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: Danning Yu, UID: 305087992

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 `STRART/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 [1]:
```
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 .p
y files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-in-ipython
%load_ext autoreload
%autoreload 2
```

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 [86]:
```
# 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 d
ata.
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 [87]:
```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'hor
se', 'ship', 'truck']
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 [88]:
```
# Subsample the data for more efficient code execution in this exerci
se
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 [89]:
```
# Import the KNN class
from hw3code import KNN
```

In [90]:
```
# 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 doe
s.
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

Question 1
`knn.train()` simply saves the X and y training data into the class. It does not do any sort of calculations or processing. This is because KNN is an online algorithm.

Question 2
The advantage of this is that training basically costs nothing - precisely because we are basically doing nothing. This makes the training (or lack thereof, more precisely) very fast. However, this lack of training is also a con - we currently know nothing about the data, and when we predict something, we'll need to do calculations on the entire dataset to make a prediction for a single point. This will make predictions slower. Another advantage is that doing the prediction as data points come in means that we don't immediately commit to parameters for making predictions, which means we can be more flexible if needed for predictions. If for some reason we get a second batch of labeled data, we can easily incorporate it into our model and use it to make better predictions going forward, whereas for a normal model, you'd have to retrain it from scratch.

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 defini
tion of the norm
#   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(dist
s_L2, 'fro')))
```

```
Time to run code: 26.885533809661865
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 [91]:
```
# 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 loops.
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (should be 0): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 0.7353084087371826
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

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 [92]:
```
# 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.
# ================================================================ #
dist = knn.compute_L2_distances_vectorized(X=X_test)
y_pred = knn.predict_labels(dist)
N = len(y_pred)
num_incorrect = len([y_pred[i] for i in range(N) if y_pred[i] != y_te
st[i]])
error = num_incorrect / N
# ================================================================ #
# 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 could optimize the value of k, which is a hyperparameter that's very important in the kNN model. Doing so would probably lead to a higher accuracy.

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). However, you are not allowed to use sklean in your implementation.

Create training and validation folds

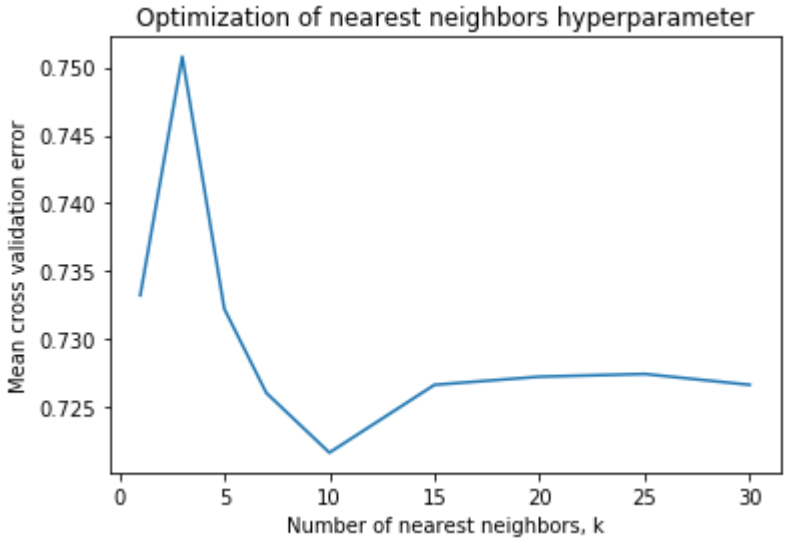
First, we will create the training and validation folds for use in k-fold cross validation.

In [93]:
```
# 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]
# ================================================================ #
indices = np.random.permutation(X_train.shape[0])
X_shuffled = X_train[indices]
y_shuffled = y_train[indices]
X_train_folds = np.split(X_shuffled, num_folds)
y_train_folds = np.split(y_shuffled, num_folds)
# ================================================================ #
# 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 [94]:
```
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.
# ================================================================ #
X_train_folds = np.asarray(X_train_folds)
y_train_folds = np.asarray(y_train_folds)
knn = KNN()
errors = []
for k in ks:
    error = 0
    for i in range(num_folds):
        X_train_subset = np.concatenate([X_train_folds[j] \
                                         for j in range(num_folds) \
                                         if i != j])
        y_train_subset = np.concatenate([y_train_folds[j] \
                                         for j in range(num_folds) \
                                         if i != j])
        knn.train(X_train_subset, y_train_subset)
        dist = knn.compute_L2_distances_vectorized(X_train_folds[i])
        y_pred_fold = knn.predict_labels(dist, k)
        y_test_fold = y_train_folds[i]
        N = len(y_pred_fold)
        num_incorrect = len([y_pred_fold[i] for i in range(N) \
                             if y_test_fold[i] != y_pred_fold[i]])
        error += num_incorrect / N
    errors.append(error / num_folds)

plt.plot(ks, errors)
plt.xlabel('Number of nearest neighbors, k')
plt.ylabel('Mean cross validation error')
plt.title('Optimization of nearest neighbors hyperparameter')
plt.show()

min_error = 10000000
best_k = -1
for index, err in enumerate(errors):
    if err < min_error:
        best_k = ks[index]
        min_error = err
print(f"Best error is {err} with k = {best_k}")

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

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




```
Best error is 0.7266 with k = 10
Computation time: 57.44
```

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

Question 1
We typically pick k as an odd number so that we don't have to worry about ties. Otherwise, for ties, we need to pick a class randomly. It is probably better to have the prediction be deterministic.

Question 2
k = 10 was found to be the optimal value, and it gave an average cross validation error of 0.7266.

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 [95]:
```
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-validatio
n.
# ================================================================ #
print(f"Using k={best_k}")
knn.train(X_train, y_train)
dist = knn.compute_L2_distances_vectorized(X_test)
y_pred = knn.predict_labels(dist, best_k)
N = len(y_pred)
num_incorrect = len([y_pred[i] for i in range(N) \
                     if y_test[i] != y_pred[i]])
error = num_incorrect / N

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

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

```
Using k=10
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

Naively choosing $k = 1$ resulted in an error rate of 0.726, while changing it to $k = 10$ resulted in a slightly lower error rate of 0.718 (L2 norm used for both), which makes sense, as $k = 10$ had a lower average cross validation error. However, this is not much of an improvement - only 0.008 less, and it could be that kNN does not work well on such a dataset. This could be because different objects have relatively similar shapes (such as a dog and horse, both 4 legged), so kNN has trouble distinguishing them.

