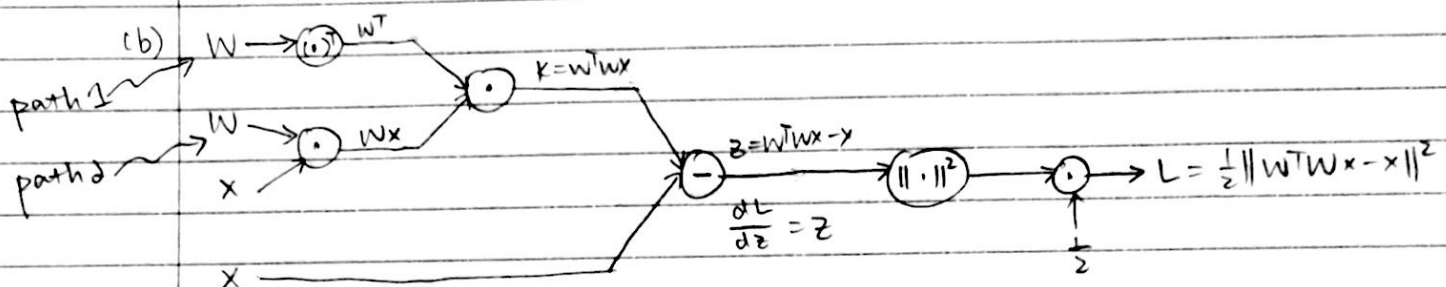
EE239AS
HW3
Yutong Lu
005027184

Problem 1 (a) Since Wx is a lower dimension representation of x, W^TWx wants to reconstruct a $\hat{x}$ as similar as x. By minimizing $L = \frac{1}{2}\|W^TWx - x\|^2$, if this loss goes to zero, then W will be an eigenvector of x and $W^TW = I$, then Wx could preserve the most important information about x with reduced dimension.

(b)



(c) These two paths should be summed up. Mathematically,

$$\frac{dL}{dW} = \underbrace{\left(\frac{dL}{dW^T}\right)^T}_{\text{path 1}} + \underbrace{\frac{d(Wx)}{dW} \cdot \frac{dL}{d(Wx)}}_{\text{path 2}}$$

(d) Let $z = W^TWx - x$ Let $K = W^TW$

$$\Rightarrow \frac{dL}{dz} = \frac{1}{2} \cdot (2z) = z$$

$$\Rightarrow \frac{dL}{dK} = \frac{dL}{dz}$$

path 1: $\frac{dL}{dW^T} = \frac{dK}{dW^T} \cdot \frac{dL}{dK} = \frac{dL}{dK}(Wx)^T = z(Wx)^T$

$\because \frac{dL}{dW^T} = \left(\frac{dL}{dW}\right)^T$ when L is a scalar $\Rightarrow \frac{dL}{dW} = \left(\frac{dL}{dW^T}\right)^T = \left(z(Wx)^T\right)^T$

$= Wxz^T$

path 2: $\frac{dL}{d(Wx)} = \frac{dK}{d(Wx)} \frac{dL}{dK} = (W^T)^T \frac{dL}{dK} = Wz$

$\frac{dL}{dW} = \frac{d(Wx)}{dW} \frac{dL}{d(Wx)} = \frac{dL}{d(Wx)} x^T = Wzx^T$

$\Rightarrow$ To sum up, $\frac{dL}{dW} = Wxz^T + Wzx^T$

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

$= Wx[(W^TW - I)x]^T + W[(W^TW - I)x]x^T$

$= Wxx^T(W^TW - I)^T + W(W^TW - I)xx^T$

This is the 2-layer neural network workbook for ECE 239AS 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.

In [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))))
```

Toy example

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

In [2]:

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

In [3]:

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

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

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

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

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

Compute forward pass scores

In [4]:

```
## Implement the forward pass of the neural network.

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

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

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

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

Difference between your scores and correct scores:
3.381231210991542e-08
```

Forward pass loss

In [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
```

In [6]:

```
print(loss)
```

```
1.071696123862817
```

Backward pass

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

In [7]:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pas
s.
# 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_nu
m, grads[param_name])))
```

```
W2 max relative error: 3.887723777584258e-10
b2 max relative error: 1.8392106647421603e-10
W1 max relative error: 4.820182284856054e-09
b1 max relative error: 1.7679551853461256e-09
```

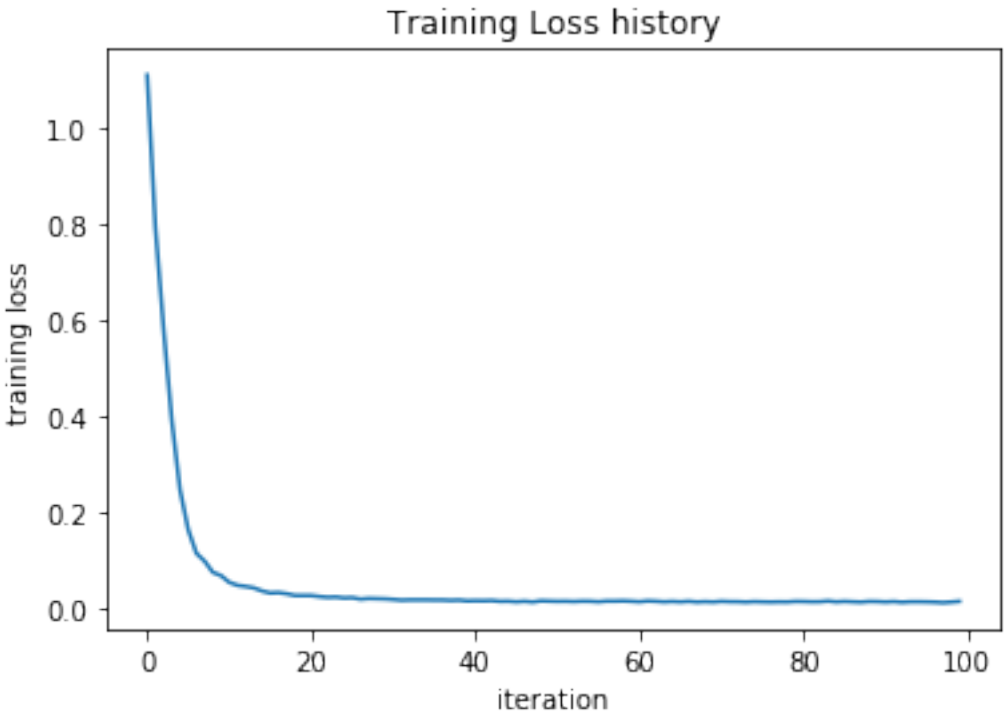
Training the network

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

In [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:  0.014497864587765941
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [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 = 'cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

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

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

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


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

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

Running SGD

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

In [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 2.3021201592072362
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.2518259043164135
iteration 400 / 1000: loss 2.1889952350467756
iteration 500 / 1000: loss 2.1162527791897743
iteration 600 / 1000: loss 2.0646708276982166
iteration 700 / 1000: loss 1.9901688623083942
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856498
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

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

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

In [11]:

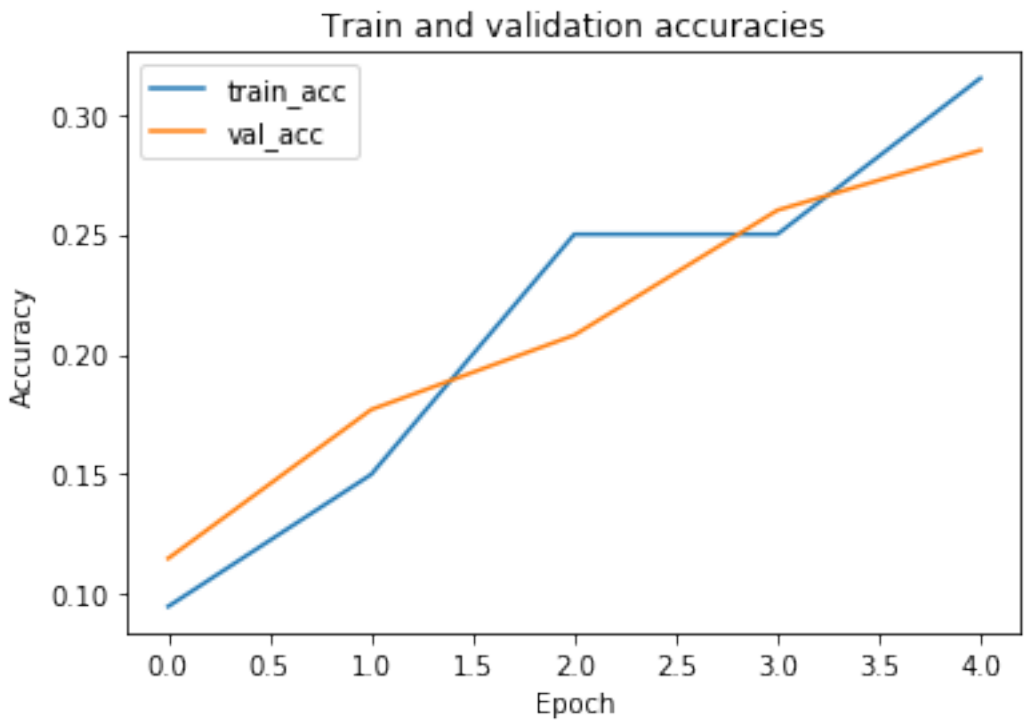
```
stats['train_acc_history']
```

Out[11]:

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

In [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

train, = plt.plot(stats['train_acc_history'], label='train')
val, = plt.plot(stats['val_acc_history'], label='val')
plt.title('Train and validation accuracies')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend([train, val], ["train_acc", "val_acc"])
plt.show()

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



Answers:

(1) During the initialization, the initialized weights are just small random numbers so that it is really bad initialization of weights. Moreover, bad choices of learning rate, learning rate decay, batch size, and regularization may also be the causes.

(2) To fix this problem, it is better to use grid search to see which combination of parameters will have better performance and accuracy.

Optimize the neural network

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

In [13]:

```
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)!
# ================================================================ #
pass

best_val = -1
results = {}
np.random.seed(0)

batch_sizes = [200,  250]
lrs = [1e-4, 5e-4, 1e-3, 3e-3]
regs = [0.1, 0.25, 0.4]
lr_decays = [0.95, 0.98]

grid_search = [(x,y,z,h) for x in batch_sizes for y in lrs for z in regs for h i
n lr_decays]

for bs, lr, reg, lr_decay in grid_search:

    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=2000, batch_size = bs,
                learning_rate=lr, learning_rate_decay=lr_decay,
                reg=reg, verbose=True)

    val_acc = (net.predict(X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)

    results[(bs,lr,reg, lr_decay)]=val_acc

    if val_acc > best_val:
        best_val = val_acc
        best_net = net
print('Best net: ', best_net)
print('Validation accuracy with the best net: ', best_val)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
# best_net = net
```

