knn

October 20, 2020

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.

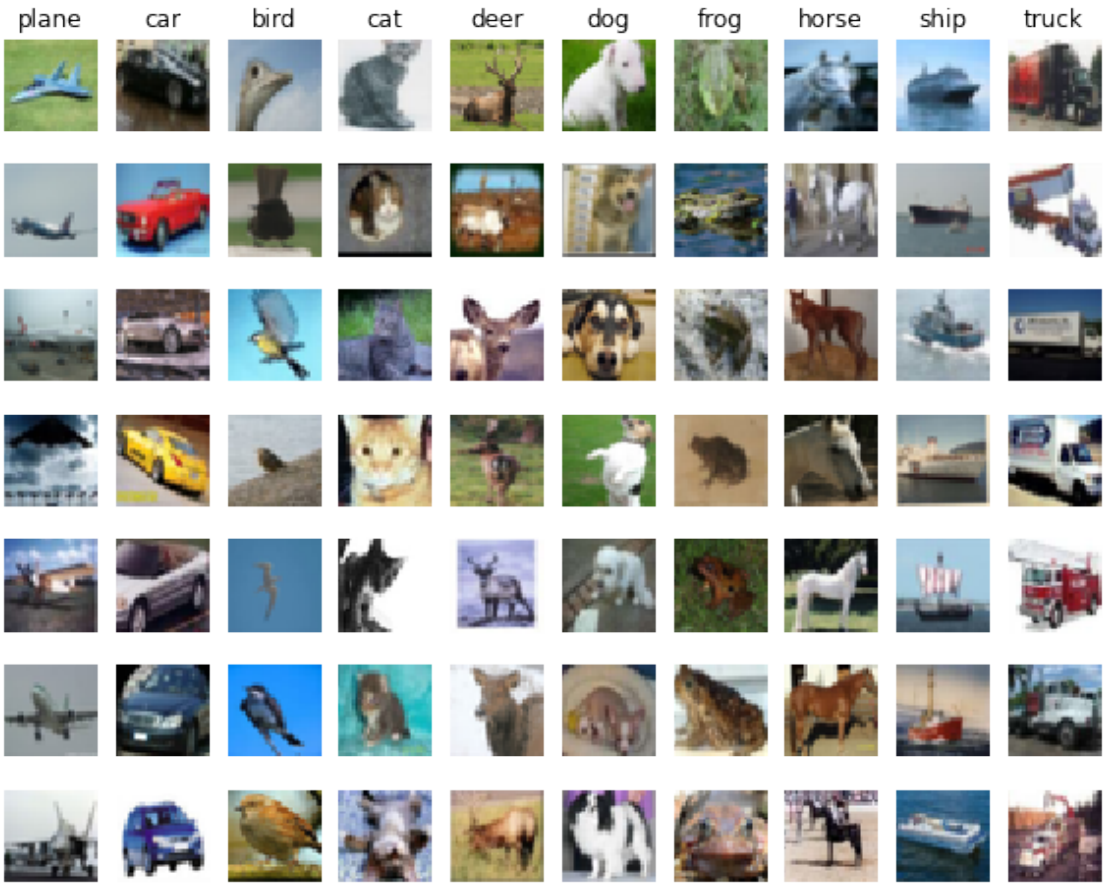
[54]:
```
# Run some setup code for this notebook.

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
```

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

[55]:
```
# 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)
```

```
Clear previously loaded data.
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[56]:
```
# 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()
```



[57]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

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

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

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

[58]:
```
from cs231n.classifiers import KNearestNeighbor