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.

```python
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] = np.linalg.norm(np.subtract(X[i], self.X_train[j]))

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

```
66
67          return dists
68
69      def compute_L2_distances_vectorized(self, X):
70          """
71          Compute the distance between each test point in X and each training point
72          in self.X_train WITHOUT using any for loops.
73
74          Inputs:
75          - X: A numpy array of shape (num_test, D) containing test data.
76
77          Returns:
78          - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
79            is the Euclidean distance between the ith test point and the jth training
80            point.
81          """
82          num_test = X.shape[0]
83          num_train = self.X_train.shape[0]
84          dists = np.zeros((num_test, num_train))
85
86          # ================================================================ #
87          # START YOUR CODE HERE
88          # ================================================================ #
89          #   Compute the L2 distance between the ith test point and the jth
90          #   training point and store the result in dists[i, j].  You may
91          #    NOT use a for loop (or list comprehension).  You may only use
92          #     numpy operations.
93          #
94          #     HINT: use broadcasting.  If you have a shape (N,1) array and
95          #   a shape (M,) array, adding them together produces a shape (N, M)
96          #   array.
97          # ================================================================ #
98
99          #use the formula |a - b| = sqrt(|a| + |b| - 2|a||b|)
100         x_square = np.sum(np.square(X), axis=1)
101         x_train_square = np.sum(np.square(self.X_train), axis=1)
102         inner_product = np.dot(X, self.X_train.T)
103         return np.sqrt(x_square[:, np.newaxis] + x_train_square - 2*inner_product)
104
105         # ================================================================ #
106         # END YOUR CODE HERE
107         # ================================================================ #
108
109         return dists
110
111
112     def predict_labels(self, dists, k=1):
113         """
114         Given a matrix of distances between test points and training points,
115         predict a label for each test point.
116
117         Inputs:
118         - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
119           gives the distance betwen the ith test point and the jth training point.
120
121         Returns:
122         - y: A numpy array of shape (num_test,) containing predicted labels for the
123           test data, where y[i] is the predicted label for the test point X[i].
124         """
125         num_test = dists.shape[0]
126         y_pred = np.zeros(num_test)
127         for i in range(num_test):
128             # A list of length k storing the labels of the k nearest neighbors to
129             # the ith test point.
130
131             closest_y = []
```