```
iteration 0 / 2000: loss 2.3026638589680744
iteration 100 / 2000: loss 2.30210176404191
iteration 200 / 2000: loss 2.2951614369643
iteration 300 / 2000: loss 2.246201353536789
iteration 400 / 2000: loss 2.203403253501954
iteration 500 / 2000: loss 2.1279197745355067
iteration 600 / 2000: loss 2.1236087070062735
iteration 700 / 2000: loss 2.039737782235252
iteration 800 / 2000: loss 1.9839960033238633
iteration 900 / 2000: loss 1.9625411952174154
iteration 1000 / 2000: loss 2.027072849275479
iteration 1100 / 2000: loss 1.9446897828687655
iteration 1200 / 2000: loss 1.814473358697451
iteration 1300 / 2000: loss 1.8494241174592863
iteration 1400 / 2000: loss 1.9008318824491692
iteration 1500 / 2000: loss 1.8385190886698948
iteration 1600 / 2000: loss 1.8368841390423025
iteration 1700 / 2000: loss 1.8307050991237732
iteration 1800 / 2000: loss 1.8442892096222672
iteration 1900 / 2000: loss 1.8238398820814163
Validation accuracy:  0.355
iteration 0 / 2000: loss 2.3026600844489074
iteration 100 / 2000: loss 2.3022648635058904
iteration 200 / 2000: loss 2.2961408606849156
iteration 300 / 2000: loss 2.2602741299128137
iteration 400 / 2000: loss 2.179575856481825
iteration 500 / 2000: loss 2.1401379021264386
iteration 600 / 2000: loss 2.102766501538382
iteration 700 / 2000: loss 2.0662616107970933
iteration 800 / 2000: loss 1.994497979078383
iteration 900 / 2000: loss 1.939616744426735
iteration 1000 / 2000: loss 1.9603583407062026
iteration 1100 / 2000: loss 1.9630546713226704
iteration 1200 / 2000: loss 1.975654076730351
iteration 1300 / 2000: loss 1.8920505014968299
iteration 1400 / 2000: loss 1.8046940150915358
iteration 1500 / 2000: loss 1.819031267681964
iteration 1600 / 2000: loss 1.7746194448267636
iteration 1700 / 2000: loss 1.761371363192013
iteration 1800 / 2000: loss 1.8618554606532083
iteration 1900 / 2000: loss 1.7395368326762994
Validation accuracy:  0.377
iteration 0 / 2000: loss 2.3027895711321635
iteration 100 / 2000: loss 2.302340324807589
iteration 200 / 2000: loss 2.2982331499201534
iteration 300 / 2000: loss 2.272371491754205
iteration 400 / 2000: loss 2.1751262596367527
iteration 500 / 2000: loss 2.1557952427077822
iteration 600 / 2000: loss 2.02372121214566
iteration 700 / 2000: loss 2.1106796963451395
iteration 800 / 2000: loss 2.0083068350962834
iteration 900 / 2000: loss 2.0194179389492293
iteration 1000 / 2000: loss 1.919561206470935
iteration 1100 / 2000: loss 1.9378796680327308
```

```
iteration 1200 / 2000: loss 1.9197001039499777
iteration 1300 / 2000: loss 1.9606243431986992
iteration 1400 / 2000: loss 1.8735707140255078
iteration 1500 / 2000: loss 1.9255726700180007
iteration 1600 / 2000: loss 1.8780747089486185
iteration 1700 / 2000: loss 1.8885250492330468
iteration 1800 / 2000: loss 1.7254101447426033
iteration 1900 / 2000: loss 1.8322493642803193
Validation accuracy:  0.367
iteration 0 / 2000: loss 2.3027867607663053
iteration 100 / 2000: loss 2.3023625915358568
iteration 200 / 2000: loss 2.2994698471630173
iteration 300 / 2000: loss 2.268441389330448
iteration 400 / 2000: loss 2.2053255333985007
iteration 500 / 2000: loss 2.06572481757792
iteration 600 / 2000: loss 2.1654993090171977
iteration 700 / 2000: loss 2.040174334021532
iteration 800 / 2000: loss 2.0330840561011083
iteration 900 / 2000: loss 1.9039512873161162
iteration 1000 / 2000: loss 1.9616421022287012
iteration 1100 / 2000: loss 1.9099180144357022
iteration 1200 / 2000: loss 1.822829617668141
iteration 1300 / 2000: loss 1.8836115109613791
iteration 1400 / 2000: loss 1.8290961593740198
iteration 1500 / 2000: loss 1.779442353197949
iteration 1600 / 2000: loss 1.7860612787421801
iteration 1700 / 2000: loss 1.7431522869771354
iteration 1800 / 2000: loss 1.7188814685249874
iteration 1900 / 2000: loss 1.6842397923857368
Validation accuracy:  0.378
iteration 0 / 2000: loss 2.302894980993095
iteration 100 / 2000: loss 2.3024554858903303
iteration 200 / 2000: loss 2.2955905810137986
iteration 300 / 2000: loss 2.2499620813934347
iteration 400 / 2000: loss 2.2075320679313806
iteration 500 / 2000: loss 2.13777966841273
iteration 600 / 2000: loss 2.109832855241814
iteration 700 / 2000: loss 2.004260682389872
iteration 800 / 2000: loss 1.914936083213602
iteration 900 / 2000: loss 1.9543060029839685
iteration 1000 / 2000: loss 2.0322188792119995
iteration 1100 / 2000: loss 1.8388993488009338
iteration 1200 / 2000: loss 1.8728455832926412
iteration 1300 / 2000: loss 1.9158020102902893
iteration 1400 / 2000: loss 1.8894226578104671
iteration 1500 / 2000: loss 1.850559385296674
iteration 1600 / 2000: loss 1.8023317670439098
iteration 1700 / 2000: loss 1.8069508104780119
iteration 1800 / 2000: loss 1.7685984607572813
iteration 1900 / 2000: loss 1.839047684956903
Validation accuracy:  0.353
iteration 0 / 2000: loss 2.3028647639860305
iteration 100 / 2000: loss 2.3022780692103084
```

```
iteration 200 / 2000: loss 2.2983083777878606
iteration 300 / 2000: loss 2.240124452699326
iteration 400 / 2000: loss 2.2470912706329527
iteration 500 / 2000: loss 2.1281797003746825
iteration 600 / 2000: loss 2.03737188363572
iteration 700 / 2000: loss 1.991936023639477
iteration 800 / 2000: loss 1.965547905589006
iteration 900 / 2000: loss 1.9596715636669633
iteration 1000 / 2000: loss 1.9671423372457493
iteration 1100 / 2000: loss 1.9142978704828562
iteration 1200 / 2000: loss 1.922332229724551
iteration 1300 / 2000: loss 1.8349243122317853
iteration 1400 / 2000: loss 1.9394900564768567
iteration 1500 / 2000: loss 1.7143013238434393
iteration 1600 / 2000: loss 1.871382142084722
iteration 1700 / 2000: loss 1.7314428790118177
iteration 1800 / 2000: loss 1.8102553509716577
iteration 1900 / 2000: loss 1.7793173640616466
Validation accuracy:  0.37
iteration 0 / 2000: loss 2.3026780177452992
iteration 100 / 2000: loss 2.1259148137621344
iteration 200 / 2000: loss 1.9605740913722922
iteration 300 / 2000: loss 1.867186002356755
iteration 400 / 2000: loss 1.883828558840508
iteration 500 / 2000: loss 1.7253703846576007
iteration 600 / 2000: loss 1.715307238282935
iteration 700 / 2000: loss 1.6699906381952374
iteration 800 / 2000: loss 1.5946188204818403
iteration 900 / 2000: loss 1.5765058940457224
iteration 1000 / 2000: loss 1.552390422287512
iteration 1100 / 2000: loss 1.6443139532959081
iteration 1200 / 2000: loss 1.5997937061015053
iteration 1300 / 2000: loss 1.503691297931066
iteration 1400 / 2000: loss 1.4084430699999848
iteration 1500 / 2000: loss 1.4745305808772524
iteration 1600 / 2000: loss 1.4509423595394024
iteration 1700 / 2000: loss 1.4553626913408246
iteration 1800 / 2000: loss 1.4539606265785343
iteration 1900 / 2000: loss 1.419997888994787
Validation accuracy:  0.478
iteration 0 / 2000: loss 2.302689735953845
iteration 100 / 2000: loss 2.118293225852991
iteration 200 / 2000: loss 1.94789728106905
iteration 300 / 2000: loss 1.7750478735775646
iteration 400 / 2000: loss 1.7702986785718602
iteration 500 / 2000: loss 1.6789460674345218
iteration 600 / 2000: loss 1.7453065984951313
iteration 700 / 2000: loss 1.566964362107688
iteration 800 / 2000: loss 1.5679791925601294
iteration 900 / 2000: loss 1.6241809314971607
iteration 1000 / 2000: loss 1.579485094705646
iteration 1100 / 2000: loss 1.5669900487620176
iteration 1200 / 2000: loss 1.52325672539396
```

```
iteration 1300 / 2000: loss 1.545243146366852
iteration 1400 / 2000: loss 1.5050555127251073
iteration 1500 / 2000: loss 1.5003626278866098
iteration 1600 / 2000: loss 1.5899512440745176
iteration 1700 / 2000: loss 1.5452378282153836
iteration 1800 / 2000: loss 1.5121359670301677
iteration 1900 / 2000: loss 1.557258982412892
Validation accuracy:  0.499
iteration 0 / 2000: loss 2.3027992680458627
iteration 100 / 2000: loss 2.0849930334121742
iteration 200 / 2000: loss 1.8207540774806603
iteration 300 / 2000: loss 1.916228307672527
iteration 400 / 2000: loss 1.7640271151794218
iteration 500 / 2000: loss 1.674983692126361
iteration 600 / 2000: loss 1.7004010274398522
iteration 700 / 2000: loss 1.7350059037053946
iteration 800 / 2000: loss 1.7006447798733526
iteration 900 / 2000: loss 1.8029850565234933
iteration 1000 / 2000: loss 1.5169420424823477
iteration 1100 / 2000: loss 1.506650862330644
iteration 1200 / 2000: loss 1.5062798301157567
iteration 1300 / 2000: loss 1.556485441021899
iteration 1400 / 2000: loss 1.5112190131276753
iteration 1500 / 2000: loss 1.5051875093318325
iteration 1600 / 2000: loss 1.4979911324353385
iteration 1700 / 2000: loss 1.5673270745781456
iteration 1800 / 2000: loss 1.4951072005142616
iteration 1900 / 2000: loss 1.4972180407221214
Validation accuracy:  0.489
iteration 0 / 2000: loss 2.3027838447846376
iteration 100 / 2000: loss 2.0945838277084436
iteration 200 / 2000: loss 1.854833477485209
iteration 300 / 2000: loss 1.773948268025538
iteration 400 / 2000: loss 1.7138884030693393
iteration 500 / 2000: loss 1.6716139094385372
iteration 600 / 2000: loss 1.7953573075410378
iteration 700 / 2000: loss 1.7711836577668356
iteration 800 / 2000: loss 1.671932563540694
iteration 900 / 2000: loss 1.4812696380446881
iteration 1000 / 2000: loss 1.5890314090874185
iteration 1100 / 2000: loss 1.488347666923681
iteration 1200 / 2000: loss 1.6519744152578546
iteration 1300 / 2000: loss 1.5861491590151962
iteration 1400 / 2000: loss 1.4025543102412403
iteration 1500 / 2000: loss 1.5172708066710678
iteration 1600 / 2000: loss 1.6107597240815363
iteration 1700 / 2000: loss 1.5269989242910227
iteration 1800 / 2000: loss 1.409272309774122
iteration 1900 / 2000: loss 1.3960280813053159
Validation accuracy:  0.467
iteration 0 / 2000: loss 2.302899750011732
iteration 100 / 2000: loss 2.134061081658805
iteration 200 / 2000: loss 1.9899168239944731
```

