knn

October 8, 2023

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'Fall_2023/809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

The kNN classifier consists of two stages:

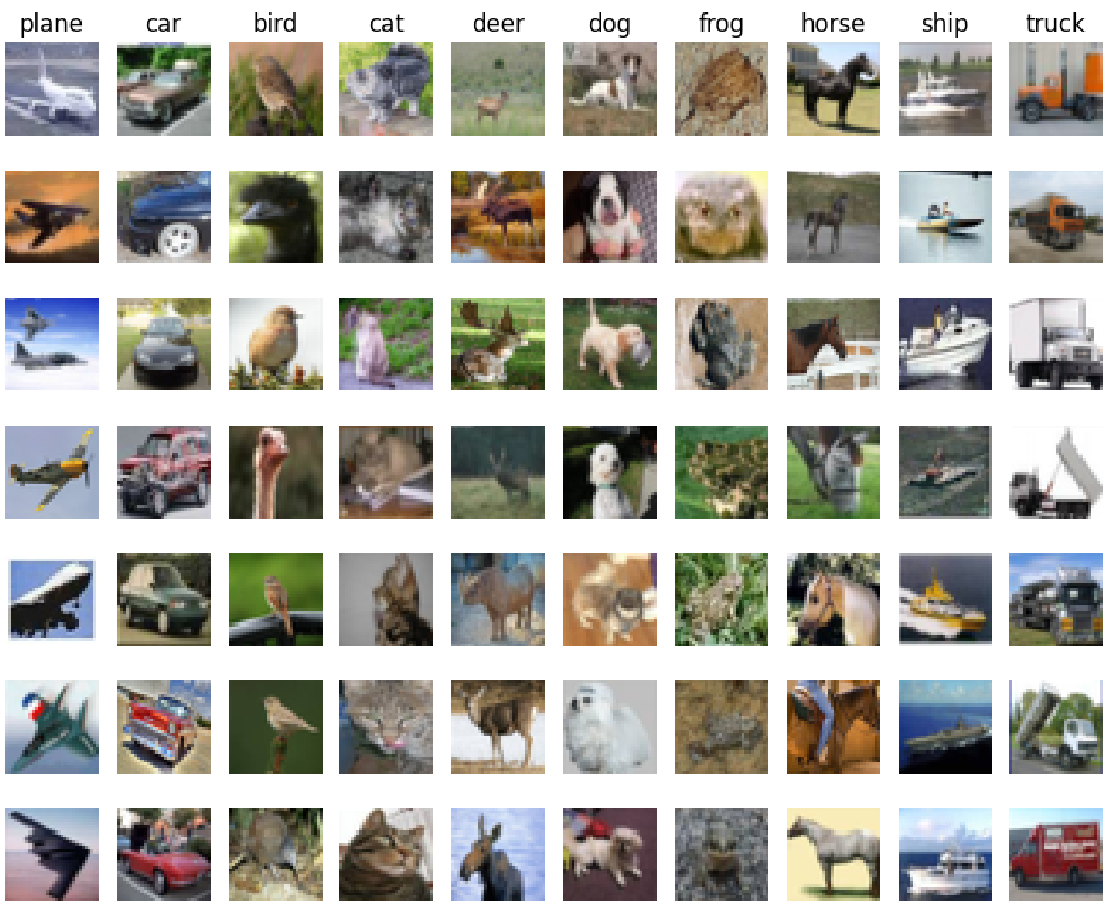
- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

[2]:
```python
# Run some setup code for this notebook.

import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the␣
 ↪notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[3]:
```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause␣
 ↪memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

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.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
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)
# PLEASE DO NOT MODIFY THE MARKERS
print('````````````````````````````````````````````````')
```

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

[4]:

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'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()
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
```
```

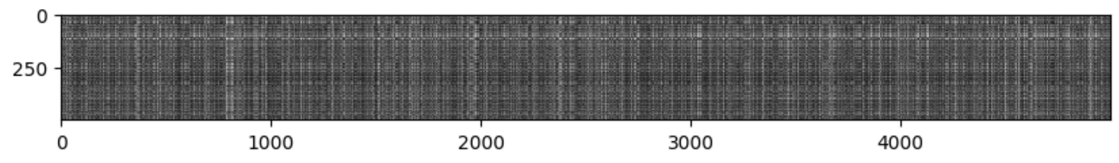


```
`````````````````````````````````````````````

[5]:
```python
# 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))
# PLEASE DO NOT MODIFY THE MARKERS
print('|||||||||||||||||||||||||||||||||||||||||||||||')
print(X_train.shape, X_test.shape)
print('````````````````````````````````````````````````')
```

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

[6]:
```
from cs231n.classifiers import KNearestNeighbor
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
print('``')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
~~~
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

**Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.**

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

[7]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement

# Importing KNearestNeighbor from cs231n/classifiers
from cs231n.classifiers import KNearestNeighbor

# compute_distances_two_loops.
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
# PLEASE DO NOT MODIFY THE MARKERS
```
~~~

```
print('````````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
(500, 5000)
````````````````````````````````````````````````
```

[8]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
plt.imshow(dists, interpolation='none')
plt.show()
# PLEASE DO NOT MODIFY THE MARKERS
print('````````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||||
```


0
250
0
1000
2000
3000
4000

```
~~~
```

Inline Question 1

Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

$\color{blue}{\textit Your Answer:}$

When you see bright rows or columns in the distance matrix, it's like a signal that something interesting or unusual is happening with certain examples in your dataset. Let me break it down:

Bright Rows (Test Examples):

Imagine bright rows as rows of test examples that stand out because they are quite different from the majority of training examples. These test examples might be like outliers or unique cases in your dataset. They have features that are very different from what the algorithm has seen during training, which leads to larger distances. Another reason for bright rows could be that your dataset has some classes that are a bit mixed up or not well-defined. Test examples from these "confusing" classes might have high distances to the training examples of other classes, creating bright rows in the distance matrix.

Bright Columns (Training Examples):
~~~

On the other hand, bright columns represent training examples that are themselves quite different from other examples within their own class. It's like having a few "odd ducks" within a group. These training examples are outliers within their respective classes, which results in higher distances when compared to test examples. Sometimes, bright columns can appear when there aren't enough training examples to fully represent the characteristics of a class. If a class has only a handful of examples or if the data for that class is limited, the distances between test examples and these underrepresented training examples may be higher. This can lead to bright columns for that class in the distance matrix.

In simpler terms, when you spot bright rows, it means there are unique or strange test examples. And when you see bright columns, it's like having some oddballs within a class or not having enough examples to describe a class properly. These patterns in the distance matrix can give you clues about your data, like finding interesting cases or areas that might need more data or attention during analysis.

[9]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
Got 137 / 500 correct => accuracy: 0.274000
``````````````````````````````````````````````
```

You should expect to see approximately 27% accuracy. Now lets try out a larger `k`, say `k = 5`:

[10]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
# PLEASE DO NOT MODIFY THE MARKERS
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
print('``')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
Got 139 / 500 correct => accuracy: 0.278000
~~~
```

You should expect to see a slightly better performance than with `k = 1`.

**Inline Question 2**
~~~

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i, j) of some image I_k,

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw} \sum_{k=1}^{n} \sum_{i=1}^{h} \sum_{j=1}^{w} p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n} \sum_{k=1}^{n} p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.) 2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.) 3. Subtracting the mean μ and dividing by the standard deviation σ. 4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}. 5. Rotating the coordinate axes of the data.

YourAnswer : Options 1, 2, 3, and 4 won't change the Nearest Neighbor classifier's performance, while option 5, rotating the coordinate axes, will change the performance of Nearest Neighbour Classifier that uses L1 distance.

YourExplanation :

Subtracting the mean (p~(k)ij = p(k)ij -):

This step subtracts the overall mean from each pixel value in all images. It centers the data around zero but does not affect the relative differences between pixel values within an image. It does not change the ranking of nearest neighbors based on L1 distance. Therefore, this preprocessing step will not change the performance of the Nearest Neighbor classifier.

Subtracting the per-pixel mean ij (p~(k)ij = p(k)ij - ij):

This step subtracts the mean specific to each pixel position across all images from the respective pixel value. Similar to the first option, it doesn't change the ranking of nearest neighbors based on L1 distance, so this preprocessing step will not change the performance.

Subtracting the mean and dividing by the standard deviation :

This step standardizes the data by centering it around zero (mean subtraction) and scaling it by the standard deviation. It affects the spread of data values. While it maintains the ranking order of nearest neighbors based on L1 distance, it scales the distances by a constant factor. However, this scaling is uniform for all data points and does not affect the relative ordering of nearest neighbors. Therefore, this preprocessing step will not change the performance.

Subtracting the pixel-wise mean ij and dividing by the pixel-wise standard deviation ij:

Similar to the third option, this step standardizes the data but at the pixel level, meaning that each pixel value is centered and scaled independently. It does not change the relative differences between pixel values within an image or the ranking of nearest neighbors based on L1 distance. This step will not change the performance.

Rotating the coordinate axes of the data:

Rotating the coordinate axes fundamentally changes the representation of the data. It alters the relationships between data points and may lead to entirely different nearest neighbors based on L1 distance. Therefore, this preprocessing step will likely change the performance of the Nearest Neighbor classifier.