```
132
133         # ========================================================== #
134         # START YOUR CODE HERE
135         # ========================================================== #
136         #   Use the distances to calculate and then store the labels of
137         #   the k-nearest neighbors to the ith test point.  The function
138         #   numpy.argsort may be useful.
139         #
140         #   After doing this, find the most common label of the k-nearest
141         #   neighbors.  Store the predicted label of the ith training example
142         #   as y_pred[i].  Break ties by choosing the smaller label.
143         # ========================================================== #
144         k_closest = np.argsort(dists[i])[:k]
145         k_closest_labels = [self.y_train[i] for i in k_closest]
146         y_pred[i] = np.bincount(k_closest_labels).argmax()
147
148         # ========================================================== #
149         # END YOUR CODE HERE
150         # ========================================================== #
151       return y_pred
```

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: Danning Yu, UID: 305087992

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).

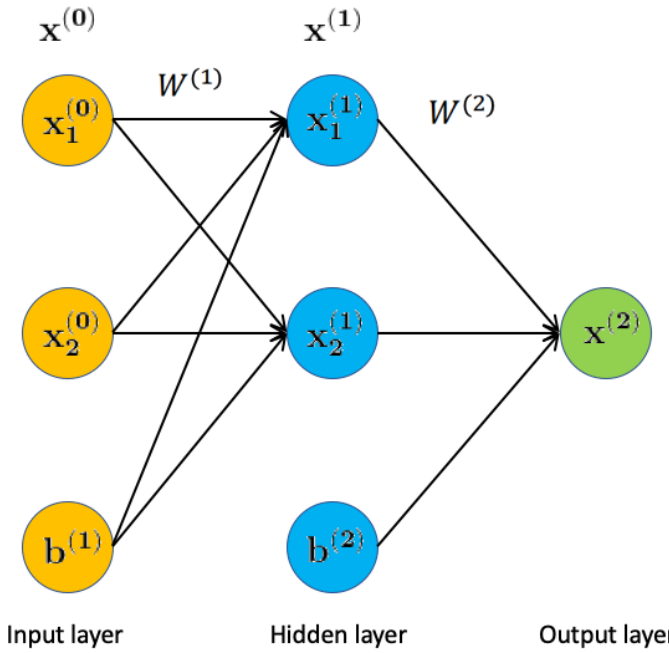
You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

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 `STRART/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 $\mathbf{x}^{(l)} = \mathbf{f}^{(l)}(\mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})$. 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:

Question 1

$$\mathbf{x}^{(1)} = f^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}^{(0)} + \mathbf{b}^{(1)}\right)$$

$$\mathbf{x}^{(1)} = \sigma\left(\begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix}\begin{bmatrix} 0.05 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix}\right)$$

$$\mathbf{x}^{(1)} = \begin{bmatrix} 0.59327 \\ 0.59688 \end{bmatrix}$$

Question 2

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

$$\mathbf{x}^{(1)} = \sigma\left(\begin{bmatrix} 0.4 & 0.45 \end{bmatrix}\begin{bmatrix} 0.59327 \\ 0.59688 \end{bmatrix} + 0.6\right)$$

$$\mathbf{x}^{(2)} = 0.75136$$

Question 3

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||^2$$

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}(0.01 - 0.75136)^2$$

$$l(\mathbf{y}, \hat{\mathbf{y}}) = 0.27481$$

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:

Question 1

Note the derivative of the sigmoid function:

$$f'(z) = \sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right)$$

For the 1st element of $\mathbf{W}^{(2)}$:

$$\frac{\partial l}{\partial \mathbf{W}_1^{(2)}} = \frac{\partial l}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \mathbf{W}_1^{(2)}}$$

$$\frac{\partial l}{\partial \mathbf{W}_1^{(2)}} = -\left(y - x^{(2)}\right) f'^{(2)}\left(z^{(2)}\right) \mathbf{x}_1^{(1)} = \delta^{(2)} \mathbf{x}_1^{(1)}$$

$$\frac{\partial l}{\partial \mathbf{W}_1^{(2)}} = -\left(0.01 - 0.75136\right)\left(0.75136\right)\left(1 - 0.75136\right)\left(0.59327\right)$$

$$\frac{\partial l}{\partial \mathbf{W}_1^{(2)}} = 0.13850(0.59327)$$

$$\frac{\partial l}{\partial \mathbf{W}_1^{(2)}} = 0.08217$$

$$\delta^{(2)} = 0.13850$$

For the 2nd element of $\mathbf{W}^{(2)}$:

$$\frac{\partial l}{\partial \mathbf{W}_2^{(2)}} = \frac{\partial l}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \mathbf{W}_2^{(2)}}$$

Now note that:

$$\frac{\partial l}{\partial \mathbf{W}_2^{(2)}} = \delta^{(2)} \mathbf{x}_2^{(1)}$$

$$\frac{\partial l}{\partial \mathbf{W}_2^{(2)}} = 0.13850(0.59688) = 0.08267$$

Thus, $\frac{\partial l}{\partial \mathbf{W}^{(2)}} = \begin{bmatrix} 0.08217 & 0.08267 \end{bmatrix}$

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

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

$$\frac{\partial l}{\partial \mathbf{b}^{(2)}} = 0.13850$$

Question 2

We use the same chain rule logic but go one layer deeper.

$$\frac{\partial l}{\partial \mathbf{W}_{11}^{(1)}} = \sum_i \frac{\partial l}{\partial x_i^{(2)}} \frac{\partial x_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial \mathbf{x}_1^{(1)}} \frac{\partial \mathbf{x}_1^{(1)}}{\partial \mathbf{z}_1^{(1)}} \frac{\partial \mathbf{z}_1^{(1)}}{\partial \mathbf{W}_{11}^{(1)}}$$

Note that there is only 1 term in the output layer, so we can drop the summation.

$$\frac{\partial l}{\partial \mathbf{W}_{11}^{(1)}} = \frac{\partial l}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \mathbf{x}_1^{(1)}} \frac{\partial \mathbf{x}_1^{(1)}}{\partial \mathbf{z}_1^{(1)}} \frac{\partial \mathbf{z}_1^{(1)}}{\partial \mathbf{W}_{11}^{(1)}}$$

$$\frac{\partial l}{\partial \mathbf{W}_{11}^{(1)}} = -\left(y - x^{(2)}\right) f'^{(2)}\left(z^{(2)}\right) \mathbf{W}_1^{(2)} f'^{(1)}\left(\mathbf{z}_1^{(1)}\right) \mathbf{x}_1^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{11}^{(1)}} = \delta^{(2)} \mathbf{W}_1^{(2)} f'^{(1)}\left(\mathbf{z}_1^{(1)}\right) \mathbf{x}_1^{(0)} = \delta_1^{(1)} \mathbf{x}_1^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{11}^{(1)}} = (0.13850)(0.4)(0.59327)(1 - 0.59327)(0.05) = 0.01337(0.05)$$

$$\frac{\partial l}{\partial \mathbf{W}_{11}^{(1)}} = 0.00067$$

$$\delta_1^{(1)} = 0.01337$$

Now repeat for the other 3 elements of $\mathbf{W}^{(1)}$:

$$\frac{\partial l}{\partial \mathbf{W}_{12}^{(1)}} = \frac{\partial l}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \mathbf{x}_1^{(1)}} \frac{\partial \mathbf{x}_1^{(1)}}{\partial \mathbf{z}_1^{(1)}} \frac{\partial \mathbf{z}_1^{(1)}}{\partial \mathbf{W}_{12}^{(1)}}$$

$$\frac{\partial l}{\partial \mathbf{W}_{12}^{(1)}} = -\left(y - x^{(2)}\right) f'^{(2)}\left(z^{(2)}\right) \mathbf{W}_1^{(2)} f'^{(1)}\left(\mathbf{z}_1^{(1)}\right) \mathbf{x}_2^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{12}^{(1)}} = \delta_1^{(1)} \mathbf{x}_2^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{12}^{(1)}} = 0.01337(0.1) = 0.00134$$

$$\frac{\partial l}{\partial \mathbf{W}_{21}^{(1)}} = \frac{\partial l}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \mathbf{x}_2^{(1)}} \frac{\partial \mathbf{x}_2^{(1)}}{\partial \mathbf{z}_2^{(1)}} \frac{\partial \mathbf{z}_2^{(1)}}{\partial \mathbf{W}_{21}^{(1)}}$$

$$\frac{\partial l}{\partial \mathbf{W}_{21}^{(1)}} = -\left(y - x^{(2)}\right) f'^{(2)}\left(z^{(2)}\right) \mathbf{W}_2^{(2)} f'^{(1)}\left(\mathbf{z}_2^{(1)}\right) \mathbf{x}_1^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{21}^{(1)}} = \delta^{(2)} \mathbf{W}_2^{(2)} f'^{(1)}\left(\mathbf{z}_2^{(1)}\right) \mathbf{x}_1^{(0)} = \delta_2^{(1)} \mathbf{x}_1^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{21}^{(1)}} = (0.13850)(0.45)(0.59688)(1 - 0.59688)(0.05) = 0.01500(0.05)$$

$$\frac{\partial l}{\partial \mathbf{W}_{21}^{(1)}} = 0.00075$$

$$\delta_2^{(1)} = 0.01500$$

$$\frac{\partial l}{\partial \mathbf{W}_{22}^{(1)}} = \frac{\partial l}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \mathbf{x}_2^{(1)}} \frac{\partial \mathbf{x}_2^{(1)}}{\partial \mathbf{z}_2^{(1)}} \frac{\partial \mathbf{z}_2^{(1)}}{\partial \mathbf{W}_{22}^{(1)}}$$

$$\frac{\partial l}{\partial \mathbf{W}_{22}^{(1)}} = -\left(y - x^{(2)}\right) f'^{(2)}\left(z^{(2)}\right) \mathbf{W}_2^{(2)} f'^{(1)}\left(\mathbf{z}_2^{(1)}\right) \mathbf{x}_2^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{22}^{(1)}} = \delta^{(2)} \mathbf{W}_2^{(2)} f'^{(1)}\left(\mathbf{z}_2^{(1)}\right) \mathbf{x}_1^{(0)} = \delta_2^{(1)} \mathbf{x}_2^{(0)}$$

$$\frac{\partial l}{\partial \mathbf{W}_{22}^{(1)}} = (0.01500)(0.1)$$

$$\frac{\partial l}{\partial \mathbf{W}_{22}^{(1)}} = 0.00150$$

Combining our results for the 4 elements gives:

$$\frac{\partial l}{\partial \mathbf{W}^{(1)}} = \begin{bmatrix} 0.00067 & 0.00134 \\ 0.00075 & 0.00150 \end{bmatrix}$$

Now for $\frac{\partial l}{\partial \mathbf{b}^{(1)}}$:

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

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

Note that this is simply equal to $\boldsymbol{\delta}^{(1)}$, so:

$$\frac{\partial l}{\partial \mathbf{b}^{(1)}} = \begin{bmatrix} 0.01337 \\ 0.01500 \end{bmatrix}$$

Section 2: Coding a two-layer neural network

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

In [2]:
```
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.ab
s(y))))
```

Toy example

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

In [3]:
```
from hw3code.neural_net import TwoLayerNet
```

In [4]:
```
# 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 wh
y
# 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:
3.381231191562639e-08
```

Forward pass loss

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

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

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 [5]:
```
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 [6]:
```
from data.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the b
ackward pass.
# If your implementation is correct, the difference between the numer
ic 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(pa
ram_grad_num, grads[param_name])))
```

