CS145 Howework 3, Part 1: kNN

Important Note: HW3 is due on **11:59 PM PT, Nov 9 (Monday, Week 6)**. Please submit through GradeScope.

Note that, Howework #3 has two jupyter notebooks to complete (Part 1: kNN and Part 2: Neural Network).

Print Out Your Name and UID

Name: Rui Deng, UID: 205123245

Before You Start

You need to first create HW2 conda environment by the given `cs145hw3.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw3.yml
conda activate hw3
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw3.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as hyperparameters) that you are allowed to edit (between `START/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

Download and prepare the dataset

Download the CIFAR-10 dataset (file size: ~163M). Run the following from the HW3 directory:

```
cd hw3/data/datasets
./get_datasets.sh
```

Make sure you put the dataset downloaded under hw3/data/datasets folder. After downloading the dataset, you can start your notebook from the HW3 directory. Note that the dataset is used in both jupyter notebooks (kNN and Neural Networks). You only need to download the dataset once for HW3.

Import the appropriate libraries

In [138]:

```python
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from data.data_utils import load_CIFAR10 # function to load the CIFAR-10 dataset.

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

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

Now, to verify that the dataset has been successfully set up, the following code will print out the shape of train/test data and labels. The output shapes for train/test data are (50000, 32, 32, 3) and (10000, 32, 32, 3), while the labels are (50000,) and (10000,) respectively.

In [139]:

```python
# Set the path to the CIFAR-10 data
cifar10_dir = './data/datasets/cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

Now we visualize some examples from the dataset by showing a few examples of training images from each class.

In [140]:

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 't
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [141]:

```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

Implement K-nearest neighbors algorithms

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

In [142]:

```
# Import the KNN class
from hw3code import KNN
```

In [143]:

```
# Declare an instance of the knn class.
knn = KNN()

# Train the classifier.
#   We have implemented the training of the KNN classifier.
#   Look at the train function in the KNN class to see what this does.
knn.train(X=X_train, y=y_train)
```

Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step of KNN?

Answers

(1) knn.train() simply stores all the training set so that later we can compare the testing data points to them.

(2) Pros: No pre-processing or evaluation required for the training set, so the training step is very fast. Cons: It is very memory consuming; moreover, we need more time for calculating the distance in the testing step.

KNN prediction

In the following sections, you will implement the functions to calculate the distances of test points to training points, and from this information, predict the class of the KNN.

In [8]:

```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of the no
#   in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2, 'fro')))
```

```
Time to run code: 42.58375787734985
Frobenius norm of L2 distances: 7906696.077040902
```

Really slow code?

Note: This probably took a while. This is because we use two for loops. We could increase the speed via vectorization, removing the for loops. Normally it may takes 20-40 seconds.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

KNN vectorization

The above code took far too long to run. If we wanted to optimize hyperparameters, it would be time-expensive. Thus, we will speed up the code by vectorizing it, removing the for loops.

In [22]:

```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any for l
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print(np.linalg.norm(dists_L2_vectorized, 'fro'))
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (should be 0): {
```

```
7906696.077040902
Time to run code: 1.8643591403961182
Difference in L2 distances between your KNN implementations (should be
0): 1.4651847440245846e-10
```

Speedup

Depending on your computer speed, you should see a 20-100x speed up from vectorization and no difference in L2 distances between two implementations.

On our computer, the vectorized form took 0.20 seconds while the naive implementation took 26.88 seconds.

Implementing the prediction

Now that we have functions to calculate the distances from a test point to given training points, we now implement the function that will predict the test point labels.

In [66]:

```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

error = 1

# ================================================================ #
# START YOUR CODE HERE
# ================================================================ #
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #

error = 1 - np.sum(knn.predict_labels(dists_L2, 1) == y_test) / len(y_test)

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

print(error)
```

```
0.726
```

If you implemented this correctly, the error should be: 0.726. This means that the k-nearest neighbors classifier is right 27.4% of the time, which is not great.

Questions:

What could you do to improve the accuracy of the k-nearest neighbor classifier you just implemented? Write down your answer in less than 30 words.

Answers:

We can improve the accuracy by using cross-validation to choose the best k.

Optimizing KNN hyperparameters k

In this section, we'll take the KNN classifier that you have constructed and perform cross validation to choose a best value of k.

If you are not familiar with cross validation, cross-validation is a technique for evaluating ML models by training several ML models on subsets of the available input data and evaluating them on the complementary subset of the data. Use cross-validation to detect overfitting, ie, failing to generalize a pattern. More specifically, in k-fold cross-validation, you evenly split the input data into k subsets of data (also known as folds). You train an ML model on all but one (k-1) of the subsets, and then evaluate the model on the subset that was not used for training. This process is repeated k times, with a different subset reserved for evaluation (and excluded from training) each time.

More details of cross validation can be found [here (https://scikit-learn.org/stable/modules/cross_validation.html)](https://scikit-learn.org/stable/modules/cross_validation.html). However, you are not allowed to use sklean in your implementation.

Create training and validation folds

First, we will create the training and validation folds for use in k-fold cross validation.

In [144]:

```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# START YOUR CODE HERE
# ================================================================ #
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #
knn.train(X_train, y_train)

for i in range(num_folds):
    X_train_folds.append(knn.X_train[1000 * i: 1000 * (i + 1)])
    y_train_folds.append(knn.y_train[1000 * i: 1000 * (i + 1)])
# ================================================================ #
# END YOUR CODE HERE
# =============================================================== #
```

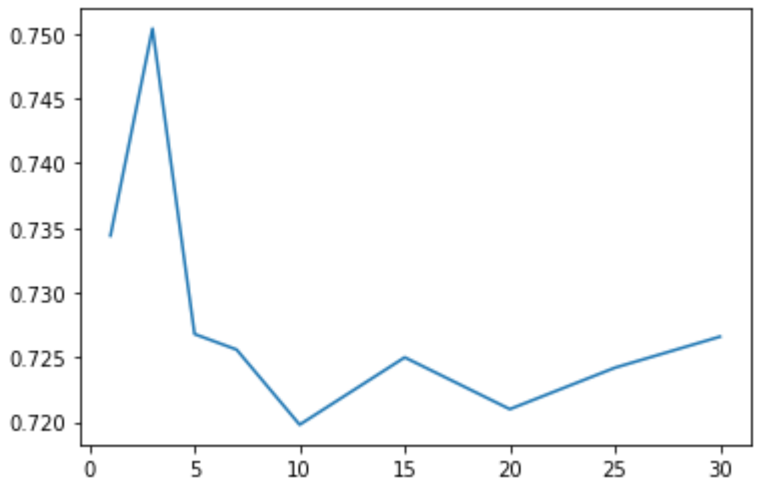
Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

In [150]:

```
time_start =time.time()

ks = [1, 3, 5, 7, 10, 15, 20, 25, 30]

# ================================================================ #
# START YOUR CODE HERE
# ================================================================ #
#   Calculate the cross-validation error for each k in ks, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. average cross-validation error.
#   Since we assume L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #
errors = np.zeros(len(ks))
for i in range(num_folds):
    first = True
    X_train_k, y_train_k = np.array([]), np.array([])
    for j in range(num_folds):
        if j != i:
            if first:
                X_train_k = X_train_folds[j]
                y_train_k = y_train_folds[j]
                first = False
            else:
                X_train_k = np.append(X_train_k, X_train_folds[j], axis=0)
                y_train_k = np.append(y_train_k, y_train_folds[j])
    knn = KNN()
    knn.train(X_train_k, y_train_k)
    X_test_k = np.array(X_train_folds[i])
    y_test_k = np.array(y_train_folds[i])
    dist = knn.compute_L2_distances_vectorized(X=X_test_k)
    for a,k in enumerate(ks):
        errors[a] += 1 - np.sum(knn.predict_labels(dist, k) == y_test_k) / len(y_tes
errors /= num_folds
plt.plot(ks, errors)

best_k = ks[np.argmin(errors)]
print(errors)
print(f'best k is: {best_k}')
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Computation time: %.2f'%(time.time()-time_start))
```

```
[0.7344 0.7504 0.7268 0.7256 0.7198 0.725  0.721  0.7242 0.7266]
best k is: 10
Computation time: 20.58
```



Questions:

(1) Why do we typically choose k as an odd number (for exmple in `ks`)

(2) What value of k is best amongst the tested k's? What is the cross-validation error for this value of k?

Answers

(1) We want k to be odd so that there will not be ties during classification. If a tie exists, then we have to choose randomly between these two classes, which leads to uncertainty.

(2) The best k: 10. The cross-validation error for this value: 0.7198.

Evaluating the model on the testing dataset.

Now, given the optimal k which you have learned, evaluate the testing error of the k-nearest neighbors model.

In [148]:

```
error = 1

# ================================================================ #
# START YOUR CODE HERE
# ================================================================ #
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #

knn.train(X_train, y_train)
error = 1 - np.sum(knn.predict_labels(dists_L2, 10) == y_test) / len(y_test)