```
iteration 300 / 2000: loss 1.812271264094113
iteration 400 / 2000: loss 1.781376963875327
iteration 500 / 2000: loss 1.7349285451080383
iteration 600 / 2000: loss 1.657314436106382
iteration 700 / 2000: loss 1.6947698636545208
iteration 800 / 2000: loss 1.5742789541928643
iteration 900 / 2000: loss 1.561195430445483
iteration 1000 / 2000: loss 1.5626548346949523
iteration 1100 / 2000: loss 1.564940978266648
iteration 1200 / 2000: loss 1.7260313380438188
iteration 1300 / 2000: loss 1.5653695663272607
iteration 1400 / 2000: loss 1.38116585610898
iteration 1500 / 2000: loss 1.5633975137836533
iteration 1600 / 2000: loss 1.5217938975910716
iteration 1700 / 2000: loss 1.5520790611618436
iteration 1800 / 2000: loss 1.5404055432249755
iteration 1900 / 2000: loss 1.4409948302623814
Validation accuracy:  0.476
iteration 0 / 2000: loss 2.3028812381348343
iteration 100 / 2000: loss 2.08930414794342
iteration 200 / 2000: loss 1.8998832713291416
iteration 300 / 2000: loss 1.9332084887715641
iteration 400 / 2000: loss 1.766450284543842
iteration 500 / 2000: loss 1.7014287314903975
iteration 600 / 2000: loss 1.8041140769167263
iteration 700 / 2000: loss 1.6648651184214636
iteration 800 / 2000: loss 1.6141846412996639
iteration 900 / 2000: loss 1.5912465242767189
iteration 1000 / 2000: loss 1.5677032569286031
iteration 1100 / 2000: loss 1.4629886524947648
iteration 1200 / 2000: loss 1.608543624193205
iteration 1300 / 2000: loss 1.4279685259523327
iteration 1400 / 2000: loss 1.58495864812971
iteration 1500 / 2000: loss 1.534308331963917
iteration 1600 / 2000: loss 1.392839561060509
iteration 1700 / 2000: loss 1.5875583300309706
iteration 1800 / 2000: loss 1.3434621579651498
iteration 1900 / 2000: loss 1.5093283168475267
Validation accuracy:  0.48
iteration 0 / 2000: loss 2.302679368786093
iteration 100 / 2000: loss 2.0195780258187837
iteration 200 / 2000: loss 1.8642810711687816
iteration 300 / 2000: loss 1.6984826364738115
iteration 400 / 2000: loss 1.5729681778467253
iteration 500 / 2000: loss 1.5594194790554412
iteration 600 / 2000: loss 1.5684042336470376
iteration 700 / 2000: loss 1.4487636450417627
iteration 800 / 2000: loss 1.521434176611547
iteration 900 / 2000: loss 1.3934034435524039
iteration 1000 / 2000: loss 1.4987515124724866
iteration 1100 / 2000: loss 1.491193028320546
iteration 1200 / 2000: loss 1.435152687192552
iteration 1300 / 2000: loss 1.4201740455233205
```

```
iteration 1400 / 2000: loss 1.5406054446194664
iteration 1500 / 2000: loss 1.3119996083012209
iteration 1600 / 2000: loss 1.5618162405378977
iteration 1700 / 2000: loss 1.4560503346763807
iteration 1800 / 2000: loss 1.40479150508626
iteration 1900 / 2000: loss 1.3883255821829656
Validation accuracy:  0.495
iteration 0 / 2000: loss 2.302673002900049
iteration 100 / 2000: loss 1.9278465936634457
iteration 200 / 2000: loss 1.757760687545754
iteration 300 / 2000: loss 1.5955985873372527
iteration 400 / 2000: loss 1.6351120671336958
iteration 500 / 2000: loss 1.5607281592534332
iteration 600 / 2000: loss 1.5686606988285166
iteration 700 / 2000: loss 1.595828983049628
iteration 800 / 2000: loss 1.5180052230431722
iteration 900 / 2000: loss 1.5869252547509562
iteration 1000 / 2000: loss 1.3553674096870845
iteration 1100 / 2000: loss 1.4902282491438346
iteration 1200 / 2000: loss 1.468483121630987
iteration 1300 / 2000: loss 1.4859934548278875
iteration 1400 / 2000: loss 1.3848483952583643
iteration 1500 / 2000: loss 1.3598039347660935
iteration 1600 / 2000: loss 1.4637475197205625
iteration 1700 / 2000: loss 1.3177227491551398
iteration 1800 / 2000: loss 1.4141885660519
iteration 1900 / 2000: loss 1.353463266199145
Validation accuracy:  0.491
iteration 0 / 2000: loss 2.3027642831831687
iteration 100 / 2000: loss 1.9388852843918494
iteration 200 / 2000: loss 1.8701377242668895
iteration 300 / 2000: loss 1.799445093091157
iteration 400 / 2000: loss 1.6446166778954143
iteration 500 / 2000: loss 1.5730189995805488
iteration 600 / 2000: loss 1.5397625151187067
iteration 700 / 2000: loss 1.6031665824099914
iteration 800 / 2000: loss 1.505430510158995
iteration 900 / 2000: loss 1.495610455273355
iteration 1000 / 2000: loss 1.5271863602436866
iteration 1100 / 2000: loss 1.4448670286435579
iteration 1200 / 2000: loss 1.528566668923104
iteration 1300 / 2000: loss 1.4077542051317167
iteration 1400 / 2000: loss 1.2855161254880525
iteration 1500 / 2000: loss 1.4939365712782853
iteration 1600 / 2000: loss 1.3633656450801122
iteration 1700 / 2000: loss 1.3373374864273082
iteration 1800 / 2000: loss 1.411143606936436
iteration 1900 / 2000: loss 1.4265061139282635
Validation accuracy:  0.486
iteration 0 / 2000: loss 2.302747371719807
iteration 100 / 2000: loss 1.897357911916037
iteration 200 / 2000: loss 1.786428803279426
iteration 300 / 2000: loss 1.704599467822474
```

```
iteration 400 / 2000: loss 1.4964617254401906
iteration 500 / 2000: loss 1.5463229063919242
iteration 600 / 2000: loss 1.5373615328349912
iteration 700 / 2000: loss 1.6773931324710618
iteration 800 / 2000: loss 1.4811975773222732
iteration 900 / 2000: loss 1.5334109510164635
iteration 1000 / 2000: loss 1.3996534691360842
iteration 1100 / 2000: loss 1.4596076723712372
iteration 1200 / 2000: loss 1.3736200005979877
iteration 1300 / 2000: loss 1.4637492140082564
iteration 1400 / 2000: loss 1.4281486054305155
iteration 1500 / 2000: loss 1.4103408746223653
iteration 1600 / 2000: loss 1.5223137449136583
iteration 1700 / 2000: loss 1.4176883396417612
iteration 1800 / 2000: loss 1.3340090567856635
iteration 1900 / 2000: loss 1.3312837502139425
Validation accuracy:  0.476
iteration 0 / 2000: loss 2.302908661872139
iteration 100 / 2000: loss 2.002636023588165
iteration 200 / 2000: loss 1.804536748258138
iteration 300 / 2000: loss 1.6300263328743019
iteration 400 / 2000: loss 1.6106906942375339
iteration 500 / 2000: loss 1.6582360410673733
iteration 600 / 2000: loss 1.564059532111144
iteration 700 / 2000: loss 1.6514897599243448
iteration 800 / 2000: loss 1.5165666424826703
iteration 900 / 2000: loss 1.5376323853786071
iteration 1000 / 2000: loss 1.4074696196270406
iteration 1100 / 2000: loss 1.5604353837689808
iteration 1200 / 2000: loss 1.535990862872055
iteration 1300 / 2000: loss 1.4990456613916106
iteration 1400 / 2000: loss 1.4256237503616138
iteration 1500 / 2000: loss 1.5021110878423567
iteration 1600 / 2000: loss 1.5389404945621443
iteration 1700 / 2000: loss 1.503898654538859
iteration 1800 / 2000: loss 1.3313104906497282
iteration 1900 / 2000: loss 1.3580879738022276
Validation accuracy:  0.503
iteration 0 / 2000: loss 2.302895468403578
iteration 100 / 2000: loss 1.9412300581428659
iteration 200 / 2000: loss 1.8343977172368038
iteration 300 / 2000: loss 1.6829481784481437
iteration 400 / 2000: loss 1.5645600190194382
iteration 500 / 2000: loss 1.672301166718145
iteration 600 / 2000: loss 1.458955843375531
iteration 700 / 2000: loss 1.62260916946066
iteration 800 / 2000: loss 1.5769165076168439
iteration 900 / 2000: loss 1.613161088060471
iteration 1000 / 2000: loss 1.4532339946083455
iteration 1100 / 2000: loss 1.5861365842872348
iteration 1200 / 2000: loss 1.4786754731040146
iteration 1300 / 2000: loss 1.4387325330847758
iteration 1400 / 2000: loss 1.4788133468187756
```

```
iteration 1500 / 2000: loss 1.487106219471395
iteration 1600 / 2000: loss 1.3632401054642784
iteration 1700 / 2000: loss 1.388101410275256
iteration 1800 / 2000: loss 1.448713627094077
iteration 1900 / 2000: loss 1.3448360246739226
Validation accuracy:  0.478
iteration 0 / 2000: loss 2.3026680570839027
iteration 100 / 2000: loss 1.7945742721749296
iteration 200 / 2000: loss 1.832194272635748
iteration 300 / 2000: loss 1.6237403065423122
iteration 400 / 2000: loss 1.6542061182088947
iteration 500 / 2000: loss 1.5604536530675799
iteration 600 / 2000: loss 1.485547259311235
iteration 700 / 2000: loss 1.5672012665856778
iteration 800 / 2000: loss 1.7686651562059752
iteration 900 / 2000: loss 1.6974410901609442
iteration 1000 / 2000: loss 1.4372424191566595
iteration 1100 / 2000: loss 1.3535184271418674
iteration 1200 / 2000: loss 1.4653244749979868
iteration 1300 / 2000: loss 1.5705449502523574
iteration 1400 / 2000: loss 1.5480420269980095
iteration 1500 / 2000: loss 1.7782402012431002
iteration 1600 / 2000: loss 1.5258033644079732
iteration 1700 / 2000: loss 1.4406312332737807
iteration 1800 / 2000: loss 1.4611741001682634
iteration 1900 / 2000: loss 1.5077498796943358
Validation accuracy:  0.466
iteration 0 / 2000: loss 2.3026412099071716
iteration 100 / 2000: loss 1.8451384410807854
iteration 200 / 2000: loss 1.7275291799200183
iteration 300 / 2000: loss 1.7850823913496867
iteration 400 / 2000: loss 1.714074016065132
iteration 500 / 2000: loss 1.7155843868338017
iteration 600 / 2000: loss 1.825845785142246
iteration 700 / 2000: loss 1.5589507095882782
iteration 800 / 2000: loss 1.6663351790961256
iteration 900 / 2000: loss 1.660395446050145
iteration 1000 / 2000: loss 1.7583630929981926
iteration 1100 / 2000: loss 1.7613325566863158
iteration 1200 / 2000: loss 1.7774526285366123
iteration 1300 / 2000: loss 1.6363357588398604
iteration 1400 / 2000: loss 1.568335418469334
iteration 1500 / 2000: loss 1.6881630156086438
iteration 1600 / 2000: loss 1.6232616774678092
iteration 1700 / 2000: loss 1.4044626888671528
iteration 1800 / 2000: loss 1.588013202200853
iteration 1900 / 2000: loss 1.5773259372102835
Validation accuracy:  0.453
iteration 0 / 2000: loss 2.3027623662758288
iteration 100 / 2000: loss 1.9560147892768465
iteration 200 / 2000: loss 1.7068731591028232
iteration 300 / 2000: loss 1.7326099975360636
iteration 400 / 2000: loss 1.67696358674299
```