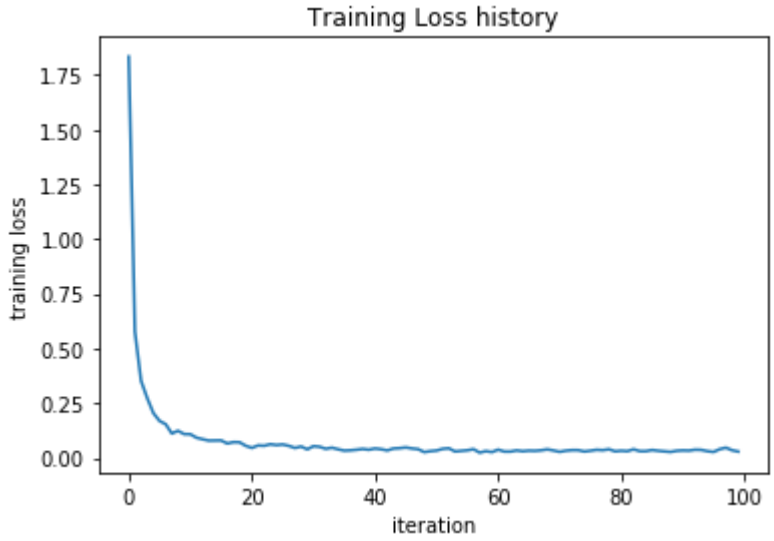
```
W2 max relative error: 8.800911222099066e-11
b2 max relative error: 2.4554844805570154e-11
W1 max relative error: 1.7476674265765602e-09
b1 max relative error: 7.382451041178829e-10
```

Training the network

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

In [7]:
```
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 [5]:
```
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%.

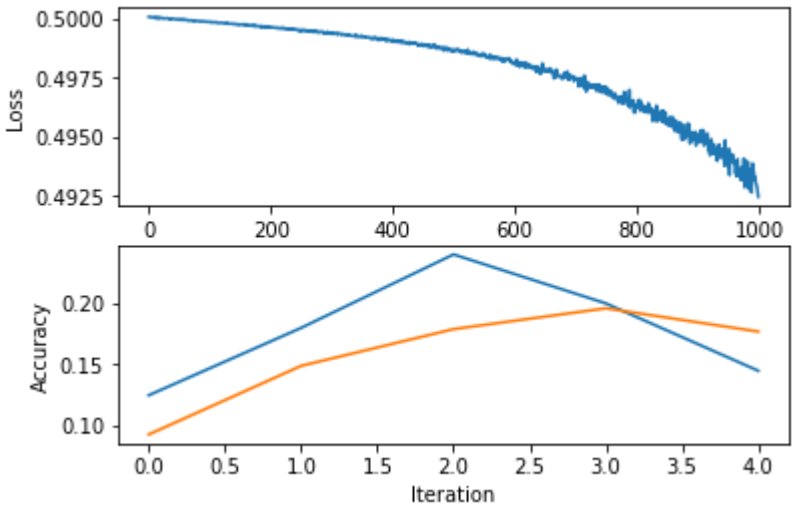
In [20]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# 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 0.5000846394148108
iteration 100 / 1000: loss 0.4998658595822322
iteration 200 / 1000: loss 0.4996269495847538
iteration 300 / 1000: loss 0.499407932435397
iteration 400 / 1000: loss 0.4990903682340476
iteration 500 / 1000: loss 0.49870038950888085
iteration 600 / 1000: loss 0.49806081336114993
iteration 700 / 1000: loss 0.4975002473802068
iteration 800 / 1000: loss 0.49623953096340356
iteration 900 / 1000: loss 0.49482064134195486
Validation accuracy:  0.173
Test accuracy (subopt_net):  0.184
```

In [21]:
```
stats['train_acc_history']
```

Out[21]:
```
[0.125, 0.18, 0.24, 0.2, 0.145]
```

In [22]:
```
# 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:

Question 1
One reason this could be the case is that the neural network does not have enough layers. Images contain a lot of data that can have complex relationships, and so having more layers would allow us to handle the data with more expressive or nonlinear functions. Another possible reason is that we haven't tuned our hyperparameters. Things like the regularization amount (lambda), gradient batch size, learning rate, and learning_rate decay were all set and not further tuned. Also, we should tune the number of neurons in the hidden layer. Hyperparmeters are an important part of getting good performance from a model, and so we should do a grid search to tune them. Finally, 1000 iterations may not be enough for convergence, as the graph above shows that the loss was still going down after 1000 iterations.

Question 2
For the hyperparameters, it would be good to do a grid search for the proper hyperparameters. Try all various combinations of gradient batch size, regularization strength, learning rate, and learning rate decay. For the number of hidden layers, we could solve this by increasing the number of hidden layers, treating it as a hyperparameter and finding the optimal # of hidden layers. Finally, to ensure the loss converges, we should increase the number of iterations.

Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net. 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 [18]:
```
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_siz
e = 50)!
# ================================================================ #

# todo: optimal parameter search (you may use grid search by for-loop
s )
best_params_and_res = None
best_valacc = 0
i = 0
for batch_size in [500, 750, 1000]:
    for learning_rate in [1e-4, 1e-3]:
        for regularization in [0.5, 1, 2]:
            for decay in [0.8, 0.6, 0.5]:
                print(f"Grid search #{i}")
                stats = net.train(X_train, y_train, X_val, y_val,
                                  num_iters=1000,
                                  batch_size=batch_size,
                                  learning_rate=learning_rate,
                                  learning_rate_decay=decay,
                                  reg=regularization, verbose=False)
                # Predict on the validation set
                val_acc = (net.predict(X_val) == y_val).mean()
                if val_acc > best_valacc:
                    best_net = net
                    best_valacc = val_acc
                    print('New best validation accuracy: ', best_vala
cc)
                    best_params_and_res = \
                        (f"batch_size: {batch_size}, "
                         f"learning_rate: {learning_rate}, "
                         f"learning_rate_decay: {decay}, "
                         f"regularization: {regularization}, "
                         f"validation accuracy: {best_valacc}")
                    print(best_params_and_res)
                i += 1


# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
# Output your results
print("== Best parameter settings ==")
```

```
print(best_params_and_res)
print("Best accuracy on validation set: {}".format(best_valacc))
```

```
Grid search #0
New best validation accuracy:  0.268
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.8, reg
ularization: 0.5, validation accuracy: 0.268
Grid search #1
New best validation accuracy:  0.285
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.6, reg
ularization: 0.5, validation accuracy: 0.285
Grid search #2
New best validation accuracy:  0.299
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.5, reg
ularization: 0.5, validation accuracy: 0.299
Grid search #3
New best validation accuracy:  0.342
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.8, reg
ularization: 1, validation accuracy: 0.342
Grid search #4
New best validation accuracy:  0.357
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.6, reg
ularization: 1, validation accuracy: 0.357
Grid search #5
New best validation accuracy:  0.362
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.5, reg
ularization: 1, validation accuracy: 0.362
Grid search #6
New best validation accuracy:  0.38
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.8, reg
ularization: 2, validation accuracy: 0.38
Grid search #7
New best validation accuracy:  0.389
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.6, reg
ularization: 2, validation accuracy: 0.389
Grid search #8
New best validation accuracy:  0.396
batch_size: 500, learning_rate: 0.0001, learning_rate_decay: 0.5, reg
ularization: 2, validation accuracy: 0.396
Grid search #9
New best validation accuracy:  0.476
batch_size: 500, learning_rate: 0.001, learning_rate_decay: 0.8, regu
larization: 0.5, validation accuracy: 0.476
Grid search #10
New best validation accuracy:  0.478
batch_size: 500, learning_rate: 0.001, learning_rate_decay: 0.6, regu
larization: 0.5, validation accuracy: 0.478
Grid search #11
New best validation accuracy:  0.48
batch_size: 500, learning_rate: 0.001, learning_rate_decay: 0.5, regu
larization: 0.5, validation accuracy: 0.48
Grid search #12
New best validation accuracy:  0.496
batch_size: 500, learning_rate: 0.001, learning_rate_decay: 0.8, regu
larization: 1, validation accuracy: 0.496
Grid search #13
Grid search #14
Grid search #15
Grid search #16
Grid search #17
```

```
Grid search #18
Grid search #19
Grid search #20
New best validation accuracy:  0.499
batch_size: 750, learning_rate: 0.0001, learning_rate_decay: 0.5, reg
ularization: 0.5, validation accuracy: 0.499
Grid search #21
New best validation accuracy:  0.502
batch_size: 750, learning_rate: 0.0001, learning_rate_decay: 0.8, reg
ularization: 1, validation accuracy: 0.502
Grid search #22
Grid search #23
Grid search #24
Grid search #25
Grid search #26
Grid search #27
Grid search #28
Grid search #29
Grid search #30
Grid search #31
Grid search #32
New best validation accuracy:  0.505
batch_size: 750, learning_rate: 0.001, learning_rate_decay: 0.5, regu
larization: 1, validation accuracy: 0.505
Grid search #33
Grid search #34
Grid search #35
Grid search #36
Grid search #37
Grid search #38
Grid search #39
Grid search #40
Grid search #41
Grid search #42
Grid search #43
Grid search #44
Grid search #45
New best validation accuracy:  0.506
batch_size: 1000, learning_rate: 0.001, learning_rate_decay: 0.8, reg
ularization: 0.5, validation accuracy: 0.506
Grid search #46
Grid search #47
Grid search #48
New best validation accuracy:  0.51
batch_size: 1000, learning_rate: 0.001, learning_rate_decay: 0.8, reg
ularization: 1, validation accuracy: 0.51
Grid search #49
New best validation accuracy:  0.512
batch_size: 1000, learning_rate: 0.001, learning_rate_decay: 0.6, reg
ularization: 1, validation accuracy: 0.512
Grid search #50
Grid search #51
Grid search #52
Grid search #53
== Best parameter settings ==
batch_size: 1000, learning_rate: 0.001, learning_rate_decay: 0.6, reg
```

```
ularization: 1, validation accuracy: 0.512
Best accuracy on validation set: 0.512
```

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

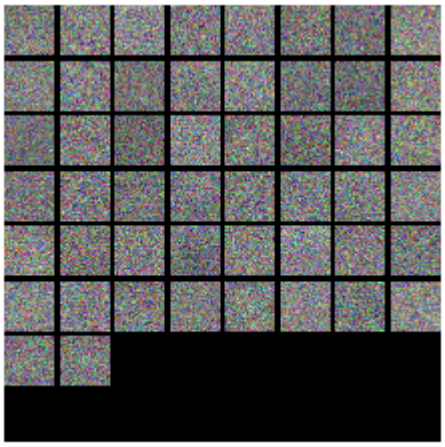
Question 1
The best parameter settings from my grid search of [500, 750, 1000] for gradient descent batch size, [1e-4, 1e-3] for learning rate, [0.5, 1, 2, 5] for regularization, and [0.8, 0.6, 0.5] for learning decay rate was:
Gradient descent batch size: 1000
Learning rate: 1e-3 (or 0.001)
Learning rate decay: 0.6
Regularization: 1

These parameters resulted in a validation accuracy of 0.512.