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

print('Error rate achieved: {}'.format(error))
```

```
Error rate achieved: 0.718
```

Question:

How much did your error change by cross-validation over naively choosing $k = 1$ and using the L2-norm?

Answers

Please write down your answer here!

Error change = 0.726 - 0.718 = 0.008

End of Homework 3, Part 1 :)

After you've finished both parts the homework, please print out the both of the entire `ipynb` notebooks and `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Do not include any dataset in your submission.

CS145 Howework 3, Part 2: Neural Networks

Important Note: HW3 is due on **11:59 PM PT, Nov 9 (Monday, Week 6)**. Please submit through GradeScope.

Note that, Howework #3 has two jupyter notebooks to complete (Part 1: kNN and Part 2: Neural Network).

Print Out Your Name and UID

Name: Rui Deng, UID: 205123245

Before You Start

You need to first create HW3 conda environment by the given `cs145hw3.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw3.yml
conda activate hw3
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw3.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

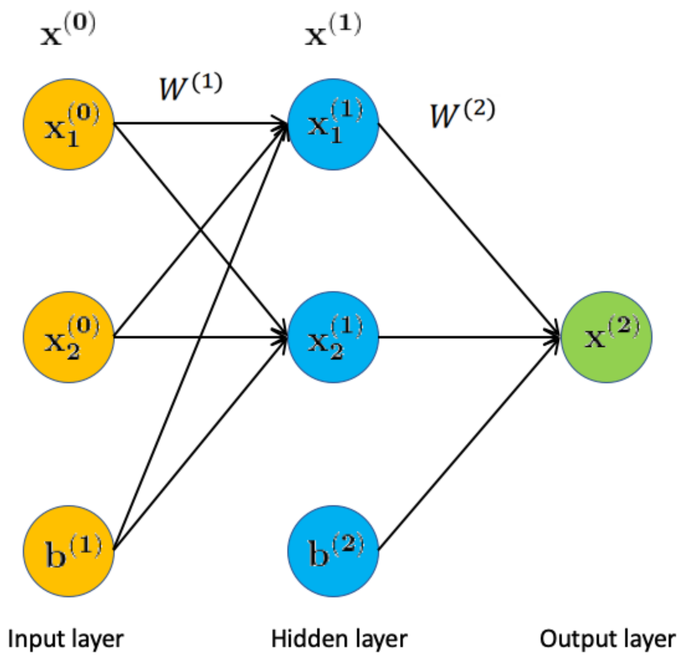
You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as hyperparameters) that you are allowed to edit (between `START/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

Section 1: Backprop in a neural network

Note: Section 1 is "question-answer" style problem. You do not need to code anything and you are required to calculate by hand (with a scientific calculator), which helps you understand the back propagation in neural networks.

In this question, let's consider a simple two-layer neural network and manually do the forward and backward pass. For simplicity, we assume our input data is two dimension. Then the model architecture looks like the following. Notice that in the example we saw in class, the bias term `b` was not explicit listed in the architecture diagram. Here we include the term `b` explicitly for each layer in the diagram. Recall the formula for computing $\mathbf{x^{(l)}}$ in the l-th layer from $\mathbf{x^{(l-1)}}$ in the $(l-1)$-th layer is $\mathbf{x^{(l)}} = \mathbf{f^{(l)}(W^{(l)}x^{(l-1)} + b^{(l)})}$. The activation function $\mathbf{f^{(l)}}$ we choose is the `sigmoid` function for all layers, i.e. $\mathbf{f^{(l)}}(z) = \frac{1}{1+\exp(-z)}$. The final loss function is $\frac{1}{2}$ of the `mean squared error` loss, i.e. $l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||^2$.



We initialize our weights as

$$\mathbf{W^{(1)}} = \begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix}, \quad \mathbf{W^{(2)}} = [0.4, 0.45], \quad \mathbf{b^{(1)}} = [0.35, 0.35], \quad \mathbf{b^{(2)}} = 0.6$$

Forward pass

Questions

1. When the input $\mathbf{x^{(0)}} = [0.05, 0.1]$, what will be the value of $\mathbf{x^{(1)}}$ in the hidden layer? (Show your work).
2. Based on the value $\mathbf{x^{(1)}}$ you computed, what will be the value of $\mathbf{x^{(2)}}$ in the output layer? (Show your work).
3. When the target value of this input is $y = 0.01$, based on the value $\mathbf{x^{(2)}}$ you computed, what will be the loss? (Show your work).

Answers:

1.

$$x^{(1)} = \mathbf{f^{(1)}(W^{(1)}x^{(0)} + b^{(1)})} = f^{(1)}(\begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix} * [0.05, 0.1]^T + [0.35, 0.35])$$

$$= f^{(1)}([0.3775, 0.3925]) = [0.593, 0.5969]$$

2.

$$x^{(2)} = \mathbf{f^{(2)}(W^{(2)}x^{(1)} + b^{(2)})}$$

$$= f^{(2)}([0.59326, 0.59688][0.4, 0.45]^T + 0.6) = 0.751$$

3.

$$loss = l(\mathbf{y}, \mathbf{x^{(2)}}) = \frac{1}{2}||\mathbf{y} - \mathbf{x^{(2)}}||^2 = \frac{1}{2} * ||0.01 - 0.7514||^2 = 0.2748$$

Backward pass

With the loss computed below, we are ready for a backward pass to update the weights in the neural network. Kindly remind that the gradients of a variable should have the same shape with the variable.

Questions

1. Consider the loss l of the same input $\mathbf{x^{(0)}} = [0.05, 0.1]$, what will be the update of $\mathbf{W^{(2)}}$ and $\mathbf{b^{(2)}}$ when we backprop, i.e. $\frac{\partial l}{\partial \mathbf{W^{(2)}}}$, $\frac{\partial l}{\partial \mathbf{b^{(2)}}}$ (Show your work in detailed calculation steps. Answers without justification will not be credited.).
2. Based on the result you computed in part 1, when we keep backproping, what will be the update of $\mathbf{W^{(1)}}$ and $\mathbf{b^{(1)}}$, i.e. $\frac{\partial l}{\partial \mathbf{W^{(1)}}}$, $\frac{\partial l}{\partial \mathbf{b^{(1)}}}$ (Show your work in details calculation steps. Answers without justification will not be credited.).

Answers:

1. $\frac{\partial l}{\partial \mathbf{W^{(2)}}} = \frac{\partial l}{\partial \mathbf{x^{(2)}}}\frac{\partial \mathbf{x^{(2)}}}{\partial z}\frac{\partial z}{\partial W^{(2)}}$

$$= -(y - x^{(2)}) * f^{(2)'}(z) * x^{(1)} = -(y - x^{(2)}) * (f(z) * (1 - f(z)) * (x^{(1)})$$

$$= -(0.01 - 0.7514) * (f(1.1059) * (1 - f(1.1059)) * ([0.59326, 0.59688]) = [0.0822, 0.0827]$$

$$\frac{\partial l}{\partial b^{(2)}} = \frac{\partial l}{\partial \mathbf{x^{(2)}}}\frac{\partial \mathbf{x^{(2)}}}{\partial z}\frac{\partial z}{\partial b^{(2)}} = -(y - x^{(2)}) * f^{(2)'}(z) * 1$$

$$= -(y - x^{(2)}) * (f(z) * (1 - f(z)) * 1 = -(0.01 - 0.7514) * (f(1.1059) * (1 - f(1.1059)) = 0.1385$$

2. $\frac{\partial l}{\partial \mathbf{W^{(1)}}} = \frac{\partial l}{\partial \mathbf{x^{(2)}}}\frac{\partial \mathbf{x^{(2)}}}{\partial z}\frac{\partial z^{(2)}}{\partial x^{(1)}}\frac{\partial \mathbf{x^{(1)}}}{\partial z^{(1)}}\frac{\partial \mathbf{z^{(1)}}}{\partial W^{(1)}}$

$$= -(y - x^{(2)}) * f^{(2)'}(z^{(2)}) * W^{(2)} * f^{(1)'}(z^{(1)}) * (x^{(0)})$$

$$= -(0.01 - 0.7514) * (f(1.1059) * (1 - f(1.1059)) * [0.4, 0.45] * (f([0.3775, 0.3925]) * (1 - f([0.377$$

$$= \begin{bmatrix} 0.0006683 & 0.0013367 \\ 0.0007498. & 0.0014996 \end{bmatrix}$$

$$\frac{\partial l}{\partial \mathbf{b^{(1)}}} = \frac{\partial l}{\partial \mathbf{x^{(2)}}}\frac{\partial \mathbf{x^{(2)}}}{\partial z}\frac{\partial z^{(2)}}{\partial x^{(1)}}\frac{\partial \mathbf{x^{(1)}}}{\partial z^{(1)}}\frac{\partial \mathbf{z^{(1)}}}{\partial b^{(1)}}$$

$$= -(y - x^{(2)}) * f^{(2)'}(z^{(2)}) * W^{(2)} * f^{(1)'}(z^{(1)}) * 1$$

$$= -(0.01 - 0.7514) * (f(1.1059) * (1 - f(1.1059)) * [0.4, 0.45] * (f([0.3775, 0.3925]) * (1 - f([0.377$$

$$= [0.01337, 0.0150]$$

Section 2: Coding a two-layer neural network

Import libraries and define relative error function, which is used to check results later.

In [1]:

```
import random
import numpy as np
from data.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0)
%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 hw3code.neural_net import TwoLayerNet
```

In [63]:

```
# 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 [22]:

```
## 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)))
```

```
(3,)
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:
3.381231204052648e-08
```

Forward pass loss

The total loss includes data loss (MSE) and regularization loss, which is,

$$L = L_{data} + L_{reg} = \frac{1}{2N}\sum_{i=1}^{N}\left(y_{\text{pred}} - y_{\text{target}}\right)^2 + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

More specifically in multi-class situation, if the output of neural nets from one sample is $y_{\text{pred}} = (0.1, 0.1, 0.8)$ and $y_{\text{target}} = (0, 0, 1)$ from the given label, then the MSE error will be
$Error = (0.1 - 0)^2 + (0.1 - 0)^2 + (0.8 - 1)^2 = 0.06$

Implement data loss and regularization loss. In the MSE function, you also need to return the gradients which need to be passed backward. This is similar to batch gradient in linear regression. Test your implementation of loss functions. The Difference should be less than 1e-12.

```
In [32]:
```

```python
loss, _ = net.loss(X, y, reg=0.05)
correct_loss_MSE = 1.8973332763705641

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

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

Backward pass (You do not need to implemented this part)

We have already implemented the backwards pass of the neural network for you. Run the block of code to check your gradients with the gradient check utilities provided. The results should be automatically correct (tiny relative error).

If there is a gradient error larger than 1e-8, the training for neural networks later will be negatively affected.