[11]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words,␣
↪reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
One loop difference was: 0.000000
Good! The distance matrices are the same
``````````````````````````````````````````````
```

[12]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
```
```

```
else:
    print('Uh-oh! The distance matrices are different')
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
No loop difference was: 0.000000
Good! The distance matrices are the same
``````````````````````````````````````````````
```

[13]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took␣
 ↪to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# You should see significantly faster performance with the fully vectorized␣
 ↪implementation!

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
Two loop version took 57.682658 seconds
One loop version took 42.592971 seconds
No loop version took 0.572125 seconds
~~~
```
~~~

1.0.1 Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

[14]:
```
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||||')
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Split the training data into num_folds
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Looping over each possible value of k
for k in k_choices:

    # List to store accuracies for different folds
    accuracies_for_k = []

    # Loop over each fold
    for i in range(num_folds):

        # Creating a validation set from the current fold
        X_validation = X_train_folds[i]
        y_validation = y_train_folds[i]

        # Combining other folds into the training set
        X_train_fold = np.concatenate([fold for j, fold in␣
 ↪enumerate(X_train_folds) if j != i])
        y_train_fold = np.concatenate([fold for j, fold in␣
 ↪enumerate(y_train_folds) if j != i])

        # Create a k-nearest neighbors classifier instance with the current k␣
 ↪value
        classifier = KNearestNeighbor()
        classifier.train(X_train_fold, y_train_fold)

        # Predict labels for the current validation set
        y_predict_val = classifier.predict(X_validation, k=k)

        # Compute the accuracy for the current fold
        correct_pred = np.sum(y_predict_val == y_validation)
        accuracy = float(correct_pred) / len(y_validation)

        # Store the accuracy for the current fold
        accuracies_for_k.append(accuracy)

    # Store the list of accuracies for this k value in the dictionary
    k_to_accuracies[k] = accuracies_for_k

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('`````````````````````````````````````````````````')
```

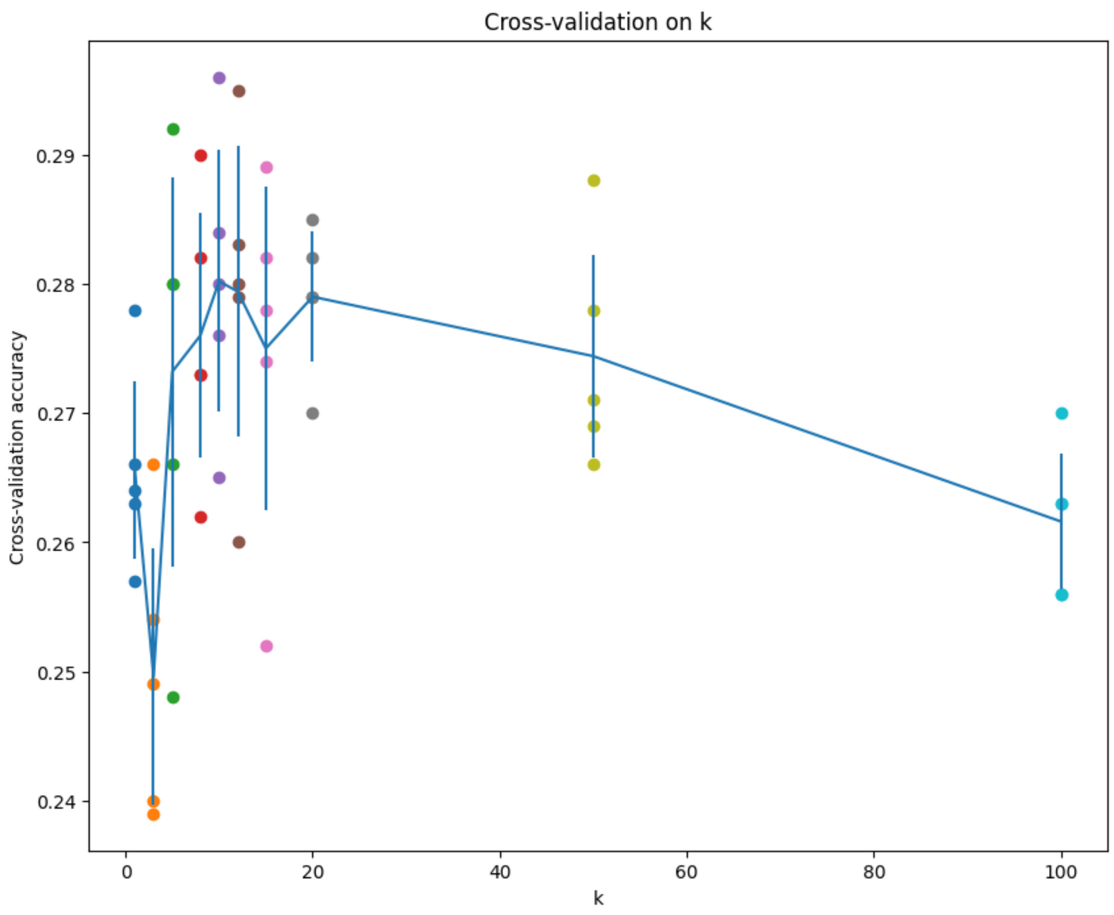
```
|||||||||||||||||||||||||||||||||||||||||||||||
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
```

~~~
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
`````````````````````````````````````````````
~~~

[15]:

```python
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||||')
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
# PLEASE DO NOT MODIFY THE MARKERS
print('``````````````````````````````````````````````')
```

```
||||||||||||||||||||||||||||||||||||||||||||||
```



```
````````````````````````````````````````````
~~~

[16]:
```python
# PLEASE DO NOT MODIFY THE MARKERS
print('||||||||||||||||||||||||||||||||||||||||||||')
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.

# From the above results, we can see that for k=10, we get the maximum accuracy
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
# PLEASE DO NOT MODIFY THE MARKERS
print('````````````````````````````````````````````')
~~~

```
||||||||||||||||||||||||||||||||||||||||||||
Got 141 / 500 correct => accuracy: 0.282000
````````````````````````````````````````````
```

**Inline Question 3**

Which of the following statements about $k$-Nearest Neighbor ($k$-NN) are true in a classification setting, and for all $k$? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

$\color{blue}{Your Answer:}$ The true statements for KNN in classification are:

The training error of a 1-NN will always be lower than or equal to that of 5-NN.

The time needed to classify a test example with the k-NN classifier grows with the size of the training set.

$\color{blue}{Your Explanation:}$

1. The decision boundary of the k-NN classifier is linear.

**False:** The decision boundary of the k-NN classifier is not linear. It's a non-linear boundary that follows the shape of the training data. The classifier assigns a test point to the class that is most common among its k-nearest neighbors, leading to potentially complex and non-linear decision boundaries.
```

2. The training error of a 1-NN will always be lower than or equal to that of 5-NN.

**True:** In a 1-NN classifier, each training point is its own neighbor. This means that the classifier will always predict the correct class for every training point, resulting in a training error of 0 or very close to 0. In contrast, a 5-NN classifier considers the majority class among the 5 nearest neighbors, which may not always be the correct class for every training point. Therefore, the training error of a 1-NN will be lower or equal to that of a 5-NN.

3. The test error of a 1-NN will always be lower than that of a 5-NN.

**False:** The test error of a 1-NN is not guaranteed to be lower than that of a 5-NN. While a 1-NN classifier can fit the training data perfectly (leading to low training error), it can also be highly sensitive to noise and outliers, resulting in poor generalization to unseen data and potentially higher test error. A 5-NN classifier, by considering a larger number of neighbors, may be more robust to noise and provide better generalization in some cases.

4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.

**True:** The time needed to classify a test example with the k-NN classifier does grow with the size of the training set. To make a prediction, the classifier needs to compute distances between the test example and all training examples, and this computation becomes more time-consuming as the training set size increases. The larger the training set, the more distances need to be computed, leading to longer classification times.

# svm

October 8, 2023

```
[1]: # This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'Fall_2023/809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1
```

## 1 Multiclass Support Vector Machine exercise

*Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.*

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

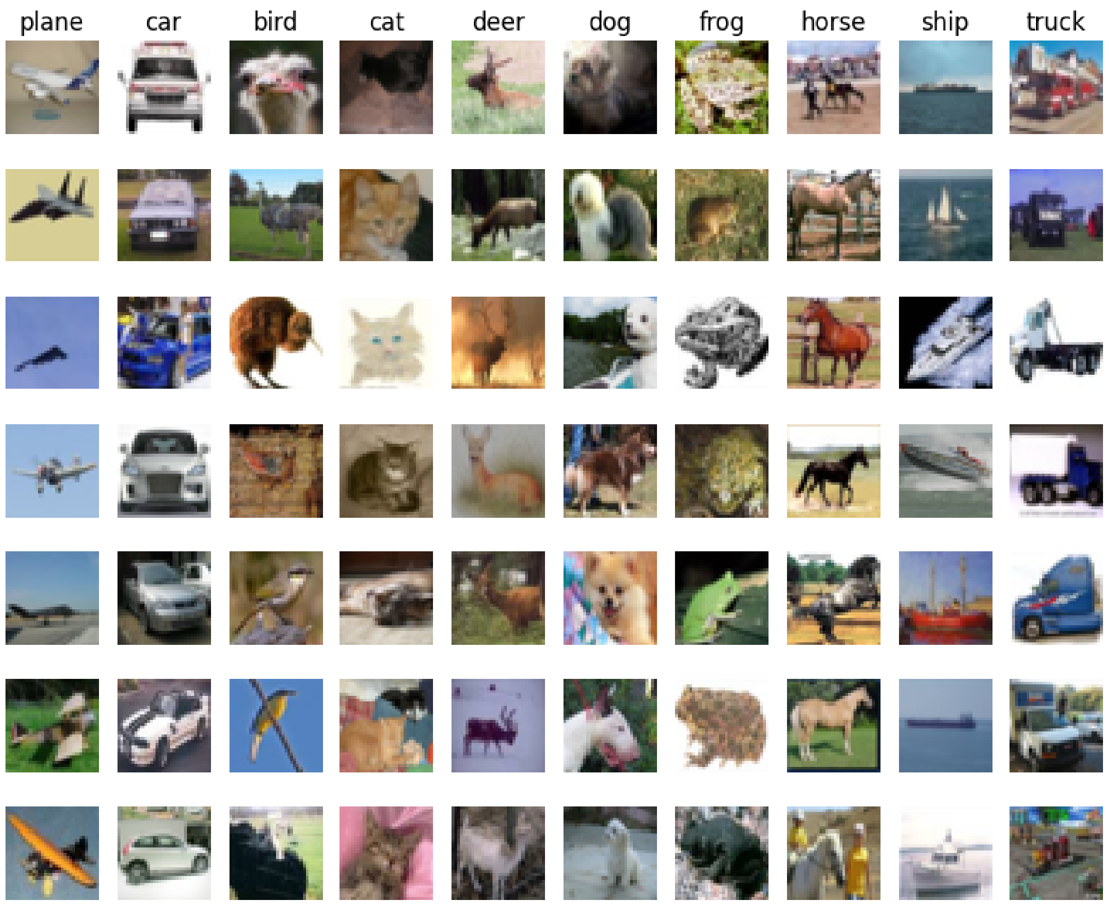
```
[2]: # Run some setup code for this notebook.
     import random
     import numpy as np
     from cs231n.data_utils import load_CIFAR10
     import matplotlib.pyplot as plt

     # This is a bit of magic to make matplotlib figures appear inline in the
     # notebook rather than in a new window.
     %matplotlib inline
     plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
     plt.rcParams['image.interpolation'] = 'nearest'
     plt.rcParams['image.cmap'] = 'gray'

     # Some more magic so that the notebook will reload external python modules;
     # see http://stackoverflow.com/questions/1907993/
      ↪autoreload-of-modules-in-ipython
     %load_ext autoreload
     %autoreload 2
```

## 1.1 CIFAR-10 Data Loading and Preprocessing

```
[3]: # Load the raw CIFAR-10 data.
     cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

     # Cleaning up variables to prevent loading data multiple times (which may cause␣
      ↪memory issue)
     try:
        del X_train, y_train
        del X_test, y_test
        print('Clear previously loaded data.')
     except:
        pass

     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,)
```