# 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)
```

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.

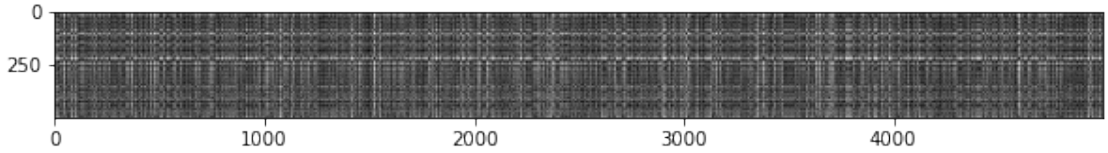
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[59]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

[60]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



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.)

- 1.1 What in the data is the cause behind the distinctly bright rows?
- 1.2 What causes the columns?
- 1.3 What would the rank of the matrix *dists* be if there were no intra-class variation and inter-class separation was the same between all pairs of classes?

YourAnswer : Las distancias se computan pixel a pixel. Debido a esto si la i-ésima imagen de test tiene valores de pixeles distintos a la j-ésima de train, esto se denotará con un color mas claro en la matriz de distancias. Por ejemplo, aunque ambas imágenes sean de un avión, si una es sacada con un cielo nocturno y otra durante el día, la diferencia pixel a pixel de los fondos de las imágenes contribuirán a una gran distancia, es decir, color más claro en la entrada ij de la matriz. Con lo anterior se deduce que las filas más claras indican que la imagen i-ésima de test está a mayor distancia de las imágenes de test. Y que las columnas más claras indican que la imágenes j-ésima de train esta a mayor distancia de el conjunto de test.

[61]:
```
# 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
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
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`:

[62]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
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))
```

```
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 image by 90 degrees.

YourAnswer :

Los que no afectan son 1, 2, 3 y 5

YourExplanation :

1: Restarle la media a las imagenes solo provoca un offset que se anula cuando se toma la diferencia $img_{te}(i) - img_{tr}(j)$, por lo cual el resultado es el mismo.

2: Se llega al resultado con el mismo razonamiento anterior, la media por pixel se anula al tomar la diferencia.

3: Esto lleva los datos a una distribucion normal. Si la norma L1 es:

$L_1(p, q) = ||p - q||_1 = \sum(|p_i - q_i|)$

Entonces, al normalizar:

$L_1((p-\mu)/\sigma, (q-\mu)/\sigma) = ||p - q||_1 = \sum(|(p_ij - \mu)/\sigma - (q_i - \mu)/\sigma|) = \frac{1}{|\sigma|}\sum(|p_i - q_i|)$

La desviacion estandar es un factor de escala uniforme para toda la matriz, no afectando la decisión.

4: El factor de escala depende de la entrada ij por lo que al no ser uniforme para toda la matriz afecta el resultado.

5: Al rotar ambas imagenes por 90 grados, la diferencia pixel a pixel será la misma.

[63]:
```
# 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')
```

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

[64]:
```
# 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')
```

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

[65]:
```
# 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.
```

```
Two loop version took 981.502700 seconds
One loop version took 46.280092 seconds
No loop version took 0.157408 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.

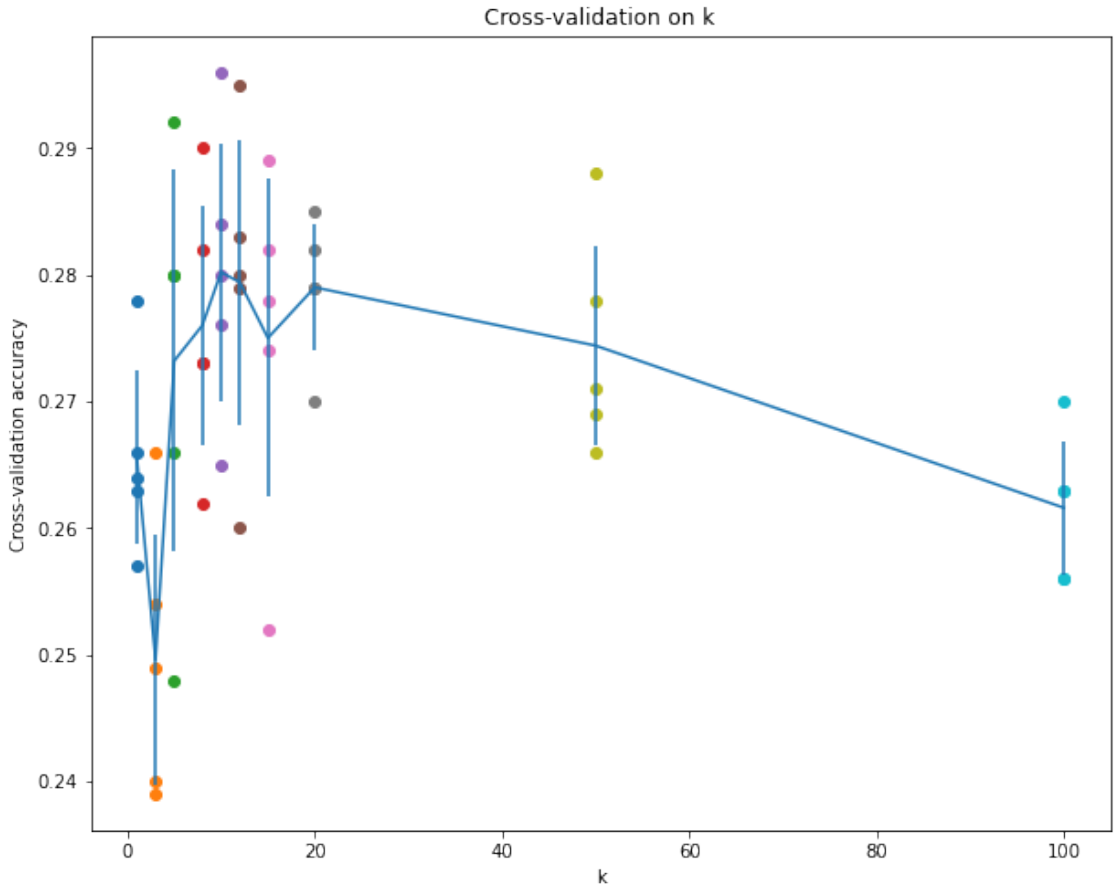
[69]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)*****

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)*****

for k in k_choices:

    k_to_accuracies[k] = []

    for val_fold in range(num_folds):
        # creo un clasificador
        classifier = KNearestNeighbor()

        # entreno utilizando todos menos el fold de validacion
        classifier.train(np.concatenate(X_train_folds[:val_fold] +␣
 ↪X_train_folds[val_fold + 1:]), \
                         np.concatenate(y_train_folds[:val_fold] +␣
 ↪y_train_folds[val_fold + 1:]))

        y_pred = classifier.predict(X_train_folds[val_fold], k=k)

        k_to_accuracies[k].append(np.mean(y_pred == y_train_folds[val_fold]))

# *****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))
```

```
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
```