Question 2
I tuned learning rate, learning rate decay, gradient descent batch size, and regularization strength. Learning rate determines how big of a update we make for each iteration of gradient descent, and so it looks like increasing it in comparison to the previous value of 1e-4 caused it to converge faster to the optimal value. The learning rate decay of 0.5, which is lower than the original value of 0.95, causes the learning rate to decrease more rapidly, so as we go through more iterations and we get closer to the optimal neural network parameters, it reduces how much the parameters are updated. The batch size increased from 500 to 750, reflecting the fact that using more points results in a better gradient descent update. Finally, increasing the regularization from 0.1 to 1 probably helped prevent overfitting.

Visualize the weights of your neural networks

In [23]:
```
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 best net are a lot more clearly defined than that for the suboptimal net, which just looks like pure static. In the best net, it is evident that certain features are being emphasized and more visible--these are features or shapes that the NN has been able to pick out of the image. Thus, the weights in the NN are learning particular attributes or features contained within the image that help it to better make a classification.

Evaluate on test set

In [24]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy (best_net): ', test_acc)
```

```
Test accuracy (best_net):  0.491
```

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:

Question 1
Using the best NN I got, the test accuracy is 0.491. This is a lot better than the best kNN with k=10, which had a test accuracy of 1 - 0.718 = 0.282. This is approximately a 70% increase in accuracy. NNs perform a lot better than kNNs because they are a lot complex of a model, and thus they can better express the complexity inherent in classifying images. Images have many attributes that may not be visible at first glance, and kNN can only classify based on class, while NNs have hidden layers that allow it to make the distinctions between various attributes. This is evident in the images of the weights - the NN one has features that are a lot more clearly defined that that of the kNN model.

Question 2
It would probably be good to add more hidden layers, as images are complex and the NN would probably benefit from hidden layers. This will allow it to find more attributes to distinguish different objects from each other. Furthermore, more training data would be helpful, especially as the model gets more complex.

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,

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

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.

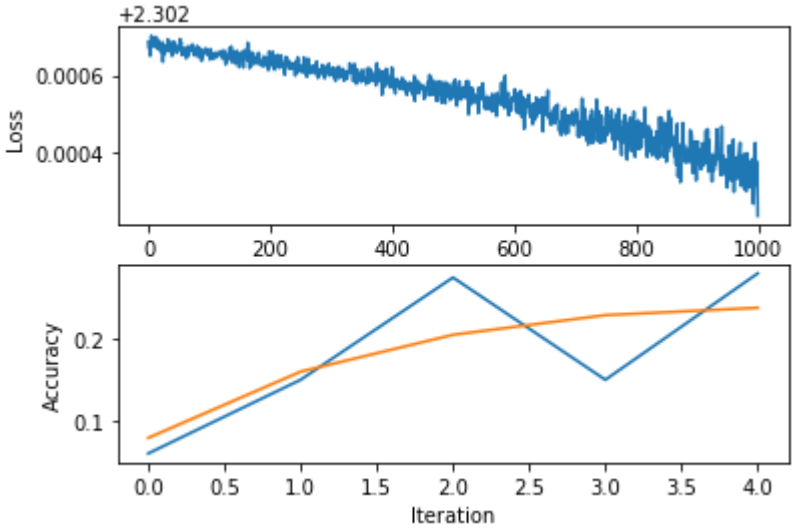
Change the loss from MSE to cross entropy, you only need to change you `MSE_loss(x,y)` in `TwoLayerNet.loss()` function to `softmax_loss(x,y)`.

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 = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network - use same parameters as subopt_net
print("Training with same parameters as subopt_net")
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=False)

# 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)

stats['train_acc_history']

# 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()


# Now retrain with some hyperparameter tuning
print("=====Retraining with optimized hyperparameters=====")
best_net = None
best_test_acc = 0
best_stats = None
best_params_res = ""
for batch_size in [400, 500, 600]:
    for reg in [0.1, 0.5, 1]:
        stats = net.train(X_train, y_train, X_val, y_val,
                    num_iters=1000, batch_size=batch_size,
                    learning_rate=5e-3, learning_rate_decay=0.5,
                    reg=reg, verbose=True)
        # Predict on the validation set
        val_acc = (net.predict(X_val) == y_val).mean()
```

```
        print('Validation accuracy: ', val_acc)
        test_acc = (net.predict(X_test) == y_test).mean()
        print('Test accuracy (best_net): ', test_acc)
        if test_acc > best_test_acc:
            best_test_acc = test_acc
            best_net = net
            best_stats = stats
            best_params_res = f"batch size={batch_size}, " \
                              f"regularization={reg}, " \
                              f"learning rate=5e-3, " \
                              f"learning rate decay=0.5, " \
                              f"test accuracy={test_acc}"
            print(f"New best: {best_params_res}")

best_stats['train_acc_history']

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

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

plt.show()
print(f"Best result: {best_params_res}")
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