```
In [33]:
```

```python
from data.gradient_check import eval_numerical_gradient

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

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

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=Fals
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, g
```

```
W2 max relative error: 6.774278173332322e-11
b2 max relative error: 1.887502392114964e-11
W1 max relative error: 1.7476665046687833e-09
b1 max relative error: 7.382451041178829e-10
```

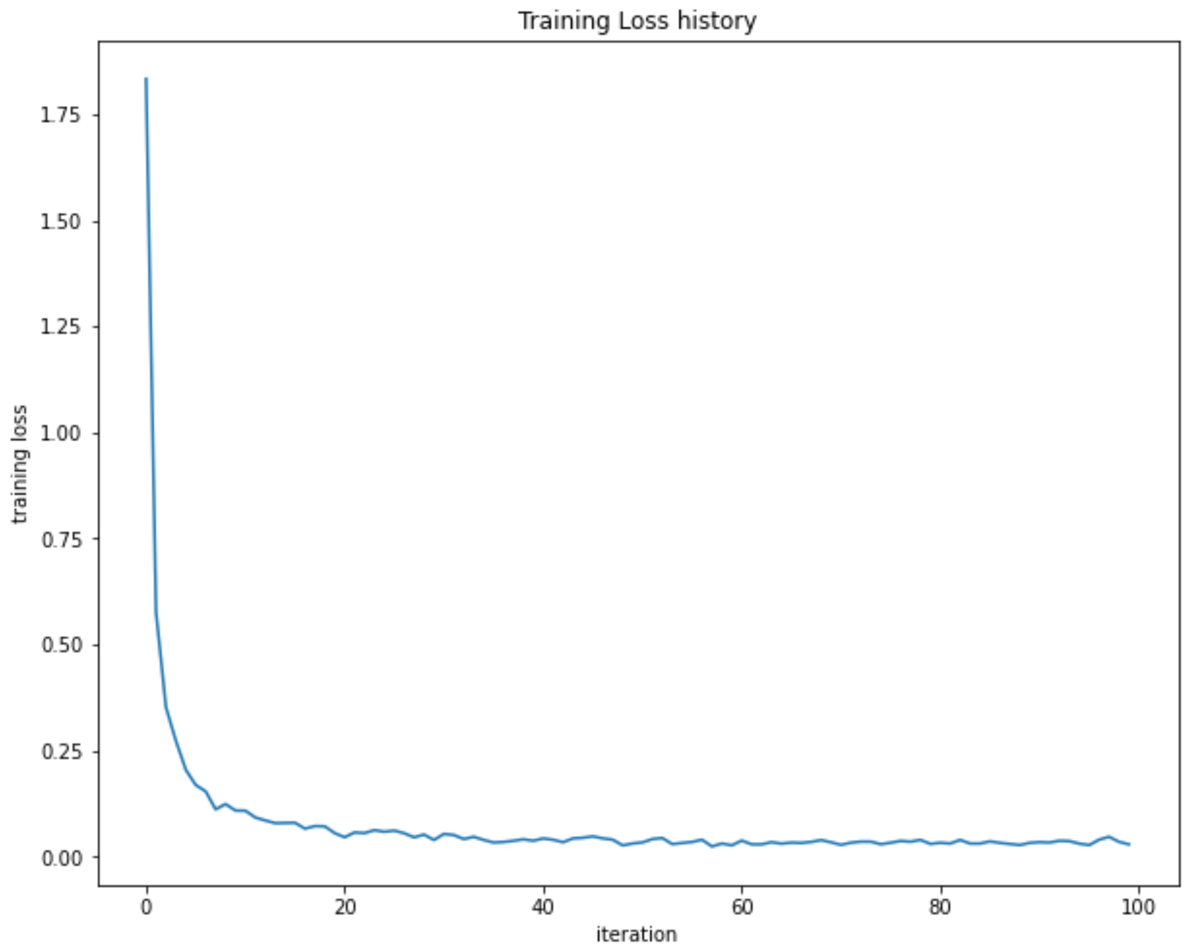
Training the network

Implement neural_net.train() to train the network via stochastic gradient descent, much like the linear regression.

In [64]:

```
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:  0.02950555626206818
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [67]:

```
from data.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 = './data/datasets/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 15-18%.

In [61]:

```
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-5, learning_rate_decay=0.95,
            reg=0.1, 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
test_acc = (subopt_net.predict(X_test) == y_test).mean()
print('Test accuracy (subopt_net): ', test_acc)
```