```
iteration 500 / 2000: loss 1.6422541834045643
iteration 600 / 2000: loss 1.5186573765924434
iteration 700 / 2000: loss 1.6846214016047731
iteration 800 / 2000: loss 1.5884152783125915
iteration 900 / 2000: loss 1.5729815611699622
iteration 1000 / 2000: loss 1.5513576991514777
iteration 1100 / 2000: loss 1.50887969117808
iteration 1200 / 2000: loss 1.7149875179992313
iteration 1300 / 2000: loss 1.7038864857103135
iteration 1400 / 2000: loss 1.3954223937897803
iteration 1500 / 2000: loss 1.744308954921646
iteration 1600 / 2000: loss 1.4686527838426255
iteration 1700 / 2000: loss 1.487563737635669
iteration 1800 / 2000: loss 1.4775647982816944
iteration 1900 / 2000: loss 1.3935284148894702
Validation accuracy:  0.463
iteration 0 / 2000: loss 2.3027949189199783
iteration 100 / 2000: loss 2.009824118563338
iteration 200 / 2000: loss 1.8600550639267746
iteration 300 / 2000: loss 1.7258405156135155
iteration 400 / 2000: loss 1.6036636297718276
iteration 500 / 2000: loss 1.690643321849726
iteration 600 / 2000: loss 1.7608836248157758
iteration 700 / 2000: loss 1.802557932112294
iteration 800 / 2000: loss 1.5525913075528157
iteration 900 / 2000: loss 2.142620149941261
iteration 1000 / 2000: loss 1.6469155844927015
iteration 1100 / 2000: loss 1.641211489212414
iteration 1200 / 2000: loss 1.628098084869995
iteration 1300 / 2000: loss 1.5425529113076213
iteration 1400 / 2000: loss 1.5603308380550462
iteration 1500 / 2000: loss 1.6916093221767023
iteration 1600 / 2000: loss 1.738285828225348
iteration 1700 / 2000: loss 1.6395565532703644
iteration 1800 / 2000: loss 1.5372169779950178
iteration 1900 / 2000: loss 1.6610719973739978
Validation accuracy:  0.48
iteration 0 / 2000: loss 2.302912895701103
iteration 100 / 2000: loss 1.7851823063172967
iteration 200 / 2000: loss 1.8657604783439412
iteration 300 / 2000: loss 1.6273707590080442
iteration 400 / 2000: loss 1.7563350725130873
iteration 500 / 2000: loss 1.6743839440120039
iteration 600 / 2000: loss 1.7176371244865085
iteration 700 / 2000: loss 1.6495678467565262
iteration 800 / 2000: loss 1.717228684844256
iteration 900 / 2000: loss 1.7841126875026454
iteration 1000 / 2000: loss 1.5180012805664842
iteration 1100 / 2000: loss 1.5435953840783214
iteration 1200 / 2000: loss 1.510579691340592
iteration 1300 / 2000: loss 1.6665204891287386
iteration 1400 / 2000: loss 1.585827431189911
iteration 1500 / 2000: loss 1.618707563536237
```

```
iteration 1600 / 2000: loss 1.4860344570786035
iteration 1700 / 2000: loss 1.5550508855763836
iteration 1800 / 2000: loss 1.5965011365936639
iteration 1900 / 2000: loss 1.845547881616708
Validation accuracy:  0.469
iteration 0 / 2000: loss 2.302893818782306
iteration 100 / 2000: loss 1.8285683075662038
iteration 200 / 2000: loss 1.7776880894660836
iteration 300 / 2000: loss 1.757795138412241
iteration 400 / 2000: loss 1.699882788557239
iteration 500 / 2000: loss 1.7704184685450037
iteration 600 / 2000: loss 1.6313333230952578
iteration 700 / 2000: loss 1.766328949207234
iteration 800 / 2000: loss 1.9139695710753764
iteration 900 / 2000: loss 1.6855461614271272
iteration 1000 / 2000: loss 1.5476702105170992
iteration 1100 / 2000: loss 1.8780753519577496
iteration 1200 / 2000: loss 1.946076810846501
iteration 1300 / 2000: loss 1.5823106625753416
iteration 1400 / 2000: loss 1.864233351088775
iteration 1500 / 2000: loss 1.991902867000987
iteration 1600 / 2000: loss 1.813198165278381
iteration 1700 / 2000: loss 2.0293660849394755
iteration 1800 / 2000: loss 1.5599354714361053
iteration 1900 / 2000: loss 1.5198666356377133
Validation accuracy:  0.439
iteration 0 / 2000: loss 2.30263396321753
iteration 100 / 2000: loss 2.3021278109850503
iteration 200 / 2000: loss 2.2969251675691473
iteration 300 / 2000: loss 2.247111852412272
iteration 400 / 2000: loss 2.1478993218440086
iteration 500 / 2000: loss 2.1528156800183598
iteration 600 / 2000: loss 2.0516213418164164
iteration 700 / 2000: loss 2.02755085067638
iteration 800 / 2000: loss 2.04815119659061
iteration 900 / 2000: loss 1.9979708810212193
iteration 1000 / 2000: loss 1.9596222579009284
iteration 1100 / 2000: loss 1.8706178929275172
iteration 1200 / 2000: loss 1.9478891927750641
iteration 1300 / 2000: loss 1.8713384627894387
iteration 1400 / 2000: loss 1.8571388585645736
iteration 1500 / 2000: loss 1.8505504674999222
iteration 1600 / 2000: loss 1.7968548781167528
iteration 1700 / 2000: loss 1.7902937464674882
iteration 1800 / 2000: loss 1.7423562446853385
iteration 1900 / 2000: loss 1.8621728705293785
Validation accuracy:  0.359
iteration 0 / 2000: loss 2.3026611441301785
iteration 100 / 2000: loss 2.3022565913911337
iteration 200 / 2000: loss 2.297299574506293
iteration 300 / 2000: loss 2.250054098785149
iteration 400 / 2000: loss 2.212103109476443
iteration 500 / 2000: loss 2.069301491210498
```

```
iteration 600 / 2000: loss 2.0603145747276397
iteration 700 / 2000: loss 2.016557634571314
iteration 800 / 2000: loss 2.0189094526906066
iteration 900 / 2000: loss 1.9044883503917829
iteration 1000 / 2000: loss 1.8998631508628712
iteration 1100 / 2000: loss 1.9326657166084442
iteration 1200 / 2000: loss 1.7902862878536987
iteration 1300 / 2000: loss 1.8711159409048201
iteration 1400 / 2000: loss 1.8868518148136717
iteration 1500 / 2000: loss 1.7840075159616509
iteration 1600 / 2000: loss 1.8693980236708188
iteration 1700 / 2000: loss 1.7751991265474192
iteration 1800 / 2000: loss 1.8070933004089762
iteration 1900 / 2000: loss 1.864992559154446
Validation accuracy:  0.372
iteration 0 / 2000: loss 2.302799564943597
iteration 100 / 2000: loss 2.3023869823026293
iteration 200 / 2000: loss 2.297139420278263
iteration 300 / 2000: loss 2.264917011452694
iteration 400 / 2000: loss 2.209375864288572
iteration 500 / 2000: loss 2.177577122509616
iteration 600 / 2000: loss 2.125845495953006
iteration 700 / 2000: loss 2.034526606554556
iteration 800 / 2000: loss 2.076498454126965
iteration 900 / 2000: loss 2.0551660991486145
iteration 1000 / 2000: loss 1.9697511679015525
iteration 1100 / 2000: loss 1.9997651335057274
iteration 1200 / 2000: loss 1.8951595743131804
iteration 1300 / 2000: loss 1.8250813932297483
iteration 1400 / 2000: loss 1.8353035094007146
iteration 1500 / 2000: loss 1.8567584853557917
iteration 1600 / 2000: loss 1.863186742800982
iteration 1700 / 2000: loss 1.8900423884818434
iteration 1800 / 2000: loss 1.78332296158417
iteration 1900 / 2000: loss 1.8392051659932935
Validation accuracy:  0.358
iteration 0 / 2000: loss 2.3027982182966538
iteration 100 / 2000: loss 2.3023698709049163
iteration 200 / 2000: loss 2.2993439539033025
iteration 300 / 2000: loss 2.2750032074098585
iteration 400 / 2000: loss 2.180231513142964
iteration 500 / 2000: loss 2.147392705150913
iteration 600 / 2000: loss 2.0234361107583
iteration 700 / 2000: loss 2.0105469925797412
iteration 800 / 2000: loss 2.0465271124310824
iteration 900 / 2000: loss 1.848345936846603
iteration 1000 / 2000: loss 1.8795276543151933
iteration 1100 / 2000: loss 1.8902183732303672
iteration 1200 / 2000: loss 1.8560224699145318
iteration 1300 / 2000: loss 1.8487832298146913
iteration 1400 / 2000: loss 1.783028746993995
iteration 1500 / 2000: loss 1.8982608593795733
iteration 1600 / 2000: loss 1.8743645820507342
```

```
iteration 1700 / 2000: loss 1.7500160196697316
iteration 1800 / 2000: loss 1.796837441138722
iteration 1900 / 2000: loss 1.6493213266088032
Validation accuracy:  0.371
iteration 0 / 2000: loss 2.3028978193730842
iteration 100 / 2000: loss 2.3023852570806747
iteration 200 / 2000: loss 2.2977123086291713
iteration 300 / 2000: loss 2.2522312225617913
iteration 400 / 2000: loss 2.1980961273939292
iteration 500 / 2000: loss 2.1173644431379532
iteration 600 / 2000: loss 2.0458970473302864
iteration 700 / 2000: loss 2.068843812082815
iteration 800 / 2000: loss 2.0156597405353205
iteration 900 / 2000: loss 1.9210502631707702
iteration 1000 / 2000: loss 1.9922436022065673
iteration 1100 / 2000: loss 1.8903934716665933
iteration 1200 / 2000: loss 1.941859028172481
iteration 1300 / 2000: loss 1.889159451837128
iteration 1400 / 2000: loss 1.8797006057735208
iteration 1500 / 2000: loss 1.9355453042564608
iteration 1600 / 2000: loss 1.8200473048114252
iteration 1700 / 2000: loss 1.8863119341833432
iteration 1800 / 2000: loss 1.7881680849687707
iteration 1900 / 2000: loss 1.8163404526660296
Validation accuracy:  0.348
iteration 0 / 2000: loss 2.302888005546075
iteration 100 / 2000: loss 2.3023875295107556
iteration 200 / 2000: loss 2.296938197883292
iteration 300 / 2000: loss 2.2658842141760696
iteration 400 / 2000: loss 2.1748543667014193
iteration 500 / 2000: loss 2.088674616728459
iteration 600 / 2000: loss 2.0546242491168707
iteration 700 / 2000: loss 1.988055738961801
iteration 800 / 2000: loss 1.954490823915326
iteration 900 / 2000: loss 1.9639159131812305
iteration 1000 / 2000: loss 1.9833585199246302
iteration 1100 / 2000: loss 1.8964180242968296
iteration 1200 / 2000: loss 1.9226304701585566
iteration 1300 / 2000: loss 1.8225280617238688
iteration 1400 / 2000: loss 1.8638862151165816
iteration 1500 / 2000: loss 1.8064280955619598
iteration 1600 / 2000: loss 1.8089178563726072
iteration 1700 / 2000: loss 1.746982345975569
iteration 1800 / 2000: loss 1.7173059153192685
iteration 1900 / 2000: loss 1.7825299632464597
Validation accuracy:  0.371
iteration 0 / 2000: loss 2.3026703337607315
iteration 100 / 2000: loss 2.1006194008841854
iteration 200 / 2000: loss 1.833332031732493
iteration 300 / 2000: loss 1.8162515484693034
iteration 400 / 2000: loss 1.7072125498636197
iteration 500 / 2000: loss 1.6780759841537398
iteration 600 / 2000: loss 1.621049459760123
```