[5]:
```python
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'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()
```



[6]:
```python
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

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)
```

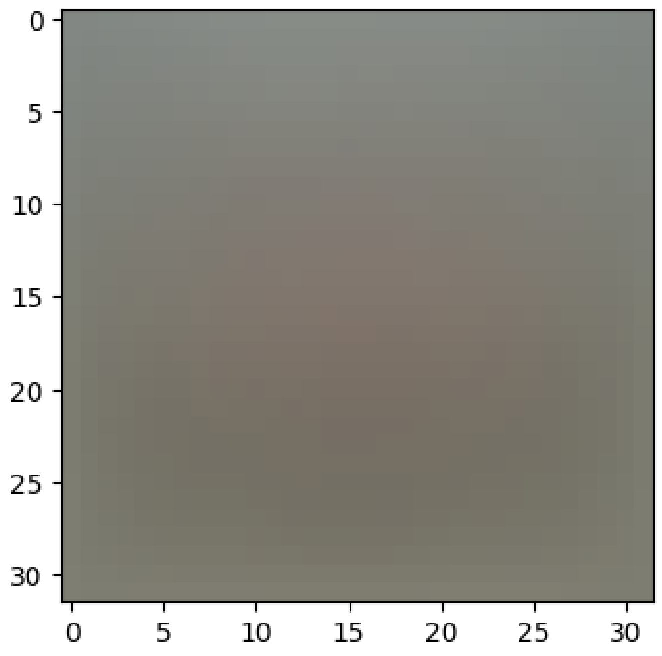
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

```
[7]: # Preprocessing: reshape the image data into rows
     X_train = np.reshape(X_train, (X_train.shape[0], -1))
     X_val = np.reshape(X_val, (X_val.shape[0], -1))
     X_test = np.reshape(X_test, (X_test.shape[0], -1))
     X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

     # As a sanity check, print out the shapes of the data
     print('Training data shape: ', X_train.shape)
     print('Validation data shape: ', X_val.shape)
     print('Test data shape: ', X_test.shape)
     print('dev data shape: ', X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

```
[8]: # Preprocessing: subtract the mean image
     # first: compute the image mean based on the training data
     mean_image = np.mean(X_train, axis=0)
     print(mean_image[:10]) # print a few of the elements
     plt.figure(figsize=(4,4))
     plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
      ↪image
     plt.show()

     # second: subtract the mean image from train and test data
     X_train -= mean_image
     X_val -= mean_image
     X_test -= mean_image
     X_dev -= mean_image

     # third: append the bias dimension of ones (i.e. bias trick) so that our SVM
     # only has to worry about optimizing a single weight matrix W.
     X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
     X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
     X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
     X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

     print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

## 1.2 SVM Classifier

Your code for this section will all be written inside `cs231n/classifiers/linear_svm.py`.

As you can see, we have prefilled the function `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

[9]:
```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
# print('grad:', (grad, ))
```

```
loss: 8.755628
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

[10]:
```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should␣
 ↪match
# almost exactly along all dimensions.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

```
numerical: 13.881934 analytic: 13.881934, relative error: 1.158822e-12
numerical: 18.210502 analytic: 18.210502, relative error: 1.337996e-11
numerical: 50.252043 analytic: 50.252043, relative error: 9.886464e-12
numerical: -47.900083 analytic: -47.900083, relative error: 8.919966e-12
numerical: 5.994252 analytic: 5.994252, relative error: 2.514451e-11
numerical: -7.652078 analytic: -7.652078, relative error: 9.202279e-12
numerical: 1.058104 analytic: 1.058104, relative error: 2.564292e-10
numerical: 15.087501 analytic: 15.087501, relative error: 6.179543e-12
numerical: 15.104610 analytic: 15.104610, relative error: 1.759062e-11
numerical: -18.471427 analytic: -18.471427, relative error: 7.965869e-12
numerical: -6.942231 analytic: -6.942231, relative error: 2.016822e-11
numerical: 8.725024 analytic: 8.725024, relative error: 4.213547e-11
numerical: -17.738145 analytic: -17.738145, relative error: 2.225385e-12
numerical: 6.981637 analytic: 6.981637, relative error: 5.856049e-11
numerical: 9.809651 analytic: 9.809651, relative error: 5.632827e-11
numerical: 35.437390 analytic: 35.437390, relative error: 1.097252e-11
numerical: -12.690991 analytic: -12.690991, relative error: 1.416775e-11
numerical: -5.753252 analytic: -5.753252, relative error: 4.618132e-11
numerical: -8.882835 analytic: -8.882835, relative error: 1.262431e-11
numerical: -4.935892 analytic: -4.935892, relative error: 6.360917e-11
```

**Inline Question 1**

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

*Your Answer :*

Gradient checking discrepancies can occasionally arise due to the non-differentiable nature of the SVM loss function, particularly at points where the max(0, x) function is involved. For instance, when x is exactly zero or very close to it, gradients can be undefined or cause differences between analytical and numerical gradients. The margin (delta) in the SVM loss function can affect the frequency of such occurrences: a larger margin reduces sensitivity to small score changes, while a smaller margin increases sensitivity. While small discrepancies are common due to non-differentiability, significant differences should be examined for potential issues in the code.

[11]:
```
# Next implement the function svm_loss_vectorized; for now only compute the␣
  ↪loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much␣
  ↪faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.755628e+00 computed in 0.104493s
Vectorized loss: 8.755628e+00 computed in 0.016430s
difference: 0.000000
```

[12]:
```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.105259s
Vectorized loss and gradient: computed in 0.011973s
difference: 0.000000
```

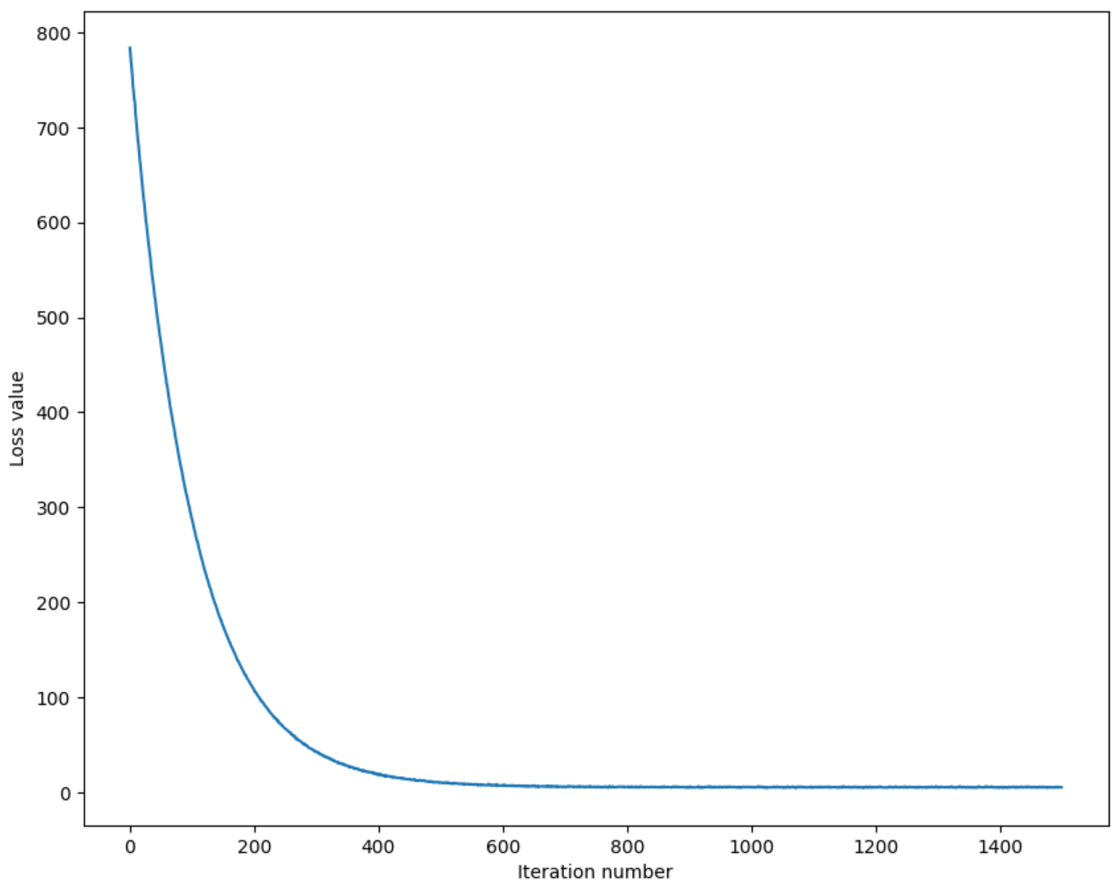
### 1.2.1 Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[13]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 783.510271
iteration 100 / 1500: loss 287.144046
iteration 200 / 1500: loss 107.280377
iteration 300 / 1500: loss 42.407943
iteration 400 / 1500: loss 18.218376
iteration 500 / 1500: loss 10.120924
iteration 600 / 1500: loss 7.102378
iteration 700 / 1500: loss 5.735536
iteration 800 / 1500: loss 5.573736
iteration 900 / 1500: loss 5.196049
iteration 1000 / 1500: loss 5.670228
iteration 1100 / 1500: loss 4.976097
iteration 1200 / 1500: loss 5.295354
iteration 1300 / 1500: loss 4.948975
iteration 1400 / 1500: loss 5.336650
That took 11.034664s
```

[14]:
```
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



