(Epoch 4 / 10) train acc: 0.437000; val_acc: 0.426000
(Iteration 2001 / 4900) loss: 1.864591
(Iteration 2101 / 4900) loss: 1.895908
(Iteration 2201 / 4900) loss: 1.901670
```

```
(Iteration 2301 / 4900) loss: 2.061896
(Iteration 2401 / 4900) loss: 1.868035
(Epoch 5 / 10) train acc: 0.429000; val_acc: 0.449000
(Iteration 2501 / 4900) loss: 1.867693
(Iteration 2601 / 4900) loss: 1.842856
(Iteration 2701 / 4900) loss: 1.843426
(Iteration 2801 / 4900) loss: 2.024520
(Iteration 2901 / 4900) loss: 1.878428
(Epoch 6 / 10) train acc: 0.437000; val_acc: 0.447000
(Iteration 3001 / 4900) loss: 1.784270
(Iteration 3101 / 4900) loss: 1.768215
(Iteration 3201 / 4900) loss: 1.837510
(Iteration 3301 / 4900) loss: 2.032792
(Iteration 3401 / 4900) loss: 1.762165
(Epoch 7 / 10) train acc: 0.409000; val_acc: 0.410000
(Iteration 3501 / 4900) loss: 1.794370
(Iteration 3601 / 4900) loss: 1.800006
(Iteration 3701 / 4900) loss: 1.838808
(Iteration 3801 / 4900) loss: 1.751549
(Iteration 3901 / 4900) loss: 1.816655
(Epoch 8 / 10) train acc: 0.428000; val_acc: 0.433000
(Iteration 4001 / 4900) loss: 1.930700
(Iteration 4101 / 4900) loss: 1.942737
(Iteration 4201 / 4900) loss: 1.733148
(Iteration 4301 / 4900) loss: 1.718822
(Iteration 4401 / 4900) loss: 1.866450
(Epoch 9 / 10) train acc: 0.475000; val_acc: 0.450000
(Iteration 4501 / 4900) loss: 1.895115
(Iteration 4601 / 4900) loss: 1.729397
(Iteration 4701 / 4900) loss: 1.806893
(Iteration 4801 / 4900) loss: 1.844434
(Epoch 10 / 10) train acc: 0.445000; val_acc: 0.460000
```

In [71]:
```
# 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()
```



1.8 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 assignment #4.

```
In [133]: 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)
```

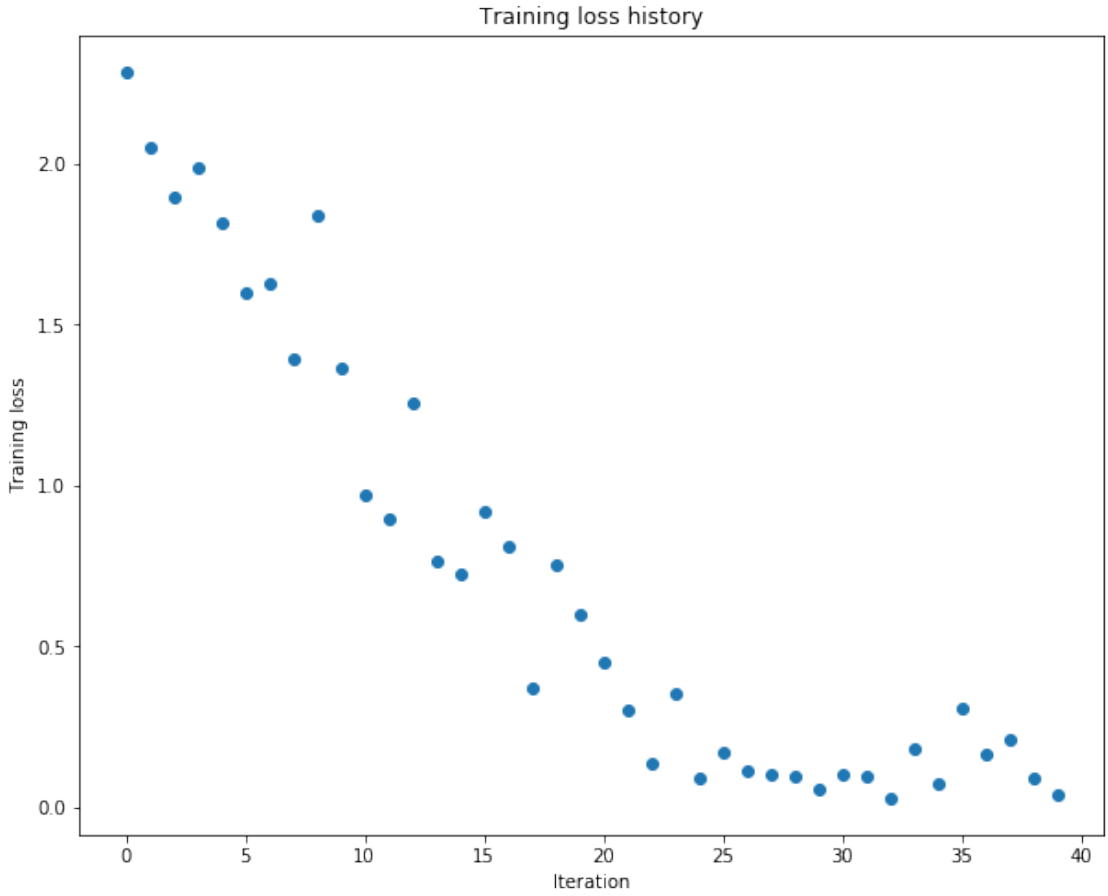
```
  loss, grads = model.loss(X, y)
  print('Initial loss: {}'.format(loss))