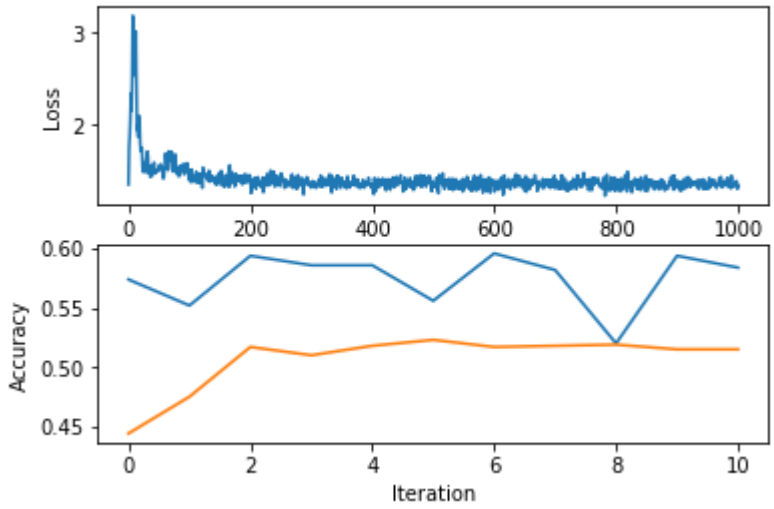
```
Training with same parameters as subopt_net
Validation accuracy:  0.24
Test accuracy (subopt_net):  0.252
```



```
=====Retraining with optimized hyperparameters=====
iteration 0 / 1000: loss 2.3022833539320606
iteration 100 / 1000: loss 1.802777982500361
iteration 200 / 1000: loss 1.6479725838240051
iteration 300 / 1000: loss 1.4993106943744123
iteration 400 / 1000: loss 1.4431336743793697
iteration 500 / 1000: loss 1.4414808159060313
iteration 600 / 1000: loss 1.4294678753425005
iteration 700 / 1000: loss 1.445247607772252
iteration 800 / 1000: loss 1.3389924074485973
iteration 900 / 1000: loss 1.3877887046327069
Validation accuracy:  0.503
Test accuracy (best_net):  0.497
New best: batch size=400, regularization=0.1, learning rate=5e-3, lea
rning rate decay=0.5, test accuracy=0.497
iteration 0 / 1000: loss 1.393111736830811
iteration 100 / 1000: loss 1.633641457435265
iteration 200 / 1000: loss 1.716029376356005
iteration 300 / 1000: loss 1.504401080028609
iteration 400 / 1000: loss 1.3844003187358411
iteration 500 / 1000: loss 1.3024170948339702
iteration 600 / 1000: loss 1.4259542484204488
iteration 700 / 1000: loss 1.4177624745704613
iteration 800 / 1000: loss 1.379620441711755
iteration 900 / 1000: loss 1.3300655052897548
Validation accuracy:  0.514
Test accuracy (best_net):  0.52
New best: batch size=400, regularization=0.5, learning rate=5e-3, lea
rning rate decay=0.5, test accuracy=0.52
iteration 0 / 1000: loss 1.4806313243486626
iteration 100 / 1000: loss 2.05358820248194
iteration 200 / 1000: loss 1.8796935321566377
iteration 300 / 1000: loss 1.523037761645025
iteration 400 / 1000: loss 1.5327950833283734
iteration 500 / 1000: loss 1.4732577173990362
iteration 600 / 1000: loss 1.4597717442491709
iteration 700 / 1000: loss 1.533984821605085
iteration 800 / 1000: loss 1.4375630029081268
iteration 900 / 1000: loss 1.3951387476298964
Validation accuracy:  0.511
Test accuracy (best_net):  0.527
New best: batch size=400, regularization=1, learning rate=5e-3, learn
ing rate decay=0.5, test accuracy=0.527
iteration 0 / 1000: loss 1.3279803756048005
iteration 100 / 1000: loss 1.4385638029040257
iteration 200 / 1000: loss 1.2653648283177397
iteration 300 / 1000: loss 1.3181632891248936
iteration 400 / 1000: loss 1.2996351193505868
iteration 500 / 1000: loss 1.3100500985425718
iteration 600 / 1000: loss 1.2541479359967007
iteration 700 / 1000: loss 1.262822071293243
iteration 800 / 1000: loss 1.215029434469136
iteration 900 / 1000: loss 1.3187480945773267
Validation accuracy:  0.515
Test accuracy (best_net):  0.53
New best: batch size=500, regularization=0.1, learning rate=5e-3, lea
rning rate decay=0.5, test accuracy=0.53
```

```
iteration 0 / 1000: loss 1.3596195366781234
iteration 100 / 1000: loss 1.4060186717603382
iteration 200 / 1000: loss 1.3750465331965076
iteration 300 / 1000: loss 1.3906298451663615
iteration 400 / 1000: loss 1.3292705284181892
iteration 500 / 1000: loss 1.3417396523421619
iteration 600 / 1000: loss 1.2818927103207352
iteration 700 / 1000: loss 1.4107123077173465
iteration 800 / 1000: loss 1.3485830199550464
iteration 900 / 1000: loss 1.3465563816126676
Validation accuracy:  0.515
Test accuracy (best_net):  0.538
New best: batch size=500, regularization=0.5, learning rate=5e-3, lea
rning rate decay=0.5, test accuracy=0.538
iteration 0 / 1000: loss 1.4611275422597338
iteration 100 / 1000: loss 1.5076056924211583
iteration 200 / 1000: loss 1.444147747100542
iteration 300 / 1000: loss 1.5055977287145095
iteration 400 / 1000: loss 1.4040928808156403
iteration 500 / 1000: loss 1.573058668453538
iteration 600 / 1000: loss 1.4687925025319046
iteration 700 / 1000: loss 1.4685567682545164
iteration 800 / 1000: loss 1.467366747305646
iteration 900 / 1000: loss 1.4186662214611265
Validation accuracy:  0.52
Test accuracy (best_net):  0.537
iteration 0 / 1000: loss 1.325512369901725
iteration 100 / 1000: loss 1.8772394468423275
iteration 200 / 1000: loss 1.5860791416147546
iteration 300 / 1000: loss 1.4252503535540886
iteration 400 / 1000: loss 1.5485913647529044
iteration 500 / 1000: loss 1.3630480426147313
iteration 600 / 1000: loss 1.5242327672836784
iteration 700 / 1000: loss 1.3604041808419374
iteration 800 / 1000: loss 1.4336488626533934
iteration 900 / 1000: loss 1.4106290170211817
Validation accuracy:  0.501
Test accuracy (best_net):  0.501
iteration 0 / 1000: loss 1.4942643449417319
iteration 100 / 1000: loss 1.5181607558718866
iteration 200 / 1000: loss 1.6213814745826673
iteration 300 / 1000: loss 1.5067113587422745
iteration 400 / 1000: loss 1.633628807147488
iteration 500 / 1000: loss 1.6871606880848922
iteration 600 / 1000: loss 1.7065622525805415
iteration 700 / 1000: loss 1.5042257393186425
iteration 800 / 1000: loss 1.5985477974909281
iteration 900 / 1000: loss 1.5065962140238198
Validation accuracy:  0.498
Test accuracy (best_net):  0.488
iteration 0 / 1000: loss 1.6227560735451187
iteration 100 / 1000: loss 1.7233258291746316
iteration 200 / 1000: loss 1.6612769672655663
iteration 300 / 1000: loss 1.8822272619551539
iteration 400 / 1000: loss 1.5839562803081122
iteration 500 / 1000: loss 1.678761016104926
iteration 600 / 1000: loss 1.6459737606926867
```

```
iteration 700 / 1000: loss 1.64452914902552
iteration 800 / 1000: loss 1.449956250870989
iteration 900 / 1000: loss 1.7697185685482184
Validation accuracy:  0.468
Test accuracy (best_net):  0.477
```




```
Best result: batch size=500, regularization=0.5, learning rate=5e-3,
learning rate decay=0.5, test accuracy=0.538
```

We can see that when the same parameters as the suboptimal net are used, the test accuracy is 0.252 for softmax error, versus 0.184 for when MSE error is used. When I tuned the hyperparameters, the best test accuracy I got was 0.538 (for parameter details, see output of cell above), which is higher than the 0.491 that I got using MSE error. The graphs also show the loss converging quite rapidly to around 1.35. Thus, it seems like softmax classification leads to slightly better performance on this dataset when using a neural network.