```
iteration 0 / 1000: loss 0.5000859824290886
iteration 100 / 1000: loss 0.4998653761648168
iteration 200 / 1000: loss 0.499701368056399
iteration 300 / 1000: loss 0.499470925649445
iteration 400 / 1000: loss 0.49918854812872415
iteration 500 / 1000: loss 0.49889840589905077
iteration 600 / 1000: loss 0.49844936986971783
iteration 700 / 1000: loss 0.4978859032020733
iteration 800 / 1000: loss 0.49683796709957606
iteration 900 / 1000: loss 0.49548027339542006
Validation accuracy:  0.168
Test accuracy (subopt_net):  0.193
```

In [39]:

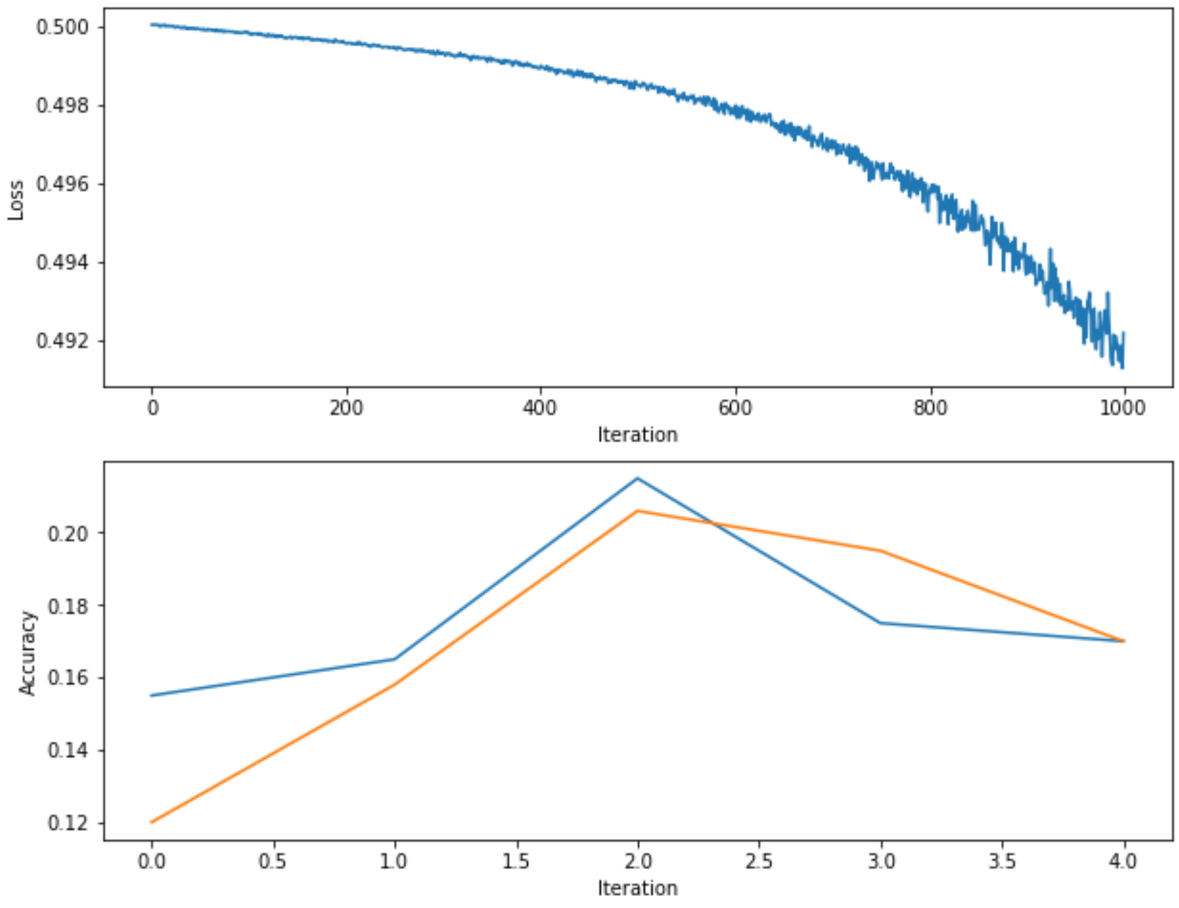
```
stats['train_acc_history']
```

Out[39]:

```
[0.155, 0.165, 0.215, 0.175, 0.17]
```

In [40]:

```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')

plt.show()
```



Questions:

The training accuracy isn't great. It seems even worse than simple KNN model, which is not as good as expected.

(1) What are some of the reasons why this is the case? Based on previous observations, please provide at least two possible reasons with justification.

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

Answers:

(1)

The first reason is that the learning rate is too small for the parameters to converge to the optimized configure. We can see that the loss only decreases by 0.1 from the frist iteration to the 1000 iteration. That is to say, we do choose the right direction to perform gradient descent, but with very little step sizes, we are still far from the optimal solution even after 1000 iterations.

The second reason is that we run the algorithm for too few iterations. We can see a clearly decreasing trend of the loss function in the last few iterations, which means that our algorithm does not converge to an optimal configure yet. That is to say, we need to train it for more iterations so that the loss stops decreasing, i.e. we reach the optimal solution for this problem.

(2)

To fix the first problem, we can use learning rate = 1e-3 or 1e-4.

To fix the second problem, we can increase the number of iterations to 3000 or 5000.

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. To get the full credit of the neural nets, you should get at least **45%** accuracy on validation set.

**Reminder: Think about whether you should retrain a new model from scratch every time your try a new set of hyperparameters.* *

In [51]:

```
best_net = None # store the best model into this

# ================================================================ #
# START YOUR CODE HERE:
# ================================================================ #
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 45% 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 - 23%)) / %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)!
# ================================================================ #

# todo: optimal parameter search (you may use grid search by for-loops )
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net_opt = TwoLayerNet(input_size, hidden_size, num_classes)

learning_rate = [1e-5, 1e-4, 1e-3]
num_iters = [1000, 3000, 5000]
best_lr = 0.0
best_iters = 0
best_valacc = 0
best_net = net

for l in learning_rate:
    for num_iter in num_iters:
        # Train the network
        stats = net_opt.train(X_train, y_train, X_val, y_val,
                    num_iters=num_iter, batch_size=200,
                    learning_rate=l, learning_rate_decay=0.95,
                    reg=0.1, verbose=False)

        # Predict on the validation set
        val_acc = (net_opt.predict(X_val) == y_val).mean()
        if val_acc > best_valacc:
            best_valacc = val_acc
            best_lr = l
            best_iters = num_iter
            best_net = net_opt


# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
# Output your results
print("== Best parameter settings ==")
print(f'learning rate: {best_lr}, iterations: {best_iters}')
print("Best accuracy on validation set: {}".format(best_valacc))
```