[15]:
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.371224
validation accuracy: 0.375000
```

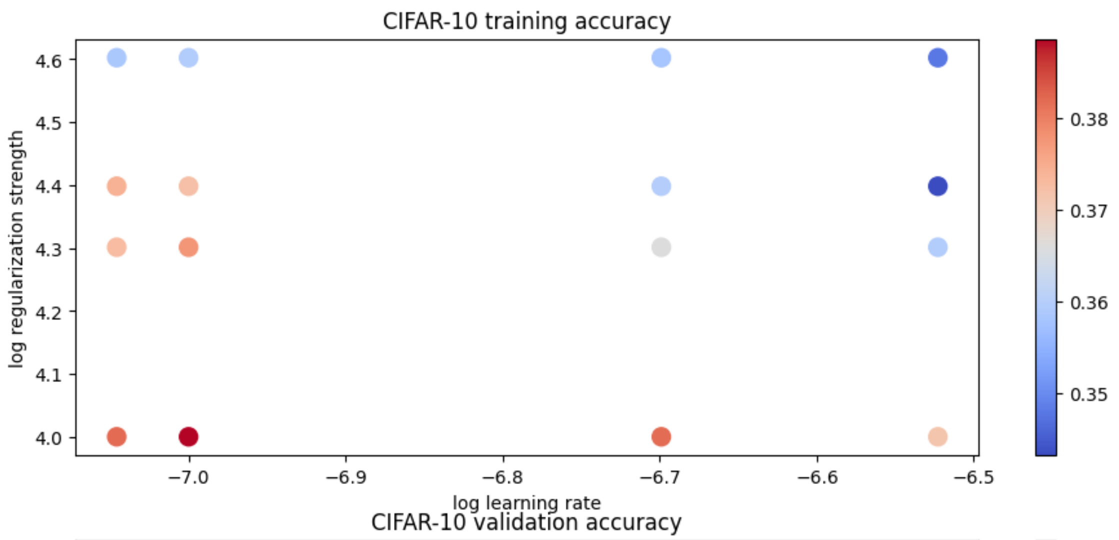
[16]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation␣
 ↪rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 2e-7, 3e-7, 9e-8]
regularization_strengths = [1e4, 2e4, 2.5e4, 4e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# looping through each Learning Rate and regularization_strength for getting␣
 ↪accuracy

combination_num = 0

# looping through each Learning Rate
for lr in learning_rates:

  # looping through each regularization_strength
  for reg_strn in regularization_strengths:

    # Creating an SVM model instance
    svm = LinearSVM()

    # Training the model for evry combination of lr and reg_strn
    loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=reg_strn,
                      num_iters=1500, verbose=False)
    # Making predictions on train data
    y_train_pred = svm.predict(X_train)

    # Calculating training accuracy
    train_accuracy = np.mean(y_train == y_train_pred)

    # making predictions on validation data
    y_val_pred = svm.predict(X_val)

    # Calculating validation accuracy
    val_accuracy = np.mean(y_val == y_val_pred)

    # Storing the (training accuracy, Validation accuracy) pair results in the␣
 ↪results dict
    results[(lr,reg_strn)] = (train_accuracy,val_accuracy)

    # Getting the best validation accuracy and the best model which gives it
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm

    combination_num = combination_num + 1

    print("Combination of learning rate and Regularization Strength number",
          combination_num ,"training completed")

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

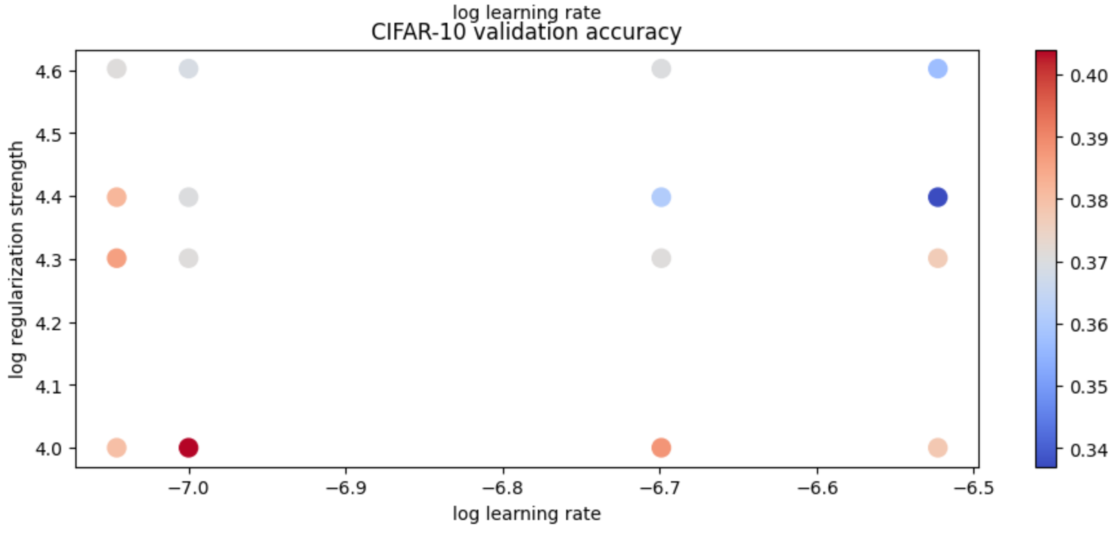
```
Combination of learning rate and Regularization Strength number 1 training
completed
Combination of learning rate and Regularization Strength number 2 training
completed
Combination of learning rate and Regularization Strength number 3 training
completed
Combination of learning rate and Regularization Strength number 4 training
completed
Combination of learning rate and Regularization Strength number 5 training
```

```
completed
Combination of learning rate and Regularization Strength number 6 training
completed
Combination of learning rate and Regularization Strength number 7 training
completed
Combination of learning rate and Regularization Strength number 8 training
completed
Combination of learning rate and Regularization Strength number 9 training
completed
Combination of learning rate and Regularization Strength number 10 training
completed
Combination of learning rate and Regularization Strength number 11 training
completed
Combination of learning rate and Regularization Strength number 12 training
completed
Combination of learning rate and Regularization Strength number 13 training
completed
Combination of learning rate and Regularization Strength number 14 training
completed
Combination of learning rate and Regularization Strength number 15 training
completed
Combination of learning rate and Regularization Strength number 16 training
completed
lr 9.000000e-08 reg 1.000000e+04 train accuracy: 0.382020 val accuracy: 0.380000
lr 9.000000e-08 reg 2.000000e+04 train accuracy: 0.373000 val accuracy: 0.386000
lr 9.000000e-08 reg 2.500000e+04 train accuracy: 0.374388 val accuracy: 0.382000
lr 9.000000e-08 reg 4.000000e+04 train accuracy: 0.358490 val accuracy: 0.371000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.388653 val accuracy: 0.404000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.377653 val accuracy: 0.371000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.372163 val accuracy: 0.370000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.359408 val accuracy: 0.369000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.381735 val accuracy: 0.388000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.365939 val accuracy: 0.371000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.359837 val accuracy: 0.361000
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.357939 val accuracy: 0.370000
lr 3.000000e-07 reg 1.000000e+04 train accuracy: 0.371490 val accuracy: 0.378000
lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.359347 val accuracy: 0.377000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.343143 val accuracy: 0.337000
lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.347776 val accuracy: 0.357000
best validation accuracy achieved during cross-validation: 0.404000
```

[17]:
```python
# Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()
```

```
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



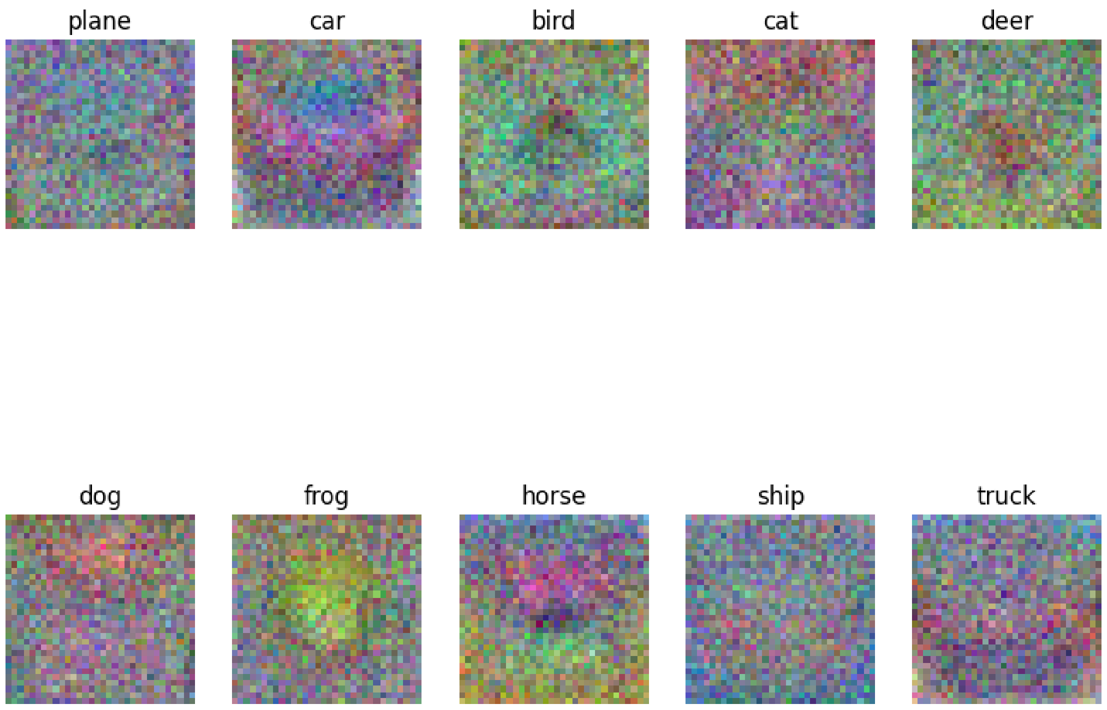


[18]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.381000
```

[19]:
```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these␣
 ↪may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
```

```python
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



**Inline question 2**

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

*Your Answer :*

The visualized SVM weights can be seen as rough sketches that outline the fundamental characteristics of each class, though they may lack the fine details. For instance, you might notice a basic car shape or a recognizable horse outline, other classes are more blurred but you can see that every class has certain strong and dominant underlying common characteristics.

These visualized weights adopt this appearance because they are a result of the training process. During training, the SVM algorithm adjusts these weight vectors to create patterns that align with the prevalent features observed in the training data. The blurriness or simplification in these weights is a result of generalization; the SVM aims to capture the most common and distinguishing characteristics of each class while avoiding overfitting to individual data points.

In essence, these visualized weights serve as templates for classification, based on the dominant features present in the training dataset. While they might not capture every intricate detail, they are effective in making class distinctions by emphasizing the shared traits that define each category.