  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: 2.304311293062037
W1 relative error: 3.604721819459194e-07
W2 relative error: 1.3969869676949217e-06
W3 relative error: 6.277600478258091e-08
b1 relative error: 1.0673671666702021e-08
b2 relative error: 7.706888918730185e-09
b3 relative error: 1.3635029330757587e-10
Running check with reg = 3.14
Initial loss: 7.284732427349271
W1 relative error: 3.76439960237075e-08
W2 relative error: 3.828501637429764e-08
W3 relative error: 1.1939082334656674e-08
b1 relative error: 2.9087816745529323e-08
b2 relative error: 3.6128665672349e-07
b3 relative error: 1.919495820143335e-10
```

In [139]:
```
# 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   #originally 1e-4

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: 2.288274
(Epoch 0 / 20) train acc: 0.320000; val_acc: 0.113000
(Epoch 1 / 20) train acc: 0.220000; val_acc: 0.103000
(Epoch 2 / 20) train acc: 0.240000; val_acc: 0.105000
(Epoch 3 / 20) train acc: 0.360000; val_acc: 0.115000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.161000
(Epoch 5 / 20) train acc: 0.720000; val_acc: 0.158000
(Iteration 11 / 40) loss: 0.969674
(Epoch 6 / 20) train acc: 0.560000; val_acc: 0.122000
(Epoch 7 / 20) train acc: 0.760000; val_acc: 0.165000
(Epoch 8 / 20) train acc: 0.860000; val_acc: 0.171000
(Epoch 9 / 20) train acc: 0.860000; val_acc: 0.171000
(Epoch 10 / 20) train acc: 0.940000; val_acc: 0.200000
(Iteration 21 / 40) loss: 0.448099
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.183000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.191000
(Epoch 13 / 20) train acc: 0.940000; val_acc: 0.205000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.210000
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.208000
(Iteration 31 / 40) loss: 0.103328
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.215000
(Epoch 17 / 20) train acc: 0.940000; val_acc: 0.187000
(Epoch 18 / 20) train acc: 0.960000; val_acc: 0.214000
(Epoch 19 / 20) train acc: 0.980000; val_acc: 0.185000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.198000
```


Training loss history
Training loss
Iteration
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35
40

fc_net

February 4, 2021

In [ ]:
```
import numpy as np

from .layers import *
from .layer_utils import *

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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.
    """
    self.params = {}
    self.reg = reg

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ============================================================ #
    std = weight_scale
    self.params['W1'] = std * np.random.randn(input_dim, hidden_dims)
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = std * np.random.randn(hidden_dims, num_classes)
    self.params['b2'] = np.zeros(num_classes)

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

  def loss(self, X, y=None):
    """
    Compute loss and gradient for a minibatch of data.