[70]:
```
# 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()
```



[73]:
```
# 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.
best_k = k_choices[accuracies_mean.argmax()]

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))
```

```
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 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.

YourAnswer : Las verdaderas son la 2 y la 4

YourExplanation :

2: Debido a que para una 1-NN su vecino es siempre el mismo, el error es cero. Esto no sucede con una 5-NN

4: Esto es cierto ya que a mayor cantidad de puntos de train es mayor la cantidad de comparaciones que habrá que hacer para clasificar un punto de test. La matriz de distancias crece con la cantitdad de puntos de train.

Inline Question 4

- Is it possible to use the kNN algorithm in a **regression** problem? If so, please give an example.

YourAnswer :

Extraído del libro "Learning from data" de Yaser S. Abu-Mostafa: - "When the output is a real number, the natural way to combine the outputs of the nearest neighbors is using some form of averaging. The simplest way to extend the k-nearest neighbor algorithm to regression is to take the average of the targe values from the k-nearest neighbors"

Con esto nos da la idea de que en vez de tomar la etiqueta mas votada, se tome la media de los valores de los k vecinos mas cercanos. Se puede sacar una probabilidad tomando como clasificador (nuestra hipótesis final) la fracción de puntos. Por ejemplo para un problema de 2 clases, g es la fracción de los k vecinos mas cercanos clasificados como +1:

$g(x) = \frac{1}{N} \sum_{i=1}^{k} [y_i = +1]$ donde $[y_i = +1]$ es 1 si $y_i = +1$ y si no es 0.

softmax

October 20, 2020

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.

In this 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

[1]:
```
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 external modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[2]:
```
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.
    """
    # 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`.

[23]:
```
# 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: 2.336528
sanity check: 2.302585
```

Inline Question 1

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

- 1.2 What are the minimum and maximum possible value for the Softmax loss?

YourAnswer :

1: La expresión para la discrepancia es $-log(s)$ con s=probabilidad de las clases. Como hay 10 clases en nuestro dataset, la probabilidad es 1/10=0.1.

2: Como la discrepancia es $L = -\sum_{i=1}^{n} log(s) + \lambda W^T W$, para $\lambda = 0$ y como "s" es una probabilidad el minimo es cero si s=1 y $\lambda = 0$. No esta acotada por arriba.

[7]:
```
# 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)