```
iteration 700 / 2000: loss 1.6590667138775477
iteration 800 / 2000: loss 1.6155502565599005
iteration 900 / 2000: loss 1.608501899656894
iteration 1000 / 2000: loss 1.615273570076806
iteration 1100 / 2000: loss 1.4762538377620673
iteration 1200 / 2000: loss 1.5067176348560816
iteration 1300 / 2000: loss 1.4247755945820122
iteration 1400 / 2000: loss 1.4848853658762793
iteration 1500 / 2000: loss 1.522756069983809
iteration 1600 / 2000: loss 1.416412050435365
iteration 1700 / 2000: loss 1.4212926656814284
iteration 1800 / 2000: loss 1.4541477545166386
iteration 1900 / 2000: loss 1.5265187542859988
Validation accuracy:  0.47
iteration 0 / 2000: loss 2.3026556583004814
iteration 100 / 2000: loss 2.1190928228143227
iteration 200 / 2000: loss 1.9296393589766991
iteration 300 / 2000: loss 1.8367627224835885
iteration 400 / 2000: loss 1.7575871347735352
iteration 500 / 2000: loss 1.6344325395648824
iteration 600 / 2000: loss 1.6388509926127353
iteration 700 / 2000: loss 1.7214324423699192
iteration 800 / 2000: loss 1.4669961452556366
iteration 900 / 2000: loss 1.5232386235602056
iteration 1000 / 2000: loss 1.5204536583191368
iteration 1100 / 2000: loss 1.6189431881031369
iteration 1200 / 2000: loss 1.4995550648855676
iteration 1300 / 2000: loss 1.4585460917483497
iteration 1400 / 2000: loss 1.562789421276314
iteration 1500 / 2000: loss 1.5015549388258194
iteration 1600 / 2000: loss 1.4452733833785287
iteration 1700 / 2000: loss 1.5426953308548792
iteration 1800 / 2000: loss 1.4702685343101174
iteration 1900 / 2000: loss 1.5095911608565218
Validation accuracy:  0.479
iteration 0 / 2000: loss 2.302769593676105
iteration 100 / 2000: loss 2.0937514433461817
iteration 200 / 2000: loss 1.9749814423927858
iteration 300 / 2000: loss 1.8857096203225479
iteration 400 / 2000: loss 1.7680419800998703
iteration 500 / 2000: loss 1.7519884238335774
iteration 600 / 2000: loss 1.6666685809400164
iteration 700 / 2000: loss 1.6794552394109659
iteration 800 / 2000: loss 1.6715766149853066
iteration 900 / 2000: loss 1.6004736782877718
iteration 1000 / 2000: loss 1.5578778169980922
iteration 1100 / 2000: loss 1.5423998301484476
iteration 1200 / 2000: loss 1.4985233982846187
iteration 1300 / 2000: loss 1.5024479521953586
iteration 1400 / 2000: loss 1.5419861946810116
iteration 1500 / 2000: loss 1.511197435836202
iteration 1600 / 2000: loss 1.590932832683944
iteration 1700 / 2000: loss 1.6272660702135942
```

```
iteration 1800 / 2000: loss 1.4711426508605037
iteration 1900 / 2000: loss 1.4807843477975213
Validation accuracy:  0.459
iteration 0 / 2000: loss 2.302761032124232
iteration 100 / 2000: loss 2.113555355970943
iteration 200 / 2000: loss 1.903271059074443
iteration 300 / 2000: loss 1.8767338857295646
iteration 400 / 2000: loss 1.7828181129024196
iteration 500 / 2000: loss 1.678836785706999
iteration 600 / 2000: loss 1.6859463728991464
iteration 700 / 2000: loss 1.585930597282966
iteration 800 / 2000: loss 1.660420580934692
iteration 900 / 2000: loss 1.6123157169603735
iteration 1000 / 2000: loss 1.4583448092649332
iteration 1100 / 2000: loss 1.5763122538623155
iteration 1200 / 2000: loss 1.5202431435594477
iteration 1300 / 2000: loss 1.6291385010052108
iteration 1400 / 2000: loss 1.4675088357339858
iteration 1500 / 2000: loss 1.5415850448391384
iteration 1600 / 2000: loss 1.4676130286287428
iteration 1700 / 2000: loss 1.5603782848355925
iteration 1800 / 2000: loss 1.586361150797385
iteration 1900 / 2000: loss 1.3967115108389225
Validation accuracy:  0.493
iteration 0 / 2000: loss 2.302906036873376
iteration 100 / 2000: loss 2.1342731585120283
iteration 200 / 2000: loss 1.9854947401969683
iteration 300 / 2000: loss 1.8719524804205467
iteration 400 / 2000: loss 1.7411005674119127
iteration 500 / 2000: loss 1.7952201689198206
iteration 600 / 2000: loss 1.717228116395628
iteration 700 / 2000: loss 1.615758859325861
iteration 800 / 2000: loss 1.6998946178968481
iteration 900 / 2000: loss 1.6024471435864014
iteration 1000 / 2000: loss 1.5618178319790863
iteration 1100 / 2000: loss 1.6080234088986303
iteration 1200 / 2000: loss 1.5991329207378915
iteration 1300 / 2000: loss 1.596740768545239
iteration 1400 / 2000: loss 1.4918984063724798
iteration 1500 / 2000: loss 1.658078010371052
iteration 1600 / 2000: loss 1.4424523599695847
iteration 1700 / 2000: loss 1.5377960438476144
iteration 1800 / 2000: loss 1.5498063000567666
iteration 1900 / 2000: loss 1.4969536421776075
Validation accuracy:  0.471
iteration 0 / 2000: loss 2.302882410611367
iteration 100 / 2000: loss 2.1298170694370198
iteration 200 / 2000: loss 1.9444813432876888
iteration 300 / 2000: loss 1.794311564247586
iteration 400 / 2000: loss 1.7689859782878754
iteration 500 / 2000: loss 1.7207724684871948
iteration 600 / 2000: loss 1.602988160814055
iteration 700 / 2000: loss 1.6568825080601937
```

```
iteration 800 / 2000: loss 1.662393778817383
iteration 900 / 2000: loss 1.6531417344413968
iteration 1000 / 2000: loss 1.5892984417300078
iteration 1100 / 2000: loss 1.4455683446585406
iteration 1200 / 2000: loss 1.4460926847937803
iteration 1300 / 2000: loss 1.5519623019519833
iteration 1400 / 2000: loss 1.4226182254048165
iteration 1500 / 2000: loss 1.4784345412014848
iteration 1600 / 2000: loss 1.3732803274923513
iteration 1700 / 2000: loss 1.4716089992235128
iteration 1800 / 2000: loss 1.4356650079743953
iteration 1900 / 2000: loss 1.488135332278079
Validation accuracy:  0.488
iteration 0 / 2000: loss 2.3026678917901044
iteration 100 / 2000: loss 1.8866342436855181
iteration 200 / 2000: loss 1.7739965586415216
iteration 300 / 2000: loss 1.6232175874797306
iteration 400 / 2000: loss 1.6402428315563329
iteration 500 / 2000: loss 1.663480495398933
iteration 600 / 2000: loss 1.5414818483607353
iteration 700 / 2000: loss 1.461367539289358
iteration 800 / 2000: loss 1.4357201540667244
iteration 900 / 2000: loss 1.4569621180520624
iteration 1000 / 2000: loss 1.4179901038083633
iteration 1100 / 2000: loss 1.320138027715432
iteration 1200 / 2000: loss 1.4585151479796692
iteration 1300 / 2000: loss 1.3689851593509967
iteration 1400 / 2000: loss 1.3103200045508516
iteration 1500 / 2000: loss 1.3562543578596764
iteration 1600 / 2000: loss 1.4598346475625996
iteration 1700 / 2000: loss 1.3025924916357692
iteration 1800 / 2000: loss 1.3302588093962262
iteration 1900 / 2000: loss 1.2553361063590387
Validation accuracy:  0.489
iteration 0 / 2000: loss 2.3026327213386635
iteration 100 / 2000: loss 1.9103815235046195
iteration 200 / 2000: loss 1.74935159980751
iteration 300 / 2000: loss 1.719308568809453
iteration 400 / 2000: loss 1.6196612149509892
iteration 500 / 2000: loss 1.5688266954748722
iteration 600 / 2000: loss 1.6174678773853792
iteration 700 / 2000: loss 1.4339910513936966
iteration 800 / 2000: loss 1.5670262943701492
iteration 900 / 2000: loss 1.39687037873628
iteration 1000 / 2000: loss 1.4595039307468725
iteration 1100 / 2000: loss 1.4454593611830129
iteration 1200 / 2000: loss 1.4062195630887384
iteration 1300 / 2000: loss 1.4599754010932329
iteration 1400 / 2000: loss 1.5554688354871262
iteration 1500 / 2000: loss 1.310071256866195
iteration 1600 / 2000: loss 1.4264742849651755
iteration 1700 / 2000: loss 1.2654472011927806
iteration 1800 / 2000: loss 1.42419271668573
```