# softmax

October 8, 2023

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'Fall_2023/809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1
```

## 1 Softmax exercise

*Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.*

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

[2]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

[3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear 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 = 'cs231n/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may␣
 ↪cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    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]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # 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
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =␣
 ↪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)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

### 1.1 Softmax Classifier

Your code for this section will all be written inside `cs231n/classifiers/softmax.py`.

[4]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.329072
sanity check: 2.302585
```

**Inline Question 1**

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

$Your Answer :$

When we start training a neural network with random weights for classifying images in the CIFAR-10 dataset, each of the 10 possible classes has an equal chance of being correct (about 10%). So, when the network makes random guesses before learning anything from the data, the expected loss is like flipping a coin with 10 sides with a probablity of 1/10. We use cross-entropy to measure the loss in probabilities. So in this case the cross entropy will be -log(0.1). Hence we expect loss close to -log(0.1)

[5]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 2.675267 analytic: 2.675267, relative error: 1.401286e-08
```

```
numerical: 1.933694 analytic: 1.933693, relative error: 1.292398e-08
numerical: 0.849572 analytic: 0.849572, relative error: 7.304242e-08
numerical: 2.073966 analytic: 2.073966, relative error: 7.041991e-09
numerical: 0.329857 analytic: 0.329857, relative error: 2.077193e-07
numerical: -0.075723 analytic: -0.075723, relative error: 2.370070e-07
numerical: -0.966070 analytic: -0.966070, relative error: 1.282502e-08
numerical: -0.895435 analytic: -0.895435, relative error: 1.607684e-09
numerical: -0.349359 analytic: -0.349359, relative error: 1.708472e-07
numerical: 2.614848 analytic: 2.614848, relative error: 1.102233e-08
numerical: -3.135158 analytic: -3.135158, relative error: 1.723245e-08
numerical: -0.363181 analytic: -0.363181, relative error: 1.950964e-07
numerical: -0.423502 analytic: -0.423502, relative error: 9.857786e-08
numerical: 1.461496 analytic: 1.461496, relative error: 1.804112e-08
numerical: 1.960713 analytic: 1.960713, relative error: 1.854986e-08
numerical: 2.022672 analytic: 2.022672, relative error: 3.002163e-08
numerical: 0.625334 analytic: 0.625334, relative error: 2.531233e-08
numerical: -0.262926 analytic: -0.262927, relative error: 3.951246e-07
numerical: -1.734791 analytic: -1.734792, relative error: 3.665335e-08
numerical: 0.023049 analytic: 0.023049, relative error: 2.886920e-07
```

[6]:
```
# Now that we have a naive implementation of the softmax loss function and its␣
↪gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version␣
↪should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
↪000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.329072e+00 computed in 0.326811s
vectorized loss: 2.329072e+00 computed in 0.032758s
Loss difference: 0.000000
```

```
Gradient difference: 0.000000
```

[7]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 2e-7, 8e-8]
regularization_strengths = [1e4, 2e4, 3e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

combination_num = 0

# for Learning Rate and Regularization Strength
for lr in learning_rates:

  for reg_strn in regularization_strengths:

    # Creating an Softmax model instance
    softmax_model = Softmax()

    # Training the model for evry combination of lr and reg_strn
    loss_hist = softmax_model.train(X_train, y_train, learning_rate=lr,␣
 ↪reg=reg_strn,
                      num_iters=1500, verbose=False)

    # Making predictions on train data
    y_train_pred = softmax_model.predict(X_train)

    # Calculating training accuracy
    train_accuracy = np.mean(y_train == y_train_pred)
    # making predictions on validation data
    y_val_pred = softmax_model.predict(X_val)

    # Calculating validation accuracy
    val_accuracy = np.mean(y_val == y_val_pred)

    # Storing the (training accuracy, Validation accuracy) pair results in the␣
 ↪results dict
    results[(lr,reg_strn)] = (train_accuracy,val_accuracy)

    # Getting the best validation accuracy and the best model which gives it
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_softmax = softmax_model

    combination_num = combination_num + 1

    print("Combination of learning rate and Regularization Strength number",
          combination_num ,"training completed")


print("---------------------------------------------------------------------------------------'

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
Combination of learning rate and Regularization Strength number 1 training
completed
Combination of learning rate and Regularization Strength number 2 training
completed
Combination of learning rate and Regularization Strength number 3 training
completed
Combination of learning rate and Regularization Strength number 4 training
completed
Combination of learning rate and Regularization Strength number 5 training
completed
Combination of learning rate and Regularization Strength number 6 training
completed
Combination of learning rate and Regularization Strength number 7 training
```

```
completed
Combination of learning rate and Regularization Strength number 8 training
completed
Combination of learning rate and Regularization Strength number 9 training
completed
--------------------------------------------------------------------------------
------
lr 8.000000e-08 reg 1.000000e+04 train accuracy: 0.310449 val accuracy: 0.314000
lr 8.000000e-08 reg 2.000000e+04 train accuracy: 0.342347 val accuracy: 0.353000
lr 8.000000e-08 reg 3.000000e+04 train accuracy: 0.342755 val accuracy: 0.362000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.332551 val accuracy: 0.337000
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.353776 val accuracy: 0.355000
lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.349429 val accuracy: 0.363000
lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.369327 val accuracy: 0.374000
lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.356939 val accuracy: 0.372000
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.349878 val accuracy: 0.357000
best validation accuracy achieved during cross-validation: 0.374000
```

[8]:
```python
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.369000
```

**Inline Question 2** - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$\color{blue}{\textit{Your Answer :}}$

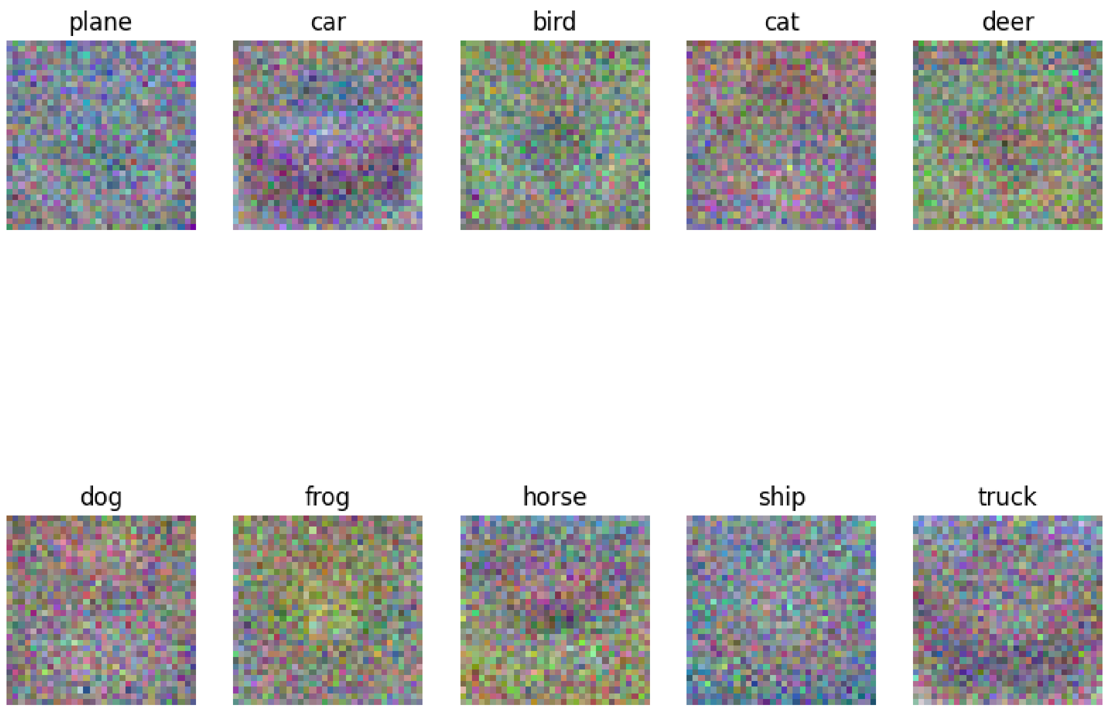
TRUE. It is possible to add a new data point to a training set that would leave the SVM loss unchanged, but this is generally not the case with the Softmax classifier loss.

$\color{blue}{\textit{Your Explanation :}}$

**SVM Loss:** The SVM loss mainly focuses on the gap or margin between different classes. When you add a new data point to your training set, it might not alter this margin if the new point doesn't impact how the existing classes are separated. In such a situation, the SVM loss will stay the same because it's tied to the margin itself and not how many data points there are. So, yes, it's possible to add a new data point that doesn't affect the SVM loss.

**Softmax Loss:** The Softmax classifier loss considers the probabilities assigned to each class for every data point. When you introduce a new data point, it affects these probabilities. Even if the new point closely resembles existing data and doesn't dramatically alter the probabilities, it still exerts some influence. As a result, when you add a new data point, it's probable that the Softmax classifier loss will change, although the change might be minor.

[9]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



[ ]:

# two_layer_net

October 8, 2023

[ ]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'Fall_2023/809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1
```

## 1 Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks using a modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output
  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

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

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

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

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

[ ]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

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

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

```
[ ]: # Load the (preprocessed) CIFAR10 data.

     data = get_CIFAR10_data()
     for k, v in list(data.items()):
       print(('%s: ' % k, v.shape))
```

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

## 2 Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

```
[ ]: # Test the affine_forward function

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

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

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

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

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

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

## 3 Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

```
[ ]: # Test the affine_backward function
     np.random.seed(231)
     x = np.random.randn(10, 2, 3)
     w = np.random.randn(6, 5)
     b = np.random.randn(5)
     dout = np.random.randn(10, 5)

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

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

     # The error should be around e-10 or less
     print('Testing affine_backward function:')
     print('dx error: ', rel_error(dx_num, dx))
     print('dw error: ', rel_error(dw_num, dw))
     print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

## 4 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

```
[ ]: # Test the relu_forward function

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

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

```
# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

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

## 5 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

[ ]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

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

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

# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

### 5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. ReLU 3. Leaky ReLU

### 5.2 Answer:

The issue of getting zero (or close to zero) gradient flow during backpropagation is called Vanishing Gradient Problem. Every activation function handels this problem differently.

**Sigmoid:**

Vanishing Gradient Problem: Sigmoid activation functions tend to have vanishing gradient issues when the input is either very large (positive or negative) or close to zero.

Input Conditions: In the one-dimensional case, sigmoid's gradient becomes close to zero when the input values are significantly positive or significantly negative. It also approaches zero when the input is around zero. One can examine this behaviour by checking the graph of the sigmoid function. It becomes flat for large values leading to vainishing gradient problem.

**ReLU:**

Vanishing Gradient Problem: ReLU activation functions are less prone to vanishing gradient problems compared to sigmoid. However, they can still have issues if the input is consistently negative.

Input Conditions: In the one-dimensional case, ReLU's gradient becomes zero for all negative input values. So ReLU faces this problem for consistent negative input values. The value of ReLU is always zero for negative inputs as one can check from its graph.

**Leaky ReLU:**

Vanishing Gradient Problem: Leaky ReLU activation function was introduced to address the vanishing gradient issue of the regular ReLU. It allows a small gradient for negative inputs.

Input Conditions: Leaky ReLU retains a non-zero gradient for all input values, even for consistently negative inputs. Therefore, it doesn't suffer from the vanishing gradient problem in the same way as the other activation functions mentioned.

In summary, among the activation functions mentioned above:

1. Sigmoid tends to have a vanishing gradient problem, especially for large positive or negative inputs.
2. ReLU can have a vanishing gradient problem for consistently negative inputs
3. Leaky ReLU is designed to mitigate the vanishing gradient issue by allowing a small gradient for negative inputs, and it doesn't have the same vanishing gradient problem as Sigmoid and ReLU.

## 6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