```
== Best parameter settings ==
learning rate: 0.001, iterations: 5000
Best accuracy on validation set: 0.505
```

Quesions

(1) What is your best parameter settings? (Output from the previous cell)

(2) What parameters did you tune? How are they changing the performance of nerural network? You can discuss any observations from the optimization.

Answers

(1)

My best parameter settings are {num_iters=5000, batch_size=200, learning_rate=0.001, learning_rate_decay=0.95, reg=0.1}.

(2)

I tuned the learning rate and the iterations (corresponding to my answers for the last question set).

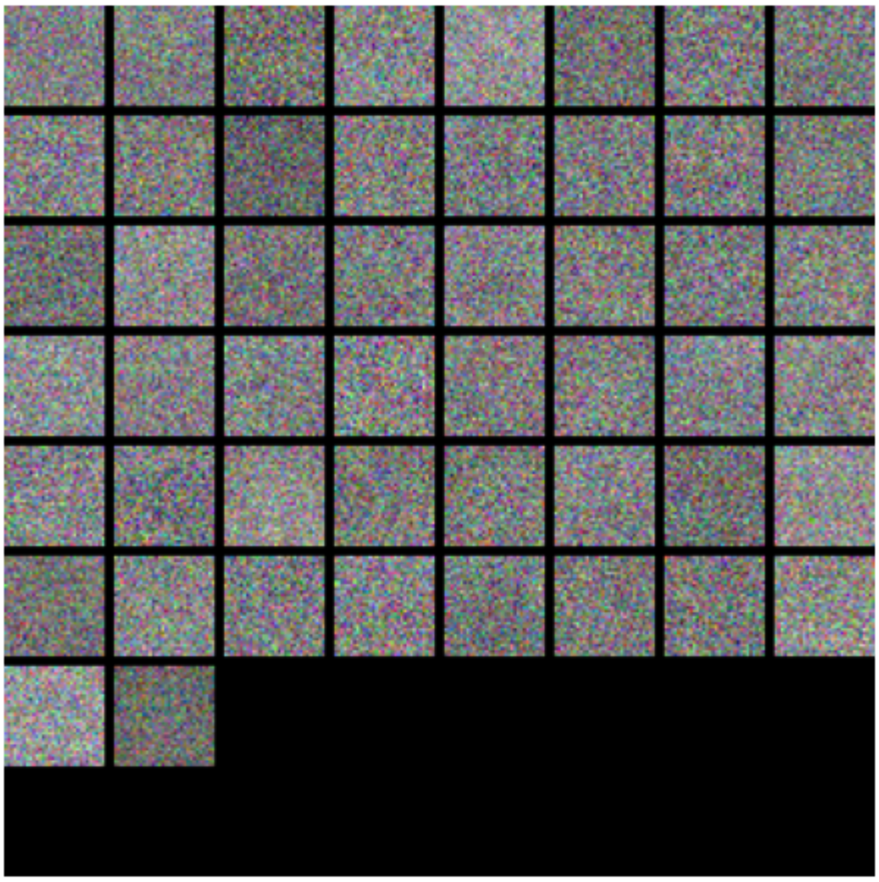
Here, increasing the learning rate increases the speed and performance of convergence. However, I also tried using learning_rate = 0.01. In this case, I got an error of overflow when computing the loss. This is possibly caused by overshooting due to the large learning rate (we jump over the optimal solution and thus diverge to larger and larger loss). That is to say, we have to choose the appropriate learning rate: not too small for algorithm to converge, not too big for algorithm to overshoot.

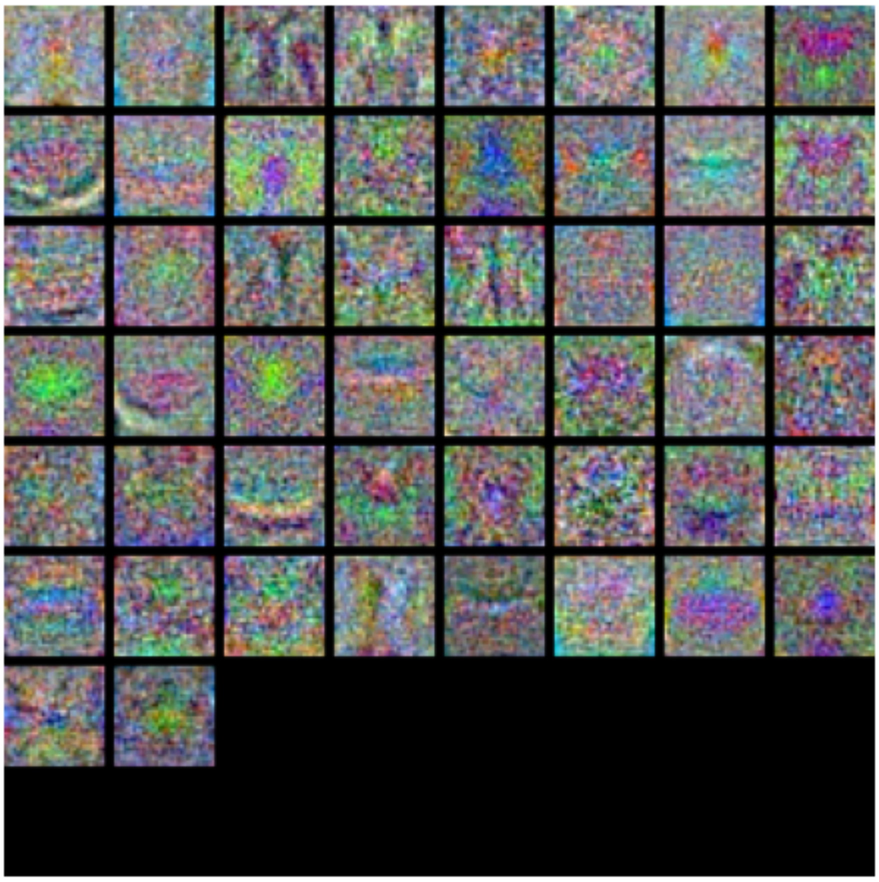
Also, increasing the number of iterations allows the algorithm to take more steps toward the optimal solution, which can therefore give us a better accuracy.

Visualize the weights of your neural networks

In [53]:

```
from data.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)
```





Questions:

What differences do you see in the weights between the suboptimal net and the best net you arrived at? What do the weights in neural networks probably learn after training?

Answer:

The weights in the suboptimal seem randomly disparsed, whereas the weights in the best net try to capture and represent certain pattern existing in the image. So it is likely that neural networks assign weights to different neurons so that each of them captures a specific feature/location in the image.

Evaluate on test set

In [55]:

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