    Inputs:
    - X: Array of input data of shape (N, d_1, ..., d_k)
    - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

    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.
# ============================================================ #

hidden, cache_hid = affine_forward(X, self.params['W1'], self.params['b1'])
relu_scores, cache_relu = relu_forward(hidden)
scores, cache_scores = affine_forward(relu_scores, self.params['W2'], self.params[



# ============================================================ #
# 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, {}
# ============================================================ #
# YOUR CODE HERE:
#   Implement the backward pass of the two-layer neural net.  Store
#   the loss as the variable 'loss' and store the gradients in the
#   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
#   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
#   i.e., grads[k] holds the gradient for self.params[k].
#
#   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
#   for each W.  Be sure to include the 0.5 multiplying factor to
#   match our implementation.
#
#   And be sure to use the layers you prior implemented.
# ============================================================ #

N = np.shape(X)[0]

loss, dscores = softmax_loss(scores,y)
loss += 0.5*self.reg*np.sum(self.params['W1']*self.params['W1']) + 0.5*self.reg*np

dx2, dw2, db2 = affine_backward(dscores, cache_scores)
dxrelu = relu_backward(dx2, cache_relu)
#print(np.shape(dxrelu), np.shape(dx2), np.shape(dscores) )
```

```
        dxrelu_reshaped = np.reshape(dxrelu, (N,-1))
        dx1, dw1, db1 = affine_backward(dxrelu_reshaped, cache_hid)
        grads['W2'] = dw2 + self.reg * self.params['W2']
        grads['b2'] = db2
        grads['W1'] = dw1 + self.reg * self.params['W1']
        grads['b1'] = db1


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

        return loss, grads


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.
- 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.
# ============================================================ #


L = self.num_layers
names = [str(i) for i in range(1, L+1 )]

for i in range(L):
    std = weight_scale
    string = names[i]
    if i==0:
        self.params['W'+ string] = std * np.random.randn(input_dim, hidden_dims[i])
        self.params['b'+string] = np.zeros(hidden_dims[i])
    elif i!=L-1:
        self.params['W'+ string] = std * np.random.randn(hidden_dims[i-1], hidden_d
        self.params['b'+string] = np.zeros(hidden_dims[i])
    else:
        self.params['W'+ string] = std * np.random.randn(hidden_dims[i-1], num_clas
        self.params['b'+string] = np.zeros(num_classes)

    #print('W'+ string,': ', np.shape(self.params['W'+ string] )  )



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

# When using dropout we need to pass a dropout_param dictionary to each
```

```
    # dropout layer so that the layer knows the dropout probability and the mode
    # (train / test). You can pass the same dropout_param to each dropout layer.
    self.dropout_param = {}
    if self.use_dropout:
      self.dropout_param = {'mode': 'train', 'p': dropout}
      if seed is not None:
        self.dropout_param['seed'] = seed

    # With batch normalization we need to keep track of running means and
    # variances, so we need to pass a special bn_param object to each batch
    # normalization layer. You should pass self.bn_params[0] to the forward pass
    # of the first batch normalization layer, self.bn_params[1] to the forward
    # pass of the second batch normalization layer, etc.
    self.bn_params = []
    if self.use_batchnorm:
      self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]

    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.