```
[ ]: from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
     np.random.seed(231)
     x = np.random.randn(2, 3, 4)
     w = np.random.randn(12, 10)
     b = np.random.randn(10)
     dout = np.random.randn(2, 10)

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

     dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w,␣
      ↪b)[0], x, dout)
```

```
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w,␣
 ↪b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w,␣
 ↪b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

## 7 Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

You can make sure that the implementations are correct by running the following:

[ ]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

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

# Test svm_loss function. Loss should be around 9 and dx error should be around␣
 ↪the order of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

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

# Test softmax_loss function. Loss should be close to 2.3 and dx error should␣
 ↪be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.0

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

## 8 Two-layer network

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. Read through it to make sure you understand the API. You can run the cell below to test your implementation.

```
[ ]: np.random.seed(231)
     N, D, H, C = 3, 5, 50, 7
     X = np.random.randn(N, D)
     y = np.random.randint(C, size=N)

     std = 1e-3
     model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

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

     print('Testing test-time forward pass ... ')
     model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
     model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
     model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
     model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
     X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
     scores = model.loss(X)
     correct_scores = np.asarray(
       [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
      ↪33206765,  16.09215096],
        [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
      ↪49994135,  16.18839143],
        [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
      ↪66781506,  16.2846319 ]])
     scores_diff = np.abs(scores - correct_scores).sum()
```

```
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

## 9 Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. You also need to imeplement the `sgd` function in `cs231n/optim.py`. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

[ ]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
```

```
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                            #
##############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Instanc of Solver with values of hyperparameters
solver = Solver(model, data, update_rule='sgd', optim_config={'learning_rate':
 ↪1e-4},
                lr_decay=0.97, num_epochs=10, batch_size=100,
                print_every=500)

solver.train()

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

```
(Iteration 1 / 4900) loss: 2.300089
(Epoch 0 / 10) train acc: 0.138000; val_acc: 0.138000
(Epoch 1 / 10) train acc: 0.297000; val_acc: 0.300000
(Iteration 501 / 4900) loss: 1.979034
(Epoch 2 / 10) train acc: 0.357000; val_acc: 0.358000
(Iteration 1001 / 4900) loss: 1.716590
(Epoch 3 / 10) train acc: 0.404000; val_acc: 0.381000
(Iteration 1501 / 4900) loss: 1.715657
(Epoch 4 / 10) train acc: 0.405000; val_acc: 0.418000
(Iteration 2001 / 4900) loss: 1.666511
(Epoch 5 / 10) train acc: 0.405000; val_acc: 0.433000
(Iteration 2501 / 4900) loss: 1.583385
(Epoch 6 / 10) train acc: 0.446000; val_acc: 0.441000
(Iteration 3001 / 4900) loss: 1.590766
(Epoch 7 / 10) train acc: 0.468000; val_acc: 0.447000
(Iteration 3501 / 4900) loss: 1.743550
(Epoch 8 / 10) train acc: 0.444000; val_acc: 0.464000
(Iteration 4001 / 4900) loss: 1.459516
(Epoch 9 / 10) train acc: 0.486000; val_acc: 0.455000
(Iteration 4501 / 4900) loss: 1.607297
(Epoch 10 / 10) train acc: 0.441000; val_acc: 0.466000
```

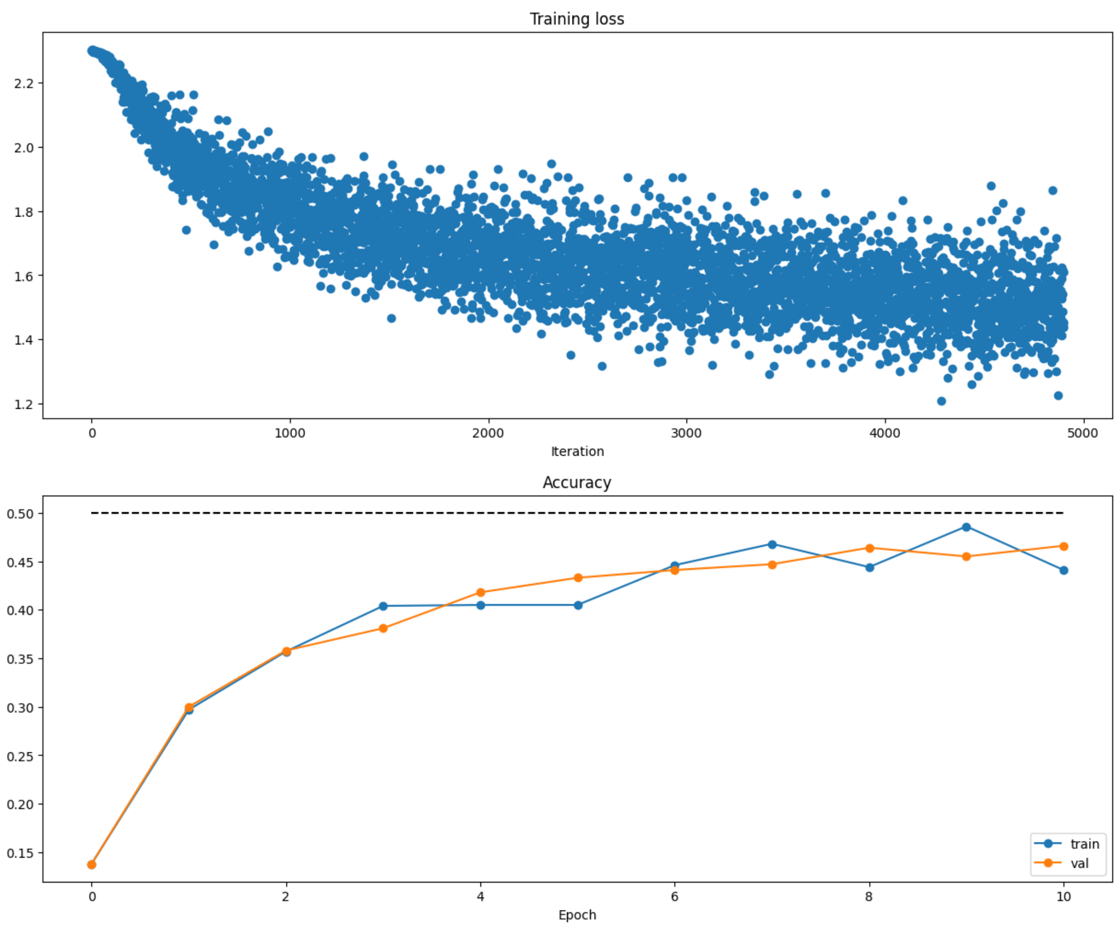
## 10 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

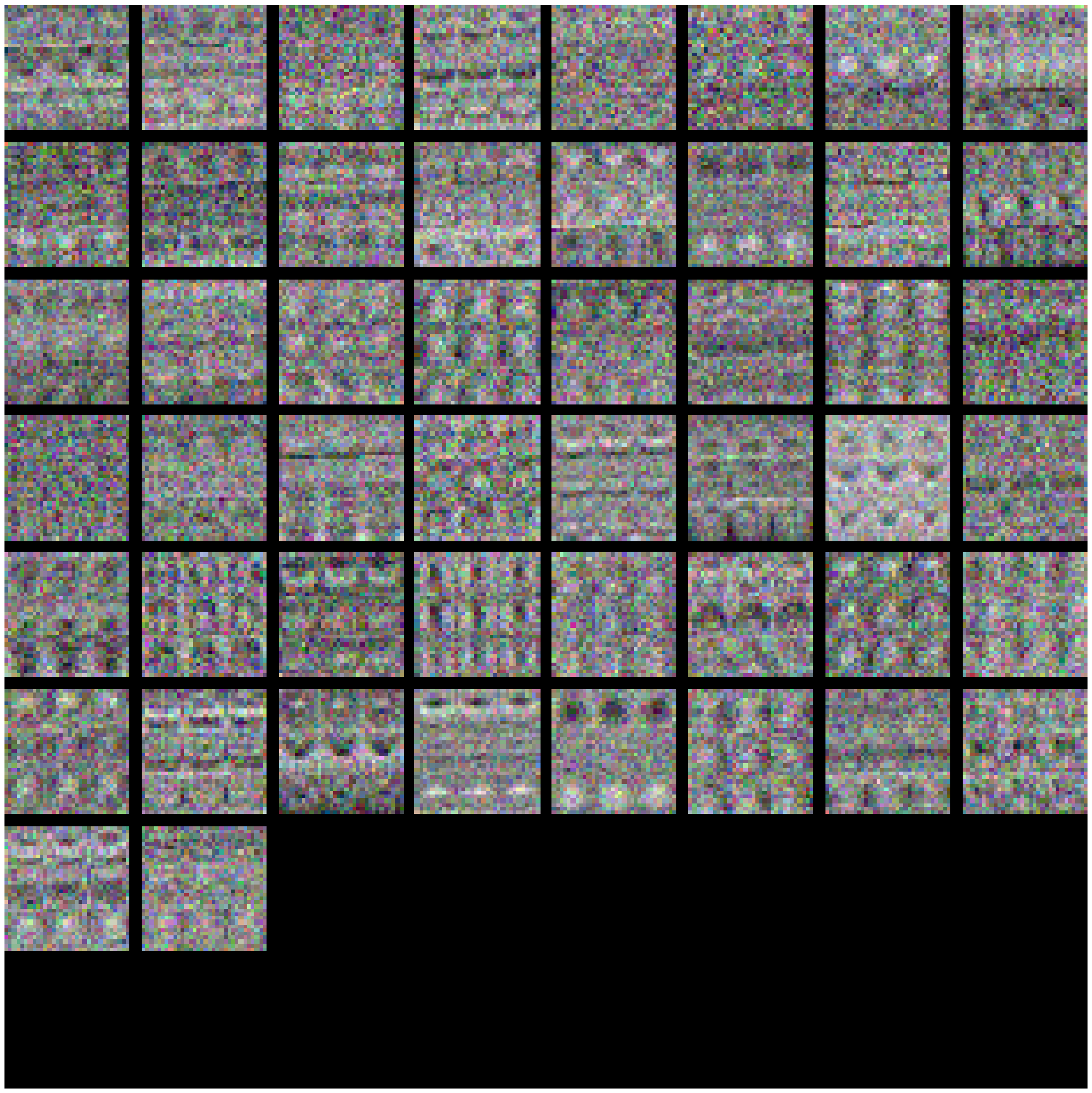
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[ ]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



[ ]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.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(model)
```