```
Test accuracy (best_net):  0.477
```

Questions:

(1) What is your test accuracy by using the best NN you have got? How much does the performance increase compared with kNN? Why can neural networks perform better than kNN?

(2) Do you have any other ideas or suggestions to further improve the performance of neural networks other than the parameters you have tried in the homework?

Answers:

(1)

My test accuracy for the best NN is 0.477, whereas the test accuracy for kNN is 1-0.718=0.282. We can see that the best NN doubles the classification accuracy of kNN.

Firstly, when we running the kNN classifier, we did subsample so that we could run it sufficiently in the homeowork; however, we trained the neural network with the entire dataset. So with more training data, the neural network can learn more features for classifying the images.

Secondly, the neural network is a more complex, non-linear classifier that extracts the feature from images instead of simply measuring the distances between image matrices. Therefore, with stronger learning ability empowered to the NN by its descenting gradients and non-linear activation functions, it should perform better than the kNN.

(2)

We can add batch normalization layer to the neural network so that inputs to the layers are normalized and aligned with each other, which would therefore speed up the converging process. Moreover, we could change the simple gradient descent schemes by introducing the momentum to SGD optimizer so that we will not be stucked at the local minimum. Finally, we can use more complex neural network, such as convolutional neural network, to do image classification.

Bonus Question: Change MSE Loss to Cross Entropy Loss

This is a bonus question. If you finish this (cross entropy loss) correctly, you will get **up to 10 points** (add up to your HW3 score).

Note: From grading policy of this course, your maximum points from homework are still 25 out of 100, but you can use the bonus question to make up other deduction of other assignments.

Pass output scores in networks from forward pass into softmax function. The softmax function is defined as,

$$p_j = \sigma(z_j) = \frac{e^{z_j}}{\sum_{c=1}^{C} e^{z_c}}$$

After softmax, the scores can be considered as probability of j-th class.

The cross entropy loss is defined as,

$$L = L_{\text{CE}} + L_{reg} = \frac{1}{N}\sum_{i=1}^{N} \log\left(p_{i,j}\right) + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

To take derivative of this loss, you will get the gradient as,

$$\frac{\partial L_{\text{CE}}}{\partial o_i} = p_i - y_i$$

More details about multi-class cross entropy loss, please check http://cs231n.github.io/linear-classify/ (http://cs231n.github.io/linear-classify/) and more explanation (https://deepnotes.io/softmax-crossentropy) about the derivative of cross entropy.

Change the loss from MSE to cross entropy, you only need to change you `MSE_loss(x,y)` in `TwoLayerNet.loss()` function to `softmax_loss(x,y)`.

Now you are free to use any code to show your results of the two-layer networks with newly-implemented cross entropy loss. You can use code from previous cells.

In [69]:

```
# Start training your networks and show your results
# ================================================================ #
# START YOUR CODE HERE:
# ================================================================ #
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net_softmax = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net_softmax.train(X_train, y_train, X_val, y_val,
            num_iters=3000, batch_size=200,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=0.1, verbose=True)

# Predict on the validation set
val_acc = (net_softmax.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
iteration 0 / 3000: loss 2.3026393267650147
iteration 100 / 3000: loss 2.0848698180201826
iteration 200 / 3000: loss 1.7003457338774617
iteration 300 / 3000: loss 1.748928135152143
iteration 400 / 3000: loss 1.728284975788102
iteration 500 / 3000: loss 1.6788535227446493
iteration 600 / 3000: loss 1.5643908219212428
iteration 700 / 3000: loss 1.4268699579181514
iteration 800 / 3000: loss 1.4306304232530025
iteration 900 / 3000: loss 1.6354606920022852
iteration 1000 / 3000: loss 1.4534764149906336
iteration 1100 / 3000: loss 1.3344675914065307
iteration 1200 / 3000: loss 1.3172689984614083
iteration 1300 / 3000: loss 1.422096505387887
iteration 1400 / 3000: loss 1.4202450777712905
iteration 1500 / 3000: loss 1.4673017137202653
iteration 1600 / 3000: loss 1.4223901494589695
iteration 1700 / 3000: loss 1.4479856331352314
iteration 1800 / 3000: loss 1.390127308797559
iteration 1900 / 3000: loss 1.2866353970841153
iteration 2000 / 3000: loss 1.314509731757051
iteration 2100 / 3000: loss 1.3608446487870944
iteration 2200 / 3000: loss 1.287792042591072
iteration 2300 / 3000: loss 1.3136696600177304
iteration 2400 / 3000: loss 1.410278665894837
iteration 2500 / 3000: loss 1.5542962066651342
iteration 2600 / 3000: loss 1.36638870330148
iteration 2700 / 3000: loss 1.3340558341257496
iteration 2800 / 3000: loss 1.3334039918134322
iteration 2900 / 3000: loss 1.254886711957481
Validation accuracy:  0.503
```

End of Homework 3, Part 2 :)

After you've finished both parts the homework, please print out the both of the entire `ipynb` notebooks and `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Do not include any dataset in your submission.

```
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and
 modified for CS145 at UCLA.
"""

class KNN(object):

    def __init__(self):
        pass

    def train(self, X, y):
        """
        Inputs:
        - X is a numpy array of size (num_examples, D)
        - y is a numpy array of size (num_examples, )
        """
        # ================================================================ #
        # START YOUR CODE HERE
        # ================================================================ #
        #   Hint: KNN does not do any further processsing, just store the
         training
        #   samples with labels into as self.X_train and self.y_train
        # ================================================================ #
        self.X_train = X
        self.y_train = y
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

    def compute_distances(self, X, norm=None):
        """
        Compute the distance between each test point in X and each training
         point
        in self.X_train.

        Inputs:
        - X: A numpy array of shape (num_test, D) containing test data.
        - norm: the function with which the norm is taken.

        Returns:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          is the Euclidean distance between the ith test point and the jth
           training
          point.
        """
        if norm is None:
            norm = lambda x: np.sqrt(np.sum(x**2)) #norm = 2
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in np.arange(num_test):

            for j in np.arange(num_train):
            # ==================================================================
             #
            # START YOUR CODE HERE
            # ==================================================================
             #
            #   Compute the distance between the ith test point and the jth
            #   training point using norm(), and store the result in dists[i,
             j].
            # ==================================================================
             #

                dists[i][j] = norm(X[i] - self.X_train[j])

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

        return dists

    def compute_L2_distances_vectorized(self, X):
        """
        Compute the distance between each test point in X and each training
         point
        in self.X_train WITHOUT using any for loops.

        Inputs:
        - X: A numpy array of shape (num_test, D) containing test data.

        Returns:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          is the Euclidean distance between the ith test point and the jth
           training
          point.
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))

        # ================================================================ #
        # START YOUR CODE HERE
        # ================================================================ #
        #   Compute the L2 distance between the ith test point and the jth
        #   training point and store the result in dists[i, j].  You may
        #    NOT use a for loop (or list comprehension).  You may only use
        #     numpy operations.
        #
        #     HINT: use broadcasting.  If you have a shape (N,1) array and
        #   a shape (M,) array, adding them together produces a shape (N, M)
        #   array.
        # ================================================================ #

        x_norm = np.expand_dims(X.dot(X.T).diagonal(), axis=-1)
        y_norm = self.X_train.dot(self.X_train.T).diagonal()
        dists = np.sqrt(x_norm + y_norm - 2 * X.dot(self.X_train.T))

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

        return dists


    def predict_labels(self, dists, k=1):
        """
        Given a matrix of distances between test points and training points,
        predict a label for each test point.