    Input / output: Same as TwoLayerNet above.
    """
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

    # Set train/test mode for batchnorm params and dropout param since they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    # ============================================================ #

    L = self.num_layers
```

```
names = [str(i) for i in range(1, L+1 )]

x = X
caches = []


for i in range(self.num_layers-1):
    #print('we are in the layer #', i+1)
    w = self.params['W' + names[i]]
    b = self.params['b' + names[i]]
    x, cache = affine_forward(x,w,b)
    #print('the w in teh cache has dims: ', np.shape(cache[1]) )
    caches.append(cache)
    x, cache = relu_forward(x)
    caches.append(cache)

w = self.params['W' + names[i+1]]
b = self.params['b' + names[i+1]]
x, cache = affine_forward(x,w,b)
#print('init cache is ', np.shape(cache[1] ) )

caches.append(cache)
#print('there are ', len(caches), ' entries in caches')
scores = x



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

# If test mode return early
if mode == 'test':
  return scores

loss, grads = 0.0, {}
# ============================================================ #
# YOUR CODE HERE:
#   Implement the backwards pass of the FC net and store the gradients
#   in the grads dict, so that grads[k] is the gradient of self.params[k]
#   Be sure your L2 regularization includes a 0.5 factor.
# ============================================================ #


loss, softmax_grad = softmax_loss(scores, y)
for i in range(self.num_layers):
    w = self.params['W'+names[i] ]
    loss += 0.5 * self.reg * np.sum(w * w)
```

```
dout = softmax_grad
#print('shape of dout is ', np.shape(dout) )
#print('cache is ', np.shape(caches[self.num_layers ][0]) , np.shape(caches[self.n


dout, dw, db = affine_backward(dout, caches[len(caches) - 1])

grads['W' + str(self.num_layers)] = dw + self.reg * self.params['W' + str(self.num
grads['b' + str(self.num_layers)] = db
#print("now we are in teh outside layer", 'W' + str(self.num_layers))

name_index = L-2
for i in range(len(caches) - 2, -1, -2):
    #print("now we are in layer #", i)
    dout = relu_backward(dout, caches[i])
    dx, dw, db = affine_backward(dout, caches[i-1] )
    #print(' the neames is ', names[name_index])
    #print(' the self.params[W + names[name_index]  ]  is ', np.shape(self.params[
    #print('W' + names[name_index] ,' the dw  is ', np.shape( dw ) )
    grads['W' + names[name_index] ] = dw + self.reg * self.params['W' + names[name
    grads['b' + names[name_index] ] = db
    name_index -= 1
    dout = dx

# ============================================================ #
# END YOUR CODE HERE
# ============================================================ #
return loss, grads
```

Solver.py

Untitled1

February 3, 2021

[ ]:
```
rom __future__ import print_function, division
from builtins import range
from builtins import object
import os
import pickle as pickle

import numpy as np

from nndl import optim


class Solver(object):
    """
    A Solver encapsulates all the logic necessary for training classification
    models. The Solver performs stochastic gradient descent using different
    update rules defined in optim.py.

    The solver accepts both training and validataion data and labels so it can
    periodically check classification accuracy on both training and validation
    data to watch out for overfitting.

    To train a model, you will first construct a Solver instance, passing the
    model, dataset, and various optoins (learning rate, batch size, etc) to the
    constructor. You will then call the train() method to run the optimization
    procedure and train the model.

    After the train() method returns, model.params will contain the parameters
    that performed best on the validation set over the course of training.
    In addition, the instance variable solver.loss_history will contain a list
    of all losses encountered during training and the instance variables
    solver.train_acc_history and solver.val_acc_history will be lists of the
    accuracies of the model on the training and validation set at each epoch.

    Example usage might look something like this:

    data = {
      'X_train': # training data
```

```
  'y_train': # training labels
  'X_val': # validation data
  'y_val': # validation labels
}
model = MyAwesomeModel(hidden_size=100, reg=10)
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=10, batch_size=100,
                print_every=100)
solver.train()


A Solver works on a model object that must conform to the following API:

- model.params must be a dictionary mapping string parameter names to numpy
  arrays containing parameter values.

- model.loss(X, y) must be a function that computes training-time loss and
  gradients, and test-time classification scores, with the following inputs
  and outputs:

  Inputs:
  - X: Array giving a minibatch of input data of shape (N, d_1, ..., d_k)
  - y: Array of labels, of shape (N,) giving labels for X where y[i] is the
    label for X[i].

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

  If y is not None, run a training time forward and backward pass and
  return a tuple of:
  - loss: Scalar 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.
"""

def __init__(self, model, data, **kwargs):
    """
    Construct a new Solver instance.