## 11 Tune your hyperparameters

**What's wrong?**. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

**Tuning**. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

**Approximate results**. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

**Experiment**: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

[ ]:
```
best_model = None

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
 ↪#
# model in best_model.                                                         ␣
 ↪#
#                                                                              ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative   ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
 ↪#
#                                                                              ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters         ␣
 ↪#
# automatically like we did on thexs previous exercises.                       ␣
 ↪  #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Hyperparameter search space
best_val_accuracy = -1
best_solver = None
results = {}
combination_num = 0

best_net = None
best_val = -1
best_model = None
results = {}
```

```
# Define different values of hyperparameter to validate with
learning_rates = [2e-3, 1e-4, 4e-4]
regularization_strengths = [2e-5, 5.5e-5]
epochs = [10, 15]
hidden_layer_sizes = [90, 120]

# Loop through hyperparameter combinations
for lr in learning_rates:
    for reg_strength in regularization_strengths:
        for hidden_size in hidden_layer_sizes:
            for num_epochs in epochs:

                # Create the model instance
                model = TwoLayerNet(input_size, hidden_dim=hidden_size,␣
 ↪num_classes=10)
                solver = Solver(model, data, update_rule='sgd', optim_config={
                                'learning_rate': lr}, lr_decay=0.95,␣
 ↪num_epochs=num_epochs,
                                batch_size=100, print_every=-1, verbose=False)

                # Train the model
                solver.train()

                # Find the best validation accuracy
                val_accuracy = solver.best_val_acc
                if best_val_accuracy < val_accuracy:
                    best_val_accuracy = val_accuracy

                    # Finding the best Model
                    best_solver = solver
                    best_model = model

                combination_num = combination_num + 1

                print("Combination of hyper parameters",
                      combination_num ,"training completed")

                print('lr %e reg %e hidden %d val accuracy: %f' % (lr,␣
 ↪reg_strength,
                                                       hidden_size,␣
 ↪val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Combination of hyper parameters 1 training completed
lr 2.000000e-03 reg 2.000000e-05 hidden 90 val accuracy: 0.482000
Combination of hyper parameters 2 training completed
lr 2.000000e-03 reg 2.000000e-05 hidden 90 val accuracy: 0.518000
Combination of hyper parameters 3 training completed
lr 2.000000e-03 reg 2.000000e-05 hidden 120 val accuracy: 0.493000
Combination of hyper parameters 4 training completed
lr 2.000000e-03 reg 2.000000e-05 hidden 120 val accuracy: 0.513000
Combination of hyper parameters 5 training completed
lr 2.000000e-03 reg 5.500000e-05 hidden 90 val accuracy: 0.501000
Combination of hyper parameters 6 training completed
lr 2.000000e-03 reg 5.500000e-05 hidden 90 val accuracy: 0.508000
Combination of hyper parameters 7 training completed
lr 2.000000e-03 reg 5.500000e-05 hidden 120 val accuracy: 0.491000
Combination of hyper parameters 8 training completed
lr 2.000000e-03 reg 5.500000e-05 hidden 120 val accuracy: 0.504000
Combination of hyper parameters 9 training completed
lr 1.000000e-04 reg 2.000000e-05 hidden 90 val accuracy: 0.458000
Combination of hyper parameters 10 training completed
lr 1.000000e-04 reg 2.000000e-05 hidden 90 val accuracy: 0.478000
Combination of hyper parameters 11 training completed
lr 1.000000e-04 reg 2.000000e-05 hidden 120 val accuracy: 0.473000
Combination of hyper parameters 12 training completed
lr 1.000000e-04 reg 2.000000e-05 hidden 120 val accuracy: 0.486000
Combination of hyper parameters 13 training completed
lr 1.000000e-04 reg 5.500000e-05 hidden 90 val accuracy: 0.471000
Combination of hyper parameters 14 training completed
lr 1.000000e-04 reg 5.500000e-05 hidden 90 val accuracy: 0.494000
Combination of hyper parameters 15 training completed
lr 1.000000e-04 reg 5.500000e-05 hidden 120 val accuracy: 0.472000
Combination of hyper parameters 16 training completed
lr 1.000000e-04 reg 5.500000e-05 hidden 120 val accuracy: 0.473000
Combination of hyper parameters 17 training completed
lr 4.000000e-04 reg 2.000000e-05 hidden 90 val accuracy: 0.521000
Combination of hyper parameters 18 training completed
lr 4.000000e-04 reg 2.000000e-05 hidden 90 val accuracy: 0.537000
Combination of hyper parameters 19 training completed
lr 4.000000e-04 reg 2.000000e-05 hidden 120 val accuracy: 0.524000
Combination of hyper parameters 20 training completed
lr 4.000000e-04 reg 2.000000e-05 hidden 120 val accuracy: 0.530000
Combination of hyper parameters 21 training completed
lr 4.000000e-04 reg 5.500000e-05 hidden 90 val accuracy: 0.535000
Combination of hyper parameters 22 training completed
```

```
lr 4.000000e-04 reg 5.500000e-05 hidden 90 val accuracy: 0.527000
Combination of hyper parameters 23 training completed
lr 4.000000e-04 reg 5.500000e-05 hidden 120 val accuracy: 0.524000
Combination of hyper parameters 24 training completed
lr 4.000000e-04 reg 5.500000e-05 hidden 120 val accuracy: 0.526000
best validation accuracy achieved during cross-validation: -1.000000
```

## 12 Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

[ ]:
```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.537
```

[ ]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.505
```

### 12.1 Inline Question 2:

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

$\textit{Your Answer :}$

Train on a larger dataset.

Increase the regularization strength.

$\textit{Your Explanation :}$

The gap between training accuracy and testing accuracy, means there is overfitting of the model.

Explanation for each of the above option:

**Train on a larger dataset:** Training on larger data can decrease the gap. With more data the model learns diverse features and can generalize well on unseen data. A larger dataset provides better generalization.

**Add more hidden units:** More hidden units increases the feature extraction capacity of the model. But increasing hidden units without regularization will lead to overfitiing, which again increases the gap. Hence adding hidden units can reduce or increase the gap depending upon fitting of the model.

**Increase the regularization strength:** Regularization is performed to reduce overfitting of the model i.e to reduce the gap. Increasing regularization strength will result in stable weights with smaller magnitude. Depending upon on the type of regularization, some weights might becoem zero as well resulting in selection of only important weights. This reduces overfitting which means the gap decreases.

So, the correct options are:

Train on a larger dataset.

Increase the regularization strength.

The choice of adding more hidden units depends on the specific scenario and should be done cautiously, often in conjunction with regularization to prevent overfitting.

[ ]:

```

```

# features

October 8, 2023

[2]:
```python
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'Fall_2023/809K/Assignments/assignment1/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1/cs231n/datasets
/content/drive/My Drive/Fall_2023/809K/Assignments/assignment1
```

## 1 Image features exercise

*Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.*

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

[3]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


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

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

## 1.1 Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

[4]:
```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may␣
 ↪cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    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]

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

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

## 1.2 Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your own interest.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

[7]:
```
from cs231n.features import *

num_color_bins = 40 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

### 1.3 Train SVM on features

Using the multiclass SVM code developed earlier in the assignment, train SVMs on top of the features extracted above; this should achieve better results than training SVMs directly on top of raw pixels.

[8]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [5e-7, 1e-5, 5e-6]
regularization_strengths = [1e2, 5e2, 5e3, 1e3, 1e4]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

combination_num = 0

# looping through each Learning Rate
for lr in learning_rates:

  # looping through each regularization_strength
  for reg_strn in regularization_strengths:

    # Creating an SVM model instance
    svm = LinearSVM()

    # Training the model for evry combination of lr and reg_strn
    loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr,␣
 ↪reg=reg_strn,
                        num_iters=2000, verbose=False)

    # Making predictions on train data
    y_train_pred = svm.predict(X_train_feats)
    # Calculating training accuracy
    train_accuracy = np.mean(y_train == y_train_pred)

    # making predictions on validation data
    y_val_pred = svm.predict(X_val_feats)

    # Calculating validation accuracy
    val_accuracy = np.mean(y_val == y_val_pred)

    # Storing the (training accuracy, Validation accuracy) pair results in the␣
 ↪results dict
    results[(lr,reg_strn)] = (train_accuracy,val_accuracy)

    # Getting the best validation accuracy and the best model which gives it
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm

    combination_num = combination_num + 1

    print("Combination of learning rate and Regularization Strength number",
          combination_num ,"training completed")

print("\n")

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved: %f' % best_val)
```