```
iteration 1900 / 2000: loss 1.260836194157699
Validation accuracy:  0.481
iteration 0 / 2000: loss 2.302749783217943
iteration 100 / 2000: loss 1.9926744084278898
iteration 200 / 2000: loss 1.718662929024807
iteration 300 / 2000: loss 1.690147029852461
iteration 400 / 2000: loss 1.6545376705332056
iteration 500 / 2000: loss 1.52453115279471
iteration 600 / 2000: loss 1.482365849199233
iteration 700 / 2000: loss 1.4131632175116309
iteration 800 / 2000: loss 1.6758641543994663
iteration 900 / 2000: loss 1.453992715447181
iteration 1000 / 2000: loss 1.4363761730316231
iteration 1100 / 2000: loss 1.461639535912317
iteration 1200 / 2000: loss 1.4648649585482134
iteration 1300 / 2000: loss 1.408766111257291
iteration 1400 / 2000: loss 1.5092029895443828
iteration 1500 / 2000: loss 1.3360878145759942
iteration 1600 / 2000: loss 1.4263724787675294
iteration 1700 / 2000: loss 1.45539747328614
iteration 1800 / 2000: loss 1.4884738429374424
iteration 1900 / 2000: loss 1.467273059782856
Validation accuracy:  0.514
iteration 0 / 2000: loss 2.3027696793754586
iteration 100 / 2000: loss 1.9469487983013873
iteration 200 / 2000: loss 1.7618957972929734
iteration 300 / 2000: loss 1.6351092848222832
iteration 400 / 2000: loss 1.6365082974996512
iteration 500 / 2000: loss 1.6385923432990899
iteration 600 / 2000: loss 1.571423151338519
iteration 700 / 2000: loss 1.5199343583762857
iteration 800 / 2000: loss 1.435468416643961
iteration 900 / 2000: loss 1.4408037772468087
iteration 1000 / 2000: loss 1.5427469076776765
iteration 1100 / 2000: loss 1.4918033498856582
iteration 1200 / 2000: loss 1.3869399751477933
iteration 1300 / 2000: loss 1.4266588722695257
iteration 1400 / 2000: loss 1.4434218951832363
iteration 1500 / 2000: loss 1.358783391641609
iteration 1600 / 2000: loss 1.337154457994172
iteration 1700 / 2000: loss 1.4178629310787763
iteration 1800 / 2000: loss 1.3287076525649573
iteration 1900 / 2000: loss 1.3867696782198868
Validation accuracy:  0.486
iteration 0 / 2000: loss 2.302925600211157
iteration 100 / 2000: loss 1.9289666821662859
iteration 200 / 2000: loss 1.7351272572477972
iteration 300 / 2000: loss 1.6490370102804037
iteration 400 / 2000: loss 1.565216783027131
iteration 500 / 2000: loss 1.6641966804997874
iteration 600 / 2000: loss 1.5397986668832415
iteration 700 / 2000: loss 1.549961896484121
iteration 800 / 2000: loss 1.4824710620971537
```

```
iteration 900 / 2000: loss 1.4447354184280914
iteration 1000 / 2000: loss 1.6265044457222126
iteration 1100 / 2000: loss 1.616429372462132
iteration 1200 / 2000: loss 1.4554086535178614
iteration 1300 / 2000: loss 1.362949409354585
iteration 1400 / 2000: loss 1.5273077099903272
iteration 1500 / 2000: loss 1.416610832435543
iteration 1600 / 2000: loss 1.3286318962780994
iteration 1700 / 2000: loss 1.5030851518138135
iteration 1800 / 2000: loss 1.3752516679697828
iteration 1900 / 2000: loss 1.4598401706948005
Validation accuracy:  0.496
iteration 0 / 2000: loss 2.3029010995748043
iteration 100 / 2000: loss 1.9499910494880959
iteration 200 / 2000: loss 1.8206974703953571
iteration 300 / 2000: loss 1.7005852388533853
iteration 400 / 2000: loss 1.6319368088664723
iteration 500 / 2000: loss 1.6987121095739066
iteration 600 / 2000: loss 1.530709097201139
iteration 700 / 2000: loss 1.5736126541858648
iteration 800 / 2000: loss 1.539846029443171
iteration 900 / 2000: loss 1.4981004096495687
iteration 1000 / 2000: loss 1.3805893192796665
iteration 1100 / 2000: loss 1.4278531056682122
iteration 1200 / 2000: loss 1.411706692019949
iteration 1300 / 2000: loss 1.4407103085312853
iteration 1400 / 2000: loss 1.4856568325519879
iteration 1500 / 2000: loss 1.4506430222784583
iteration 1600 / 2000: loss 1.4179618072790827
iteration 1700 / 2000: loss 1.4944280632664713
iteration 1800 / 2000: loss 1.3830344114249353
iteration 1900 / 2000: loss 1.3210757147241752
Validation accuracy:  0.499
iteration 0 / 2000: loss 2.302655099306456
iteration 100 / 2000: loss 1.8025253309138118
iteration 200 / 2000: loss 1.6208365009777341
iteration 300 / 2000: loss 1.606808292732267
iteration 400 / 2000: loss 1.6504366636018897
iteration 500 / 2000: loss 1.735689847434799
iteration 600 / 2000: loss 1.6381560387008147
iteration 700 / 2000: loss 1.6908903433528466
iteration 800 / 2000: loss 1.4039474253477482
iteration 900 / 2000: loss 1.448453086159831
iteration 1000 / 2000: loss 1.65708944602311
iteration 1100 / 2000: loss 1.481233729064637
iteration 1200 / 2000: loss 1.3949567158002536
iteration 1300 / 2000: loss 1.445690980152747
iteration 1400 / 2000: loss 1.3780015384866635
iteration 1500 / 2000: loss 1.476025524632465
iteration 1600 / 2000: loss 1.378335926586148
iteration 1700 / 2000: loss 1.5145134708994736
iteration 1800 / 2000: loss 1.2724740211506562
iteration 1900 / 2000: loss 1.4997946575256362
```

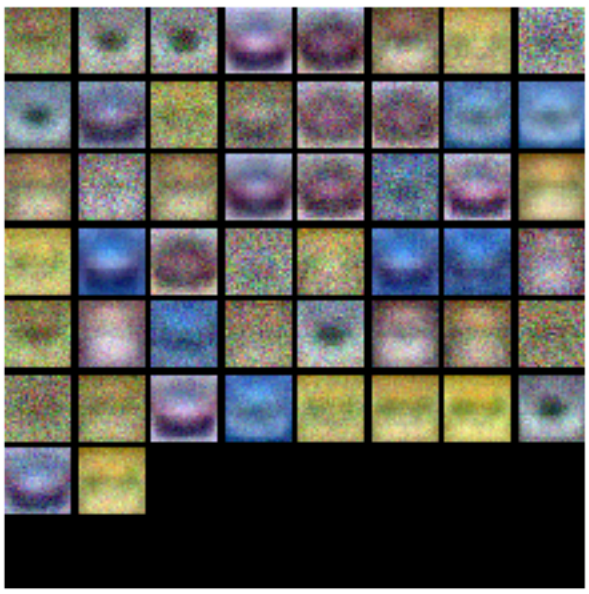
```
Validation accuracy:  0.498
iteration 0 / 2000: loss 2.3026672698648603
iteration 100 / 2000: loss 1.7679852098789663
iteration 200 / 2000: loss 1.7014062117075677
iteration 300 / 2000: loss 1.6527762957841885
iteration 400 / 2000: loss 1.4959835261408234
iteration 500 / 2000: loss 1.53871005429552
iteration 600 / 2000: loss 1.6480846795442703
iteration 700 / 2000: loss 1.7973246767876063
iteration 800 / 2000: loss 1.5527413114844268
iteration 900 / 2000: loss 1.4954074392719932
iteration 1000 / 2000: loss 1.622027793934723
iteration 1100 / 2000: loss 1.6157512321769893
iteration 1200 / 2000: loss 1.496859170751847
iteration 1300 / 2000: loss 1.475715351574424
iteration 1400 / 2000: loss 1.5379811480563763
iteration 1500 / 2000: loss 1.5691368758486426
iteration 1600 / 2000: loss 1.4186761560806418
iteration 1700 / 2000: loss 1.4550510688717604
iteration 1800 / 2000: loss 1.555444397931136
iteration 1900 / 2000: loss 1.6219125136940518
Validation accuracy:  0.434
iteration 0 / 2000: loss 2.3027851326198214
iteration 100 / 2000: loss 1.8327105363206666
iteration 200 / 2000: loss 1.7559888869736378
iteration 300 / 2000: loss 1.5162422399624997
iteration 400 / 2000: loss 1.5916544733450173
iteration 500 / 2000: loss 1.6100617160218427
iteration 600 / 2000: loss 1.6170515609516565
iteration 700 / 2000: loss 1.4571305222143736
iteration 800 / 2000: loss 1.4948409887024559
iteration 900 / 2000: loss 1.5720040586681943
iteration 1000 / 2000: loss 1.6002905420561246
iteration 1100 / 2000: loss 1.7840219655880722
iteration 1200 / 2000: loss 1.5575227463080763
iteration 1300 / 2000: loss 1.5750938797663872
iteration 1400 / 2000: loss 1.577625756884463
iteration 1500 / 2000: loss 1.6134702619723984
iteration 1600 / 2000: loss 1.4462163670035066
iteration 1700 / 2000: loss 1.4789890876971004
iteration 1800 / 2000: loss 1.3785208969082574
iteration 1900 / 2000: loss 1.4505786269746885
Validation accuracy:  0.489
iteration 0 / 2000: loss 2.3027685358917043
iteration 100 / 2000: loss 1.9273854802991242
iteration 200 / 2000: loss 1.6641320370290624
iteration 300 / 2000: loss 1.5964258529577018
iteration 400 / 2000: loss 1.7403410513432842
iteration 500 / 2000: loss 1.6136360542136732
iteration 600 / 2000: loss 1.7509801813677688
iteration 700 / 2000: loss 1.7837791511262138
iteration 800 / 2000: loss 1.4972282507126151
iteration 900 / 2000: loss 1.679724799150232
```

```
iteration 1000 / 2000: loss 1.6400023357845464
iteration 1100 / 2000: loss 1.6226175872009094
iteration 1200 / 2000: loss 1.5519933204912812
iteration 1300 / 2000: loss 1.4675184178508904
iteration 1400 / 2000: loss 1.7094373792349884
iteration 1500 / 2000: loss 1.5049894653639557
iteration 1600 / 2000: loss 1.5891786021800183
iteration 1700 / 2000: loss 1.5532656560961648
iteration 1800 / 2000: loss 1.5015679696884972
iteration 1900 / 2000: loss 1.7035796977140785
Validation accuracy:  0.438
iteration 0 / 2000: loss 2.302877990525955
iteration 100 / 2000: loss 1.7039397013957664
iteration 200 / 2000: loss 1.7596840909930616
iteration 300 / 2000: loss 1.830229335931259
iteration 400 / 2000: loss 1.6968757678545194
iteration 500 / 2000: loss 1.610108215979609
iteration 600 / 2000: loss 1.5601305559390006
iteration 700 / 2000: loss 1.6214469165779883
iteration 800 / 2000: loss 1.5698168171516365
iteration 900 / 2000: loss 1.6718575826476696
iteration 1000 / 2000: loss 1.5503313320343954
iteration 1100 / 2000: loss 1.5321968816506812
iteration 1200 / 2000: loss 1.4372851079338407
iteration 1300 / 2000: loss 1.5151293003910153
iteration 1400 / 2000: loss 1.4486001221769966
iteration 1500 / 2000: loss 1.504635476130036
iteration 1600 / 2000: loss 1.4480466931804028
iteration 1700 / 2000: loss 1.6806679452037991
iteration 1800 / 2000: loss 1.4512042257545048
iteration 1900 / 2000: loss 1.5475752376777572
Validation accuracy:  0.475
iteration 0 / 2000: loss 2.3028756686779617
iteration 100 / 2000: loss 1.7838946730643803
iteration 200 / 2000: loss 1.9200066436701255
iteration 300 / 2000: loss 1.7168225414791494
iteration 400 / 2000: loss 1.8812433562861188
iteration 500 / 2000: loss 1.5652470273424397
iteration 600 / 2000: loss 1.6762844273776856
iteration 700 / 2000: loss 1.8686171835020846
iteration 800 / 2000: loss 1.5872870352739692
iteration 900 / 2000: loss 1.8046222848194504
iteration 1000 / 2000: loss 1.7046852792877536
iteration 1100 / 2000: loss 1.6468283749432218
iteration 1200 / 2000: loss 1.553537296548796
iteration 1300 / 2000: loss 1.7365148592763244
iteration 1400 / 2000: loss 1.5292940630920024
iteration 1500 / 2000: loss 1.524713579905912
iteration 1600 / 2000: loss 1.571472770460326
iteration 1700 / 2000: loss 1.597353843574408
iteration 1800 / 2000: loss 1.3981689310365524
iteration 1900 / 2000: loss 1.4135260143970703
Validation accuracy:  0.457
```