    Required arguments:
```

```
        - model: A model object conforming to the API described above
        - data: A dictionary of training and validation data containing:
          'X_train': Array, shape (N_train, d_1, ..., d_k) of training images
          'X_val': Array, shape (N_val, d_1, ..., d_k) of validation images
          'y_train': Array, shape (N_train,) of labels for training images
          'y_val': Array, shape (N_val,) of labels for validation images

        Optional arguments:
        - update_rule: A string giving the name of an update rule in optim.py.
          Default is 'sgd'.
        - optim_config: A dictionary containing hyperparameters that will be
          passed to the chosen update rule. Each update rule requires different
          hyperparameters (see optim.py) but all update rules require a
          'learning_rate' parameter so that should always be present.
        - lr_decay: A scalar for learning rate decay; after each epoch the
          learning rate is multiplied by this value.
        - batch_size: Size of minibatches used to compute loss and gradient
          during training.
        - num_epochs: The number of epochs to run for during training.
        - print_every: Integer; training losses will be printed every
          print_every iterations.
        - verbose: Boolean; if set to false then no output will be printed
          during training.
        - num_train_samples: Number of training samples used to check training
          accuracy; default is 1000; set to None to use entire training set.
        - num_val_samples: Number of validation samples to use to check val
          accuracy; default is None, which uses the entire validation set.
        - checkpoint_name: If not None, then save model checkpoints here every
          epoch.
        """
        self.model = model
        self.X_train = data['X_train']
        self.y_train = data['y_train']
        self.X_val = data['X_val']
        self.y_val = data['y_val']

        # Unpack keyword arguments
        self.update_rule = kwargs.pop('update_rule', 'sgd')
        self.optim_config = kwargs.pop('optim_config', {})
        self.lr_decay = kwargs.pop('lr_decay', 1.0)
        self.batch_size = kwargs.pop('batch_size', 100)
        self.num_epochs = kwargs.pop('num_epochs', 10)
        self.num_train_samples = kwargs.pop('num_train_samples', 1000)
        self.num_val_samples = kwargs.pop('num_val_samples', None)

        self.checkpoint_name = kwargs.pop('checkpoint_name', None)
        self.print_every = kwargs.pop('print_every', 10)
```

```
        self.verbose = kwargs.pop('verbose', True)

        # Throw an error if there are extra keyword arguments
        if len(kwargs) > 0:
            extra = ', '.join('"%s"' % k for k in list(kwargs.keys()))
            raise ValueError('Unrecognized arguments %s' % extra)

        # Make sure the update rule exists, then replace the string
        # name with the actual function
        if not hasattr(optim, self.update_rule):
            raise ValueError('Invalid update_rule "%s"' % self.update_rule)
        self.update_rule = getattr(optim, self.update_rule)

        self._reset()


    def _reset(self):
        """
        Set up some book-keeping variables for optimization. Don't call this
        manually.
        """
        # Set up some variables for book-keeping
        self.epoch = 0
        self.best_val_acc = 0
        self.best_params = {}
        self.loss_history = []
        self.train_acc_history = []
        self.val_acc_history = []

        # Make a deep copy of the optim_config for each parameter
        self.optim_configs = {}
        for p in self.model.params:
            d = {k: v for k, v in self.optim_config.items()}
            self.optim_configs[p] = d


    def _step(self):
        """
        Make a single gradient update. This is called by train() and should not
        be called manually.
        """
        # Make a minibatch of training data
        num_train = self.X_train.shape[0]
        batch_mask = np.random.choice(num_train, self.batch_size)
        X_batch = self.X_train[batch_mask]
        y_batch = self.y_train[batch_mask]
```

```
        # Compute loss and gradient
        loss, grads = self.model.loss(X_batch, y_batch)
        self.loss_history.append(loss)