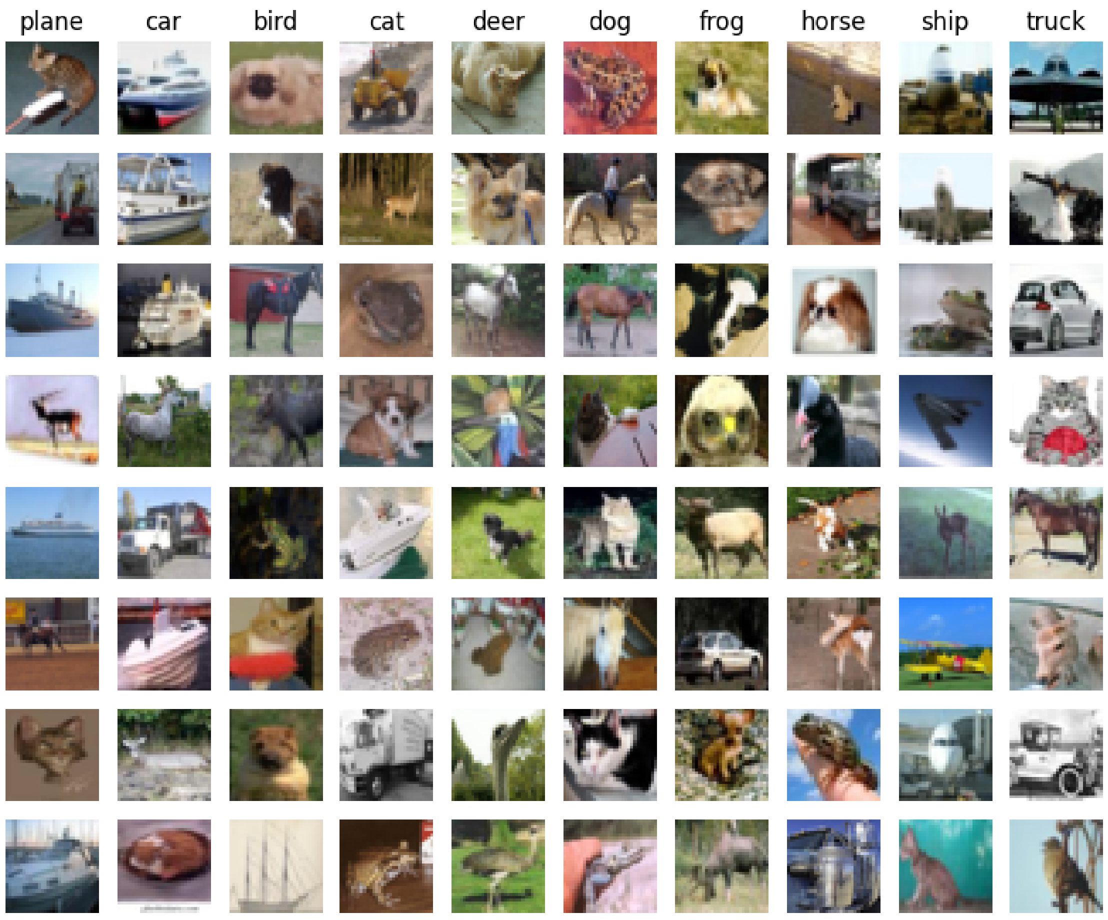
```
Combination of learning rate and Regularization Strength number 1 training
completed
Combination of learning rate and Regularization Strength number 2 training
completed
Combination of learning rate and Regularization Strength number 3 training
completed
Combination of learning rate and Regularization Strength number 4 training
completed
Combination of learning rate and Regularization Strength number 5 training
completed
Combination of learning rate and Regularization Strength number 6 training
completed
```

```
Combination of learning rate and Regularization Strength number 7 training
completed
Combination of learning rate and Regularization Strength number 8 training
completed
Combination of learning rate and Regularization Strength number 9 training
completed
Combination of learning rate and Regularization Strength number 10 training
completed
Combination of learning rate and Regularization Strength number 11 training
completed
Combination of learning rate and Regularization Strength number 12 training
completed
Combination of learning rate and Regularization Strength number 13 training
completed
Combination of learning rate and Regularization Strength number 14 training
completed
Combination of learning rate and Regularization Strength number 15 training
completed


lr 5.000000e-07 reg 1.000000e+02 train accuracy: 0.227449 val accuracy: 0.225000
lr 5.000000e-07 reg 5.000000e+02 train accuracy: 0.275143 val accuracy: 0.262000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.332347 val accuracy: 0.331000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.422980 val accuracy: 0.419000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.424755 val accuracy: 0.424000
lr 5.000000e-06 reg 1.000000e+02 train accuracy: 0.424714 val accuracy: 0.425000
lr 5.000000e-06 reg 5.000000e+02 train accuracy: 0.424061 val accuracy: 0.427000
lr 5.000000e-06 reg 1.000000e+03 train accuracy: 0.423041 val accuracy: 0.429000
lr 5.000000e-06 reg 5.000000e+03 train accuracy: 0.415367 val accuracy: 0.412000
lr 5.000000e-06 reg 1.000000e+04 train accuracy: 0.417531 val accuracy: 0.431000
lr 1.000000e-05 reg 1.000000e+02 train accuracy: 0.424020 val accuracy: 0.429000
lr 1.000000e-05 reg 5.000000e+02 train accuracy: 0.421755 val accuracy: 0.426000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.423184 val accuracy: 0.423000
lr 1.000000e-05 reg 5.000000e+03 train accuracy: 0.421755 val accuracy: 0.444000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.400122 val accuracy: 0.409000
best validation accuracy achieved: 0.444000
```

[9]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.423
```

[10]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls +␣
 ↪1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```



### 1.3.1 Inline question 1:

Describe the misclassification results that you see. Do they make sense?

$\textit{Your Answer :}$

There are many missclassifications error seen above. And these are not random mistakes, this error is due to features we are using to classify the images. We are using HOG and color histogram features to classify the images. These features give a genral idea of shape and color of the objects present in the image and not the exact details to make correct predictions.

**HOG features:** These are Primarily designed to capture the shape and edge information in images, particularly useful for detecting object contours and textures. Misclassifications can occur when objects or animals share similar edge patterns or textures. For example, the HOG features of a bird flying in the sky might resemble those of a small plane, leading to misclassification as you can see several birds are misclassified as plane. Similarly you can see that several trucks are misclassifies as cars due to similar shape.

**Color Histograms:**

These histograms capture the distribution of colors in an image, which is valuable for distinguishing objects based on their color. Misclassifications can happen when different objects have similar color distributions. For example, white plane surrounded by blue sky is miscalssified for ship surrounded by blue water. Similarly white cats having roughly similar shape are classified as dogs due to color similarity.

So the misclassification is because the features we are using for classifications are too general to make an accrate prediction.

## 1.4 Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

[11]:
```python
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 185)
(49000, 184)
```

[12]:
```
from cs231n.data_utils import get_CIFAR10_data
data = get_CIFAR10_data()

# Data
data['X_train'] = X_train_feats
data['y_train'] = y_train
data['X_val'] = X_val_feats
data['y_val'] = y_val
data['X_test'] = X_test_feats
data['y_test'] = y_test
print("\n")
for p, q in list(data.items()):
  print(('%s: ' % p, q.shape))
```

```
('X_train: ', (49000, 184))
('y_train: ', (49000,))
('X_val: ', (1000, 184))
('y_val: ', (1000,))
('X_test: ', (1000, 184))
('y_test: ', (1000,))
```

[13]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Hyperparameter search space
best_val_accuracy = -1
best_solver = None

# Define different values of hyperparameter to validate with
learning_rates = [2e-1, 5e-2, 5e-3]
regularization_strengths = [2e-3, 9e-6, 5.5e-5]
```

```
epochs = [10, 15]
hidden_layer_sizes = [90, 120]

combination_num = 0

# Loop through hyperparameter combinations
for lr in learning_rates:
    for reg_strength in regularization_strengths:
        for hidden_size in hidden_layer_sizes:
            for num_epochs in epochs:

                # Create the model instance
                NN_model = TwoLayerNet(input_dim, hidden_dim=hidden_size,␣
 ↪num_classes=10)

                # Set hyperparameter values
                solver = Solver(NN_model, data, update_rule='sgd',␣
 ↪optim_config={
                                'learning_rate': lr}, lr_decay=0.95,␣
 ↪num_epochs=num_epochs,
                                batch_size=100, print_every=-1, verbose=False)

                # Train the model
                solver.train()

                # Find the best validation accuracy
                val_accuracy = solver.best_val_acc
                if best_val_accuracy < val_accuracy:
                    best_val_accuracy = val_accuracy

                    # Finding the best Model
                    best_solver = solver
                    best_net = NN_model

                combination_num = combination_num + 1

                print("Combination of learning rate and Regularization Strength␣
 ↪number", combination_num ,"training completed")


# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved: %f' % best_val)
```

```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
Combination of learning rate and Regularization Strength number 1 training
completed
Combination of learning rate and Regularization Strength number 2 training
completed
Combination of learning rate and Regularization Strength number 3 training
completed
Combination of learning rate and Regularization Strength number 4 training
completed
Combination of learning rate and Regularization Strength number 5 training
completed
Combination of learning rate and Regularization Strength number 6 training
completed
Combination of learning rate and Regularization Strength number 7 training
completed
Combination of learning rate and Regularization Strength number 8 training
completed
Combination of learning rate and Regularization Strength number 9 training
completed
Combination of learning rate and Regularization Strength number 10 training
completed
Combination of learning rate and Regularization Strength number 11 training
completed
Combination of learning rate and Regularization Strength number 12 training
completed
Combination of learning rate and Regularization Strength number 13 training
completed
Combination of learning rate and Regularization Strength number 14 training
completed
Combination of learning rate and Regularization Strength number 15 training
completed
Combination of learning rate and Regularization Strength number 16 training
completed
Combination of learning rate and Regularization Strength number 17 training
completed
Combination of learning rate and Regularization Strength number 18 training
completed
Combination of learning rate and Regularization Strength number 19 training
completed
Combination of learning rate and Regularization Strength number 20 training
completed
Combination of learning rate and Regularization Strength number 21 training
completed
Combination of learning rate and Regularization Strength number 22 training
```

```
completed
Combination of learning rate and Regularization Strength number 23 training
completed
Combination of learning rate and Regularization Strength number 24 training
completed
Combination of learning rate and Regularization Strength number 25 training
completed
Combination of learning rate and Regularization Strength number 26 training
completed
Combination of learning rate and Regularization Strength number 27 training
completed
Combination of learning rate and Regularization Strength number 28 training
completed
Combination of learning rate and Regularization Strength number 29 training
completed
Combination of learning rate and Regularization Strength number 30 training
completed
Combination of learning rate and Regularization Strength number 31 training
completed
Combination of learning rate and Regularization Strength number 32 training
completed
Combination of learning rate and Regularization Strength number 33 training
completed
Combination of learning rate and Regularization Strength number 34 training
completed
Combination of learning rate and Regularization Strength number 35 training
completed
Combination of learning rate and Regularization Strength number 36 training
completed
lr 5.000000e-07 reg 1.000000e+02 train accuracy: 0.227449 val accuracy: 0.225000
lr 5.000000e-07 reg 5.000000e+02 train accuracy: 0.275143 val accuracy: 0.262000
lr 5.000000e-07 reg 1.000000e+03 train accuracy: 0.332347 val accuracy: 0.331000
lr 5.000000e-07 reg 5.000000e+03 train accuracy: 0.422980 val accuracy: 0.419000
lr 5.000000e-07 reg 1.000000e+04 train accuracy: 0.424755 val accuracy: 0.424000
lr 5.000000e-06 reg 1.000000e+02 train accuracy: 0.424714 val accuracy: 0.425000
lr 5.000000e-06 reg 5.000000e+02 train accuracy: 0.424061 val accuracy: 0.427000
lr 5.000000e-06 reg 1.000000e+03 train accuracy: 0.423041 val accuracy: 0.429000
lr 5.000000e-06 reg 5.000000e+03 train accuracy: 0.415367 val accuracy: 0.412000
lr 5.000000e-06 reg 1.000000e+04 train accuracy: 0.417531 val accuracy: 0.431000
lr 1.000000e-05 reg 1.000000e+02 train accuracy: 0.424020 val accuracy: 0.429000
lr 1.000000e-05 reg 5.000000e+02 train accuracy: 0.421755 val accuracy: 0.426000
lr 1.000000e-05 reg 1.000000e+03 train accuracy: 0.423184 val accuracy: 0.423000
lr 1.000000e-05 reg 5.000000e+03 train accuracy: 0.421755 val accuracy: 0.444000
lr 1.000000e-05 reg 1.000000e+04 train accuracy: 0.400122 val accuracy: 0.409000
best validation accuracy achieved: 0.444000
```

[14]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

y_test_pred = np.argmax(best_net.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.575
```

[ ]:
```

```