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.

```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
```

```
68        W1, b1 = self.params['W1'], self.params['b1']
69        W2, b2 = self.params['W2'], self.params['b2']
70        N, D = X.shape
71
72        # Compute the forward pass
73        scores = None
74
75        # ============================================================ #
76        # START YOUR CODE HERE
77        # ============================================================ #
78        #   Calculate the output scores of the neural network.  The result
79        #   should be (N, C). As stated in the description for this class,
80        #   there should not be a ReLU layer after the second fully-connected
81        #   layer.
82        #   The code is partially given
83        #   The output of the second fully connected layer is the output scores.
84        #   Do not use a for loop in your implementation.
85        #   Please use 'h1' as input of hidden layers, and 'a2' as output of
86        #   hidden layers after ReLU activation function.
87        #   [Input X] --W1,b1--> [h1] -ReLU-> [a2] --W2,b2--> [scores]
88        #   You may simply use np.maximun for implementing ReLU.
89        #   Note that there is only one ReLU layer.
90        #   Note that plase do not change the variable names (h1, h2, a2)
91        # ============================================================ #
92
93        h1 = np.matmul(X, W1.T) + b1
94        a2 = np.maximum(h1, np.zeros(h1.shape))
95        h2 = np.matmul(a2, W2.T) + b2
96        scores = h2
97
98        # ============================================================ #
99        # END YOUR CODE HERE
100       # ============================================================ #
101
102
103       # If the targets are not given then jump out, we're done
104       if y is None:
105           return scores
106
107       # Compute the loss
108       loss = None
109
110       # scores is num_examples by num_classes (N, C)
111       def softmax_loss(x, y):
112           loss, dx = 0,0
113           # ============================================================ #
114           # START YOUR CODE HERE (BONUS QUESTION)
115           # ============================================================ #
116           #   Calculate the cross entropy loss after softmax output layer.
117           #   The format are provided in the notebook.
118           #   This function should return loss and dx, same as MSE loss function.
119           # ============================================================ #
120           # x is predicted, y is actual
121
122           N = x.shape[0]
123
124           x_exps = np.zeros(x.shape)
125           for i in range(N):
126               largest_x = np.max(x[i])
127               exp_row = np.exp(x[i] - largest_x)
128               x_exps[i] = exp_row / np.sum(exp_row)
129           log_likelihood = -np.log(x_exps[range(N),y])
130           loss = np.sum(log_likelihood) / N
131           dx = np.zeros(x.shape)

132           dx = np.add(dx, x_exps)
133           dx[range(N),y] -= 1
134           dx = dx / N
135
```

```
        # ================================================================ #
        # 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 = x.shape[0]
        y_actual = np.zeros(x.shape)
        for i in range(N):
            y_actual[i][y[i]] = 1 # for row i, y[i] should be 1
        loss = np.sum(np.square(np.subtract(x, y_actual))) / (2 * N)
        dx = np.subtract(x, y_actual) / 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 = MSE_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
    # ================================================================ #
```

```
204
205         return loss, grads
206
207     def train(self, X, y, X_val, y_val,
208               learning_rate=1e-3, learning_rate_decay=0.95,
209               reg=1e-5, num_iters=100,
210               batch_size=200, verbose=False):
211         """
212         Train this neural network using stochastic gradient descent.
213
214         Inputs:
215         - X: A numpy array of shape (N, D) giving training data.
216         - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
217           X[i] has label c, where 0 <= c < C.
218         - X_val: A numpy array of shape (N_val, D) giving validation data.
219         - y_val: A numpy array of shape (N_val,) giving validation labels.
220         - learning_rate: Scalar giving learning rate for optimization.
221         - learning_rate_decay: Scalar giving factor used to decay the learning rate
222           after each epoch.
223         - reg: Scalar giving regularization strength.
224         - num_iters: Number of steps to take when optimizing.
225         - batch_size: Number of training examples to use per step.
226         - verbose: boolean; if true print progress during optimization.
227         """
228         num_train = X.shape[0]
229         iterations_per_epoch = max(num_train / batch_size, 1)
230
231         # Use SGD to optimize the parameters in self.model
232         loss_history = []
233         train_acc_history = []
234         val_acc_history = []
235
236         for it in np.arange(num_iters):
237             X_batch = None
238             y_batch = None
239
240             #  Create a minibatch (X_batch, y_batch) by sampling batch_size
241             #  samples randomly.
242
243             b_index = np.random.choice(num_train, batch_size)
244             X_batch = X[b_index]
245             y_batch = y[b_index]
246
247             # Compute loss and gradients using the current minibatch
248             loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
249             loss_history.append(loss)
250
251             # ========================================================== #
252             # START YOUR CODE HERE
253             # ========================================================== #
254             #  Perform a gradient descent step using the minibatch to update
255             #  all parameters (i.e., W1, W2, b1, and b2).
256             #  The gradient has been calculated as grads['W1'], grads['W2'],
257             #  grads['b1'], grads['b2']
258             #  For example,
259             #  W1(new) = W1(old) - learning_rate * grads['W1']
260             #  (this is not the exact code you use!)
261             # ========================================================== #
262
263             self.params['W1'] = self.params['W1'] - learning_rate * grads['W1']
264             self.params['b1'] = self.params['b1'] - learning_rate * grads['b1']
265             self.params['W2'] = self.params['W2'] - learning_rate * grads['W2']
266             self.params['b2'] = self.params['b2'] - learning_rate * grads['b2']
267
268             # ========================================================== #
269             # END YOUR CODE HERE
270             # ========================================================== #
271
272             if verbose and it % 100 == 0:
```

```
        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
```