# 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)

# 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: 1.679611 analytic: 1.679611, relative error: 2.402868e-09
numerical: 1.933362 analytic: 1.933362, relative error: 2.294655e-09
numerical: -0.891891 analytic: -0.891892, relative error: 5.692857e-08
numerical: 3.403461 analytic: 3.403461, relative error: 1.642551e-08
numerical: 2.405056 analytic: 2.405056, relative error: 1.195105e-08
numerical: -0.262773 analytic: -0.262773, relative error: 2.552053e-09
numerical: 1.204526 analytic: 1.204526, relative error: 1.379204e-09
numerical: 0.778575 analytic: 0.778575, relative error: 7.187283e-09
numerical: -4.179224 analytic: -4.179224, relative error: 9.852101e-09
numerical: -0.301104 analytic: -0.301104, relative error: 1.206384e-07
numerical: -1.859856 analytic: -1.859856, relative error: 3.559530e-08
numerical: 0.098469 analytic: 0.098469, relative error: 2.853476e-07
numerical: -0.034095 analytic: -0.034095, relative error: 2.612257e-07
numerical: -2.113373 analytic: -2.113373, relative error: 3.086922e-08
numerical: 0.299491 analytic: 0.299491, relative error: 5.602726e-08
numerical: -1.058683 analytic: -1.058683, relative error: 1.951871e-09
numerical: -1.933508 analytic: -1.933508, relative error: 8.249504e-09
numerical: 1.517070 analytic: 1.517070, relative error: 4.719595e-10
numerical: -0.213829 analytic: -0.213829, relative error: 1.041732e-07
numerical: -1.845483 analytic: -1.845483, relative error: 7.874346e-09
```

[29]:
```
# 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))

# W e 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: 2.336528e+00 computed in 0.084737s
vectorized loss: 2.336528e+00 computed in 0.002999s
Loss difference: 0.000000
Gradient difference: 0.000000
```

1.1.1 Stochastic Gradient Descent

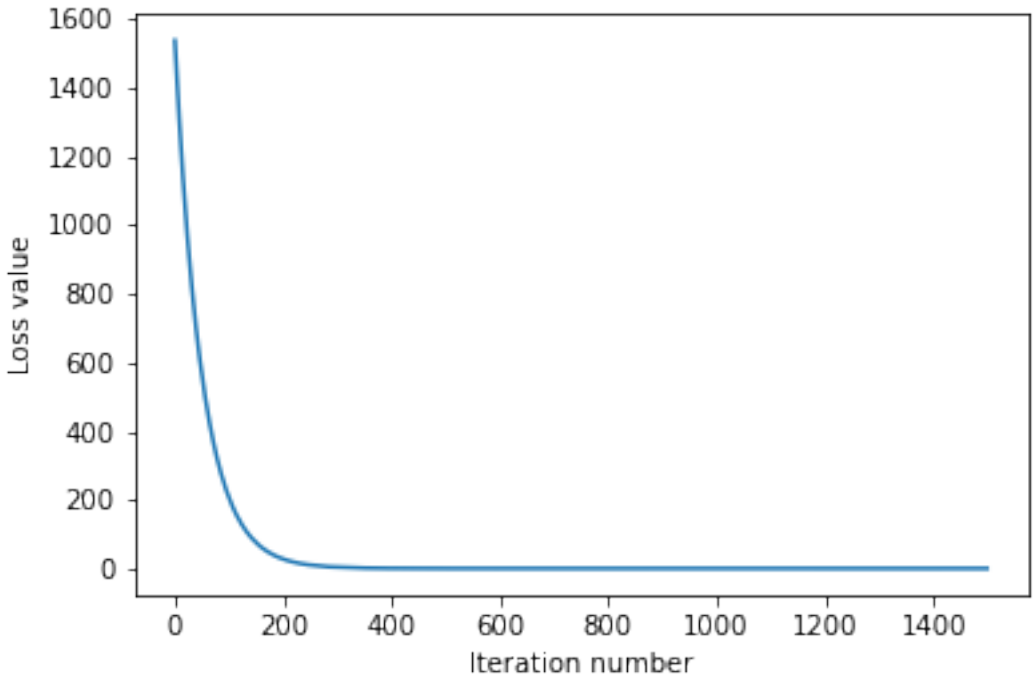
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[31]:
```
# 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 Softmax
softmax = Softmax()
tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7, reg=5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 1537.258967
iteration 100 / 1500: loss 207.095441
iteration 200 / 1500: loss 29.490943
iteration 300 / 1500: loss 5.796572
iteration 400 / 1500: loss 2.615515
```