        Inputs:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          gives the distance betwen the ith test point and the jth training
           point.

        Returns:
        - y: A numpy array of shape (num_test,) containing predicted labels for
         the
          test data, where y[i] is the predicted label for the test point X[i].
        """
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in range(num_test):
            # A list of length k storing the labels of the k nearest neighbors
             to
            # the ith test point.

            closest_y = []

            # ================================================================
             #
            # START YOUR CODE HERE
            # ================================================================
             #
            #   Use the distances to calculate and then store the labels of
            #   the k-nearest neighbors to the ith test point.  The function
            #   numpy.argsort may be useful.
            #
            #   After doing this, find the most common label of the k-nearest
            #   neighbors.  Store the predicted label of the ith training
             example
            #   as y_pred[i].  Break ties by choosing the smaller label.
            # ================================================================
             #

            closest_y = np.argsort(dists[i])
            closest_y = [self.y_train[y] for y in closest_y[:k]]
            (vals, counts) = np.unique(closest_y, return_counts=True)
            ind = np.argmax(counts)
            y_pred[i] = vals[ind]

            # ================================================================
             #
            # END YOUR CODE HERE
            # ================================================================
             #
        return y_pred
```

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

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

    ReLU function:
    (i) x = x if x >= 0  (ii) x = 0 if x < 0

    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

# ================================================================ #
# START 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 fully-connected
#   layer.
#   The code is partially given
#   The output of the second fully connected layer is the output
 scores.
#   Do not use a for loop in your implementation.
#   Please use 'h1' as input of hidden layers, and 'a2' as output of
#   hidden layers after ReLU activation function.
#   [Input X] --W1,b1--> [h1] -ReLU-> [a2] --W2,b2--> [scores]
#   You may simply use np.maximun for implementing ReLU.
#   Note that there is only one ReLU layer.
#   Note that plase do not change the variable names (h1, h2, a2)
# ================================================================ #
```

```python
h1 = X.dot(W1.T) + b1
a2 = np.maximum(h1, 0)
h2 = a2.dot(W2.T) + b2
scores = h2

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

# scores is num_examples by num_classes (N, C)
def softmax_loss(x, y):
    loss, dx = 0,0
    # ================================================================
     #
    # START YOUR CODE HERE (BONUS QUESTION)
    # ================================================================
     #
    #   Calculate the cross entropy loss after softmax output layer.
    #   The format are provided in the notebook.
    #   This function should return loss and dx, same as MSE loss
     function.
    # ================================================================
     #

    n = x.shape[0]
    stable_e = np.exp(x - np.max(x, axis=1, keepdims=True))
    p = stable_e / np.sum(stable_e, axis=1, keepdims=True)
    loss = -np.sum(np.log(p[np.arange(n), y])) / n
    p[np.arange(n), y] -= 1
    dx = p / n

    # ================================================================
     #
    # END YOUR CODE HERE
    # ================================================================
     #
    return loss, dx


def MSE_loss(x, y):
    loss, dx = 0,0
    # ==================================================================
     #
    # START YOUR CODE HERE
    # ==================================================================
     #
    #   This function should return loss and dx (gradients ready for
     back prop).
    #   The loss is MSE loss between network ouput and one hot vector
     of class
    #   labels is required for backpropogation.
    # ==================================================================
     #
    # Hint: Check the type and shape of x and y.
    #       e.g. print('DEBUG:x.shape, y.shape', x.shape, y.shape)

    N = y.shape[0]
    y_cmp = np.zeros((N, x.shape[1]))
    for i in range(N):
        y_cmp[i][y[i]] = 1
    loss = ((x - y_cmp) ** 2).sum(axis=1).sum() / (2 * N)

    dx = (x - y_cmp) / N

    # ==================================================================
     #
    # END YOUR CODE HERE
    # ==================================================================
     #
    return loss, dx

# data_loss, dscore = softmax_loss(scores, y)
# The above line is for bonus question. If you have implemented
 softmax_loss, de-comment this line instead of MSE error.

data_loss, dscore = softmax_loss(scores, y) # "comment" this line if
 you use softmax_loss
# ================================================================ #
# START YOUR CODE HERE
# ================================================================ #
#   Calculate the regularization loss. Multiply the regularization
#   loss by 0.5 (in addition to the factor reg).
# ================================================================ #
reg_loss = 0.5 * reg * (np.sum(W1*W1) + np.sum(W2*W2))

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
loss = data_loss + reg_loss

grads = {}
```

```
        # ================================================================ #
        # START YOUR CODE HERE
        # ================================================================ #
        # Backpropogation: (You do not need to change this!)
        #   Backward pass is implemented. From the dscore error, we calculate
        #   the gradient and store as grads['W1'], etc.
        # ================================================================ #
        grads['W2'] = a2.T.dot(dscore).T + reg * W2
        grads['b2'] = np.ones(N).dot(dscore)

        da_h = np.zeros(h1.shape)
        da_h[h1>0] = 1
        dh = (dscore.dot(W2) * da_h)

        grads['W1'] = np.dot(dh.T,X) + reg * W1
        grads['b1'] = np.ones(N).dot(dh)
        # ================================================================ #
        # 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

    #   Create a minibatch (X_batch, y_batch) by sampling batch_size
    #   samples randomly.

    b_index = np.random.choice(num_train, batch_size)
    X_batch = X[b_index]
    y_batch = y[b_index]

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

    # ================================================================
     #
    # START YOUR CODE HERE
    # ================================================================
     #
    #   Perform a gradient descent step using the minibatch to update
    #   all parameters (i.e., W1, W2, b1, and b2).
    #   The gradient has been calculated as grads['W1'], grads['W2'],
    #   grads['b1'], grads['b2']
    #   For example,
    #   W1(new) = W1(old) - learning_rate * grads['W1']
    #   (this is not the exact code you use!)
    # ================================================================
     #

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

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

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

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

```
            val_acc = (self.predict(X_val) == y_val).mean()
            train_acc_history.append(train_acc)
            val_acc_history.append(val_acc)

            # Decay learning rate
            learning_rate *= learning_rate_decay

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

def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest
     score.

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

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

    # ================================================================ #
    # START YOUR CODE HERE
    # ================================================================ #
    #   Predict the class given the input data.
    # ================================================================ #

    scores = self.loss(X)
    y_pred = np.argmax(scores, axis=1)

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

    return y_pred
```