```
Best net:  <nndl.neural_net.TwoLayerNet object at 0x117e58668>
Validation accuracy with the best net:  0.514
```

In [14]:

```
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)
```



Question:

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

Answer:

(1) The best net's weights look more colorful, informative and distinct from each other, while suboptimal net's weights look noisy and monotonous.

Evaluate on test set

In [15]:

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

```
Test accuracy:  0.501
```

```
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.
    # ============================================================== #
#     X.shape = num_inputs N = 5, input_size D = 4
#     W1.shape = hidden_size H = 10, input_size D
#     b1.shape = H
#     W2.shape = C, H
#     b2.shape = C = 3

#     print(X.shape) = (5,4)
#     print(W1.shape) = (10,4)

    h1 = np.dot(X, W1.T) + b1 #shape = (N, H)
    f = lambda x: x*(x>0)
    h1 = f(h1)
    scores = np.dot(h1, W2.T) + b2 #shape = (N, C)


    pass
```

```
    # ============================================================ #
    # 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).
    # ============================================================ #
    exp_scores = np.exp(scores) # shape = (N, C)
    sum_class_scores = np.sum(exp_scores, axis = 1, keepdims = 1) # shape = (N, 1)
    log_exp_scores = np.log(exp_scores[range(N), y]/sum_class_scores) # shape = (N,
     C)
    data_loss = -np.mean(log_exp_scores)
    reg_loss = 0.5 * reg * (np.sum(W1*W1) + np.sum(W2*W2))
    loss = data_loss + reg_loss


    # scores is num_examples by num_classes
    pass
    # ============================================================ #
    # 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.
    # ============================================================ #

#    loss: softmax entropy loss
#    firstly calculate dloss/dscores: (scores is the output layer: W2X + b2
    dscores = exp_scores/sum_class_scores #(N, C)
    dscores[range(N), y] -= 1
    dscores /= N

#    X.shape = num_inputs N = 5, input_size D = 4
#    W1.shape = hidden_size H = 10, input_size D
#    b1.shape = H
```

```
#    W2.shape = C, H
#    b2.shape = C = 3
#    h1.shape = N, H
#    dscores.shape = N, C

    grads['W2'] = np.dot(dscores.T, h1) #(C, H)
    grads['W2'] += reg*W2
    grads['b2'] = np.sum(dscores, axis = 0) #(C, )

    dloss_h1 = np.dot(dscores, W2) #(N, C)*(C, H) = (N, H)
    dloss_h1[h1<=0] = 0
    grads['W1'] = np.dot(dloss_h1.T, X)
    grads['W1'] += reg*W1
    grads['b1'] = np.sum(dloss_h1, axis = 0)


    pass

    # ============================================================= #
    # 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

  # ========================================================== #
  # YOUR CODE HERE:
  #   Create a minibatch by sampling batch_size samples randomly.
  # ========================================================== #
  pass
  idx = np.random.choice(num_train, batch_size)
  X_batch = X[idx]
  y_batch = y[idx]
  # ========================================================== #
  # 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']

  pass

  # ========================================================== #
  # 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.
  # ================================================================ #
  pass
  h1 = np.dot(X, self.params['W1'].T) + self.params['b1'] #shape = (N, H)
  f = lambda x: x*(x>0)
  h1 = f(h1)
  scores = np.dot(h1, self.params['W2'].T) + self.params['b2'] #shape = (N, C)

  y_pred = np.argmax(scores, axis=1)

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

  return y_pred
```

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).

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 [1]:

```
## 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_a
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

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

In [2]:

```
# Load the (preprocessed) CIFAR10 data.

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

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

Linear layers

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

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

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

Affine layer forward pass

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

In [3]:

```
# Test the affine_forward function

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

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

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

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

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

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

Affine layer backward pass

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

In [4]:

```
# 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: 4.5914190448782733e-10
dw error: 2.4148923741522363e-09
db error: 3.2757501498359554e-12
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

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

In [5]:

```
# Test the relu_forward function

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

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

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

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

ReLU backward pass

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

In [6]:

```
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: 3.2756047436557943e-12
```

Combining the affine and ReLU layers

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

Affine-ReLU layers

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

In [7]:

```
from nndl.layer_utils import affine_relu_forward, affine_relu_backward

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

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

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

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

```
Testing affine_relu_forward and affine_relu_backward:
dx error: 1.5611571526437724e-09
dw error: 9.312161930854077e-11
db error: 9.575508047575714e-12
```

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 [8]:

```
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: 8.999468743403236
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.302532430200951
dx error: 8.814442861032232e-09
```

Implementation of a two-layer NN

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

In [9]:

```
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, 15.3320676
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.4999413
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.6678150
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

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

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

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.521570767795567e-08
W2 relative error: 3.123284996414637e-10
b1 relative error: 6.548544286385526e-09
b2 relative error: 1.338456048335764e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.527915286171985e-07
W2 relative error: 1.3678369558053052e-07
b1 relative error: 1.5646802105012178e-08
b2 relative error: 9.089615724390711e-10
```

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 50%.

In [10]:

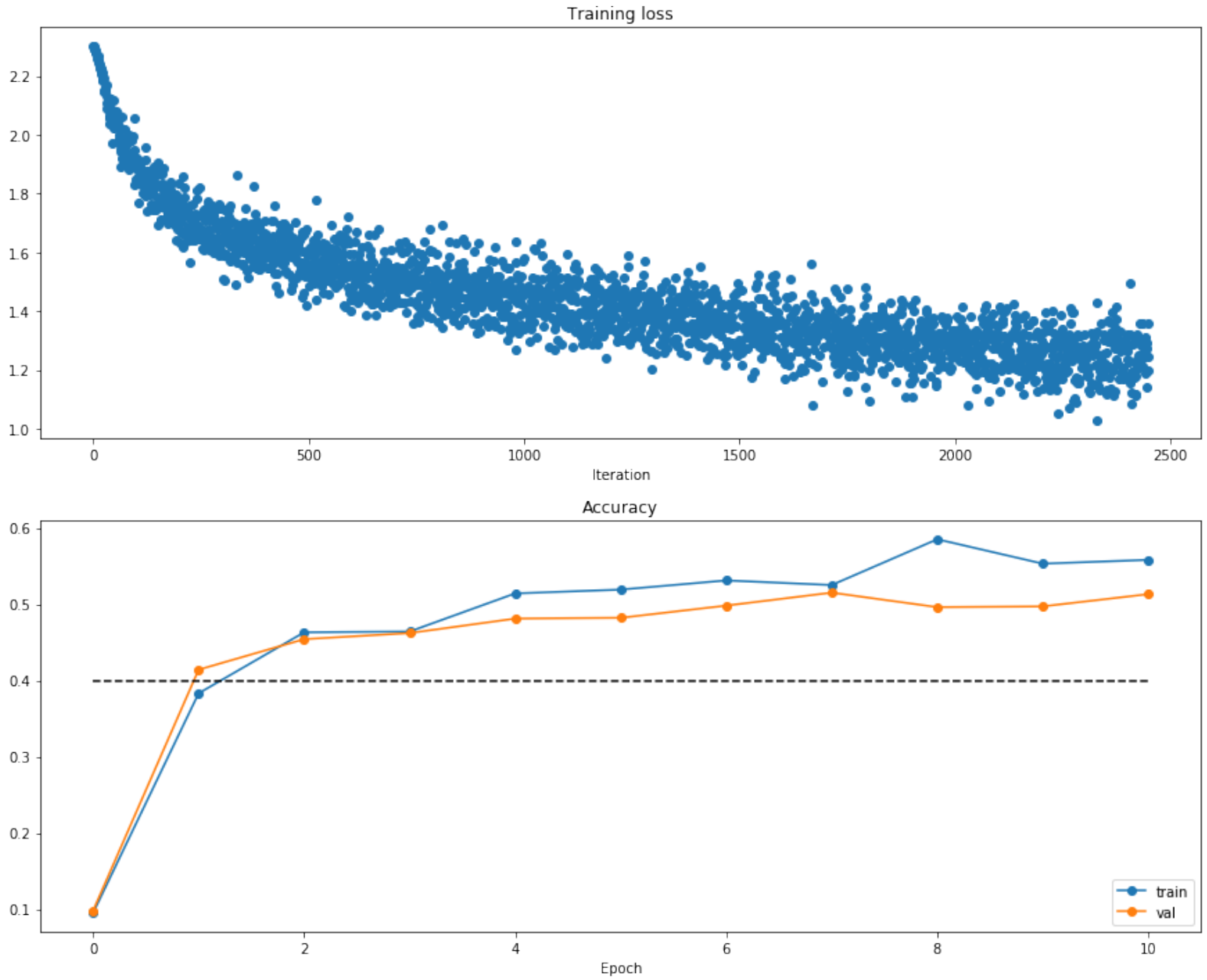
```
model = TwoLayerNet(hidden_dims = 200)
solver = None

# ================================================================ #
# 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.
# ================================================================ #
best_val = -1
# results = {}
np.random.seed(0)

# bs = [200]
# lr = [5e-4]
# lr_decay = [0.95]

# grid_search = [(x,y,h) for x in batch_sizes for y in lrs for h in lr_decays]

X_train = data['X_train']
y_train = data['y_train']
X_val = data['X_val']
y_val = data['y_val']

# for bs, lr, lr_decay in grid_search:
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={'learning_rate': 5e-4,},
                lr_decay=0.95,
                num_epochs=10, batch_size=200,
                print_every=100)
solver.train()
val_acc = solver.check_accuracy(X_val, y_val)

print('Validation accuracy: ', val_acc)
print('-'*30)
# results[(bs,lr,lr_decay)] = val_acc

if val_acc > best_val:
    best_val = val_acc
    best_solver = solver
# print('Best net: ', best_net)
# print('Best validation accuracy: ', best_val)
```

```
# print('Best validation accuracy: ', best_val)
pass

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

```
(Iteration 1 / 2450) loss: 2.299874
(Epoch 0 / 10) train acc: 0.095000; val_acc: 0.098000
(Iteration 101 / 2450) loss: 1.842674
(Iteration 201 / 2450) loss: 1.834596
(Epoch 1 / 10) train acc: 0.383000; val_acc: 0.414000
(Iteration 301 / 2450) loss: 1.749186
(Iteration 401 / 2450) loss: 1.662587
(Epoch 2 / 10) train acc: 0.463000; val_acc: 0.454000
(Iteration 501 / 2450) loss: 1.602733
(Iteration 601 / 2450) loss: 1.561971
(Iteration 701 / 2450) loss: 1.452643
(Epoch 3 / 10) train acc: 0.464000; val_acc: 0.462000
(Iteration 801 / 2450) loss: 1.451737
(Iteration 901 / 2450) loss: 1.537457
(Epoch 4 / 10) train acc: 0.514000; val_acc: 0.481000
(Iteration 1001 / 2450) loss: 1.578281
(Iteration 1101 / 2450) loss: 1.595664
(Iteration 1201 / 2450) loss: 1.339042
(Epoch 5 / 10) train acc: 0.519000; val_acc: 0.482000
(Iteration 1301 / 2450) loss: 1.342453
(Iteration 1401 / 2450) loss: 1.333728
(Epoch 6 / 10) train acc: 0.531000; val_acc: 0.498000
(Iteration 1501 / 2450) loss: 1.469659
(Iteration 1601 / 2450) loss: 1.363826
(Iteration 1701 / 2450) loss: 1.327690
(Epoch 7 / 10) train acc: 0.525000; val_acc: 0.515000
(Iteration 1801 / 2450) loss: 1.094912
(Iteration 1901 / 2450) loss: 1.111161
(Epoch 8 / 10) train acc: 0.585000; val_acc: 0.496000
(Iteration 2001 / 2450) loss: 1.305439
(Iteration 2101 / 2450) loss: 1.319213
(Iteration 2201 / 2450) loss: 1.255064
(Epoch 9 / 10) train acc: 0.553000; val_acc: 0.497000
(Iteration 2301 / 2450) loss: 1.209555
(Iteration 2401 / 2450) loss: 1.236897
(Epoch 10 / 10) train acc: 0.558000; val_acc: 0.513000
Validation accuracy:  0.515
------------------------------
```

In [11]:

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



Multilayer Neural Network

Now, we implement a multi-layer neural network.