        # Perform a parameter update
        for p, w in self.model.params.items():
            dw = grads[p]
            config = self.optim_configs[p]
            w = np.reshape(w, np.shape(dw) )       # WATCH OUT HERE !!!!!!!
            #print('shapes of w, dw, config are: ', np.shape(w), np.shape(dw),␣
↪np.shape(config['learning_rate'])  )
            next_w, next_config = self.update_rule(w, dw, config)
            self.model.params[p] = next_w
            self.optim_configs[p] = next_config


    def _save_checkpoint(self):
        if self.checkpoint_name is None: return
        checkpoint = {
          'model': self.model,
          'update_rule': self.update_rule,
          'lr_decay': self.lr_decay,
          'optim_config': self.optim_config,
          'batch_size': self.batch_size,
          'num_train_samples': self.num_train_samples,
          'num_val_samples': self.num_val_samples,
          'epoch': self.epoch,
          'loss_history': self.loss_history,
          'train_acc_history': self.train_acc_history,
          'val_acc_history': self.val_acc_history,
        }
        filename = '%s_epoch_%d.pkl' % (self.checkpoint_name, self.epoch)
        if self.verbose:
            print('Saving checkpoint to "%s"' % filename)
        with open(filename, 'wb') as f:
            pickle.dump(checkpoint, f)


    def check_accuracy(self, X, y, num_samples=None, batch_size=100):
        """
        Check accuracy of the model on the provided data.

        Inputs:
        - X: Array of data, of shape (N, d_1, ..., d_k)
        - y: Array of labels, of shape (N,)
        - num_samples: If not None, subsample the data and only test the model
          on num_samples datapoints.
```

```
        - batch_size: Split X and y into batches of this size to avoid using
          too much memory.

        Returns:
        - acc: Scalar giving the fraction of instances that were correctly
          classified by the model.
        """

        # Maybe subsample the data
        N = X.shape[0]
        if num_samples is not None and N > num_samples:
            mask = np.random.choice(N, num_samples)
            N = num_samples
            X = X[mask]
            y = y[mask]

        # Compute predictions in batches
        num_batches = N // batch_size
        if N % batch_size != 0:
            num_batches += 1
        y_pred = []
        for i in range(num_batches):
            start = i * batch_size
            end = (i + 1) * batch_size
            scores = self.model.loss(X[start:end])
            y_pred.append(np.argmax(scores, axis=1))
        y_pred = np.hstack(y_pred)
        acc = np.mean(y_pred == y)

        return acc


    def train(self):
        """
        Run optimization to train the model.
        """
        num_train = self.X_train.shape[0]
        iterations_per_epoch = max(num_train // self.batch_size, 1)
        num_iterations = self.num_epochs * iterations_per_epoch

        for t in range(num_iterations):
            self._step()

            # Maybe print training loss
            if self.verbose and t % self.print_every == 0:
                print('(Iteration %d / %d) loss: %f' % (
                       t + 1, num_iterations, self.loss_history[-1]))
```

```
        # At the end of every epoch, increment the epoch counter and decay
        # the learning rate.
        epoch_end = (t + 1) % iterations_per_epoch == 0
        if epoch_end:
            self.epoch += 1
            for k in self.optim_configs:
                self.optim_configs[k]['learning_rate'] *= self.lr_decay

        # Check train and val accuracy on the first iteration, the last
        # iteration, and at the end of each epoch.
        first_it = (t == 0)
        last_it = (t == num_iterations - 1)
        if first_it or last_it or epoch_end:
            train_acc = self.check_accuracy(self.X_train, self.y_train,
                num_samples=self.num_train_samples)
            val_acc = self.check_accuracy(self.X_val, self.y_val,
                num_samples=self.num_val_samples)
            self.train_acc_history.append(train_acc)
            self.val_acc_history.append(val_acc)
            self._save_checkpoint()

            if self.verbose:
                print('(Epoch %d / %d) train acc: %f; val_acc: %f' % (
                       self.epoch, self.num_epochs, train_acc, val_acc))

            # Keep track of the best model
            if val_acc > self.best_val_acc:
                self.best_val_acc = val_acc
                self.best_params = {}
                for k, v in self.model.params.items():
                    self.best_params[k] = v.copy()

    # At the end of training swap the best params into the model
    self.model.params = self.best_params
```