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

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

In [12]:

```
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: 2.300194602354735
W1 relative error: 1.8253622527384e-06
W2 relative error: 5.349696978095353e-06
W3 relative error: 1.2540351993686 59e-07
b1 relative error: 1.0859451411398325e-07
b2 relative error: 2.068454630220133e-09
b3 relative error: 9.960970638086798e-11
Running check with reg = 3.14
Initial loss: 6.801083587522617
W1 relative error: 2.486852099840146e-08
W2 relative error: 1.7628806831800763e-05
W3 relative error: 6.332216597030167e-08
b1 relative error: 6.846754327000344e-08
b2 relative error: 2.809562438546381e-09
b3 relative error: 2.1689236456139844e-10
```

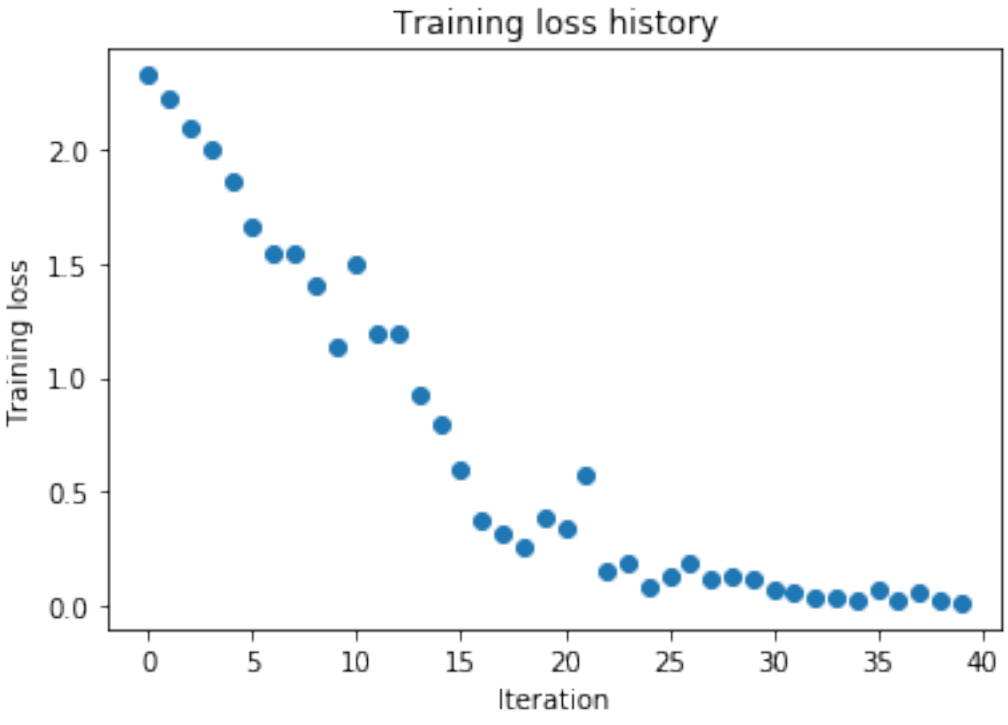
In [13]:

```
# 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 sma
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 1e-2
learning_rate = 1e-2

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

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

```
(Iteration 1 / 40) loss: 2.324190
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.119000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.106000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.137000
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.147000
(Epoch 4 / 20) train acc: 0.660000; val_acc: 0.169000
(Epoch 5 / 20) train acc: 0.500000; val_acc: 0.156000
(Iteration 11 / 40) loss: 1.494306
(Epoch 6 / 20) train acc: 0.640000; val_acc: 0.134000
(Epoch 7 / 20) train acc: 0.780000; val_acc: 0.146000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.168000
(Epoch 9 / 20) train acc: 0.900000; val_acc: 0.187000
(Epoch 10 / 20) train acc: 0.940000; val_acc: 0.204000
(Iteration 21 / 40) loss: 0.341710
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.185000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.181000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.194000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.205000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.197000
(Iteration 31 / 40) loss: 0.069791
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.172000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.179000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.186000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.192000
```

```
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.196000
```



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)
    # ============================================================ #

    pass
    self.params['W1'] = weight_scale * np.random.randn(input_dim, hidden_dims)
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = weight_scale * 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.
    # ============================================================ #
    N = X.shape[0]
    D = np.prod(X.shape[1:])
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    h1, h1_cache = affine_relu_forward(X, W1, b1)
    scores, scores_cache = affine_forward(h1, W2, b2)




    pass
    # ============================================================ #
    # 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.
    # ============================================================ #
    exp_scores = np.exp(scores) # shape = (N, C)
    sum_class_scores = np.sum(exp_scores, axis = 1, keepdims = 1) # shape = (N, 1)
    log_exp_scores = np.log(exp_scores[range(N), y]/sum_class_scores) # shape = (N,
     C)
    data_loss = -np.mean(log_exp_scores)
    reg_loss = 0.5 * self.reg * (np.sum(W1*W1)+np.sum(W2*W2))
    loss = data_loss + reg_loss

    dscores = exp_scores/sum_class_scores #(N, C)
    dscores[range(N), y] -= 1
    dscores /= N

    dloss_h1, grads['W2'], grads['b2'] = affine_backward(dscores, scores_cache)
    grads['W2'] += self.reg * W2
    dloss_h1[h1<=0] = 0
    _, grads['W1'], grads['b1'] = affine_relu_backward(dloss_h1, h1_cache)
    grads['W1'] += self.reg * W1
    pass


    # ============================================================ #
    # 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.
  # ============================================================ #
  self.h_dims = len(hidden_dims)
  for i in np.arange(len(hidden_dims) + 1):
      # eg: when len(hidden_dims) = 2, W1, W2, W3, b1, b2, b3 exist
      if i == 0:
          self.params['W'+str(i+1)] = weight_scale * np.random.randn(input_dim,
           hidden_dims[i])
```

```
            self.params['b'+str(i+1)] = np.zeros(hidden_dims[i])
        elif i == len(hidden_dims):
            self.params['W'+str(i+1)] = weight_scale *
             np.random.randn(hidden_dims[-1], num_classes)
            self.params['b'+str(i+1)] = np.zeros(num_classes)
        else:
            self.params['W'+str(i+1)] = weight_scale *
             np.random.randn(hidden_dims[i-1], hidden_dims[i])
            self.params['b'+str(i+1)] = np.zeros(hidden_dims[i])

    pass

    # ============================================================ #
    # 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".
    # ============================================================ #

    N = X.shape[0]
    D = np.prod(X.shape[1:])

    hidden = {}
    for i in np.arange(self.h_dims + 1):
        w = self.params['W'+str(i+1)]
        b = self.params['b'+str(i+1)]
        if i == 0:
            h, h_cache = affine_relu_forward(X, w, b)
            hidden['h'+str(i+1)] = h
            hidden['h_cache'+str(i+1)] = h_cache
        elif i == self.h_dims:
            scores, scores_cache = affine_forward(h, w, b)
#             hidden['h'+str(i+1)] = h
#             hidden['h_cache'+str(i+1)] = h_cache
        else:
            h, h_cache = affine_relu_forward(h, w, b)
            hidden['h'+str(i+1)] = h
            hidden['h_cache'+str(i+1)] = h_cache
    pass

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

    exp_scores = np.exp(scores) # shape = (N, C)
    sum_class_scores = np.sum(exp_scores, axis = 1, keepdims = 1) # shape = (N, 1)
    log_exp_scores = np.log(exp_scores[range(N), y]/sum_class_scores) # shape = (N,
     C)
    data_loss = -np.mean(log_exp_scores)

    reg_sum = 0
```

```
    for i in np.arange(self.h_dims+1):
        reg_sum += np.sum(self.params['W'+str(i+1)]*self.params['W'+str(i+1)])
    reg_loss = 0.5 * self.reg * reg_sum
    loss = data_loss + reg_loss

    dscores = exp_scores/sum_class_scores #(N, C)
    dscores[range(N), y] -= 1
    dscores /= N

    for i in np.arange(self.h_dims+1)[::-1]:
        if i == self.h_dims:
            dloss, w, b = affine_backward(dscores, scores_cache)
            grads['W'+str(i+1)] = w
            grads['W'+str(i+1)] += self.reg * self.params['W'+str(i+1)]
            grads['b'+str(i+1)] = b
            dloss[hidden['h'+str(i)]<=0] = 0
        elif i == 0:
            _, w, b = affine_relu_backward(dloss, hidden['h_cache'+str(i+1)])
            grads['W'+str(i+1)] = w
            grads['b'+str(i+1)] = b
            grads['W'+str(i+1)] += self.reg * self.params['W'+str(i+1)]

        else:
            dloss, w, b = affine_relu_backward(dloss, hidden['h_cache'+str(i+1)])
            grads['W'+str(i+1)] = w
            grads['b'+str(i+1)] = b
            grads['W'+str(i+1)] += self.reg * self.params['W'+str(i+1)]
            dloss[hidden['h'+str(i)]<=0] = 0

#
#         dloss_h2, grads['W3'], grads['b3'] = affine_backward(dscores, scores_cache)
#         grads['W3'] += self.reg * W3
#         dloss_h2[h2<=0] = 0
#
#         dloss_h1, grads['W2'], grads['b2'] = affine_backward(dloss_h2, h2_cache)
#         grads['W2'] += self.reg * W2
#         dloss_h1[h1<=0] = 0
#
#         _, grads['W1'], grads['b1'] = affine_relu_backward(dloss_h1, h1_cache)
#         grads['W1'] += self.reg * W1



    pass

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

```
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 = x.shape[0]
  D = np.prod(x.shape[1:])
  x2 = np.reshape(x, (N, D))
  out = np.dot(x2, w) + b
  pass

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

  pass
  N = x.shape[0]
  D = np.prod(x.shape[1:])
  dx = np.dot(dout, w.T)
  dx = np.reshape(dx, x.shape)

  x2 = np.reshape(x, (N, D))
  dw = np.dot(x2.T, dout)
  db = np.sum(dout, axis = 0)


  # ============================================================= #
  # 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.
  # ============================================================= #
```

```
  pass
  out = np.maximum(x, 0)

  # ============================================================ #
  # 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
  # ============================================================ #

  pass
  dx = np.array(dout)
  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
```