```
iteration 500 / 1500: loss 2.214368
iteration 600 / 1500: loss 2.136669
iteration 700 / 1500: loss 2.158893
iteration 800 / 1500: loss 2.108022
iteration 900 / 1500: loss 2.156927
iteration 1000 / 1500: loss 2.165162
iteration 1100 / 1500: loss 2.148197
iteration 1200 / 1500: loss 2.111676
iteration 1300 / 1500: loss 2.117838
iteration 1400 / 1500: loss 2.110695
That took 6.341039s
```

[32]:
```
# 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()
```



[33]:
```
# Write the LinearClassifier.predict function and evaluate the performance on␣
↪both the
# training and validation set
y_train_pred = softmax.predict(X_train)
print( 'training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = softmax.predict(X_val)
```

```
print( 'validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.318245
validation accuracy: 0.325000
```

[35]:
```
# 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. #
# 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, 1e-6]
regularization_strengths = [2.0e4, 4e4]

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

# imitando la implementacion de sklearn, creo una grilla (matriz) de busqueda
# permutando los hiperparametros utilizando lists comprehensions
grid_search = [[lr, rs] for lr in learning_rates for rs in␣
 ↪regularization_strengths]

for learning_rate, regularization_strength in grid_search:
    softmax = Softmax()

    softmax.train(X_train, y_train, learning_rate=learning_rate,
                      reg=regularization_strength, num_iters=1500, verbose=True)

    y_train_pred = softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)

    train_accuracy = np.mean(y_train_pred == y_train)
    val_accuracy = np.mean(y_val_pred == y_val)

    results[(learning_rate, regularization_strength)] = \
                            (train_accuracy, val_accuracy)
```

```
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_softmax = softmax

# *****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)
```

```
iteration 0 / 1500: loss 623.066807
iteration 100 / 1500: loss 279.190307
iteration 200 / 1500: loss 126.048760
iteration 300 / 1500: loss 57.597823
iteration 400 / 1500: loss 26.883731
iteration 500 / 1500: loss 13.230844
iteration 600 / 1500: loss 7.058049
iteration 700 / 1500: loss 4.293707
iteration 800 / 1500: loss 3.058020
iteration 900 / 1500: loss 2.517822
iteration 1000 / 1500: loss 2.302306
iteration 1100 / 1500: loss 2.106710
iteration 1200 / 1500: loss 2.087322
iteration 1300 / 1500: loss 2.128839
iteration 1400 / 1500: loss 2.048018
iteration 0 / 1500: loss 1232.858069
iteration 100 / 1500: loss 248.192441
iteration 200 / 1500: loss 51.248378
iteration 300 / 1500: loss 11.954030
iteration 400 / 1500: loss 4.051047
iteration 500 / 1500: loss 2.548431
iteration 600 / 1500: loss 2.217373
iteration 700 / 1500: loss 2.148554
iteration 800 / 1500: loss 2.112109
iteration 900 / 1500: loss 2.152150
iteration 1000 / 1500: loss 2.111448
iteration 1100 / 1500: loss 2.084381
iteration 1200 / 1500: loss 2.134447
iteration 1300 / 1500: loss 2.152244
iteration 1400 / 1500: loss 2.081577
iteration 0 / 1500: loss 631.301990
```

```
iteration 100 / 1500: loss 2.251729
iteration 200 / 1500: loss 2.120687
iteration 300 / 1500: loss 2.043255
iteration 400 / 1500: loss 2.097377
iteration 500 / 1500: loss 2.083946
iteration 600 / 1500: loss 2.068589
iteration 700 / 1500: loss 2.142606
iteration 800 / 1500: loss 2.063436
iteration 900 / 1500: loss 2.046683
iteration 1000 / 1500: loss 2.055025
iteration 1100 / 1500: loss 2.113905
iteration 1200 / 1500: loss 2.030739
iteration 1300 / 1500: loss 2.063585
iteration 1400 / 1500: loss 2.042827
iteration 0 / 1500: loss 1223.877265
iteration 100 / 1500: loss 2.111657
iteration 200 / 1500: loss 2.135446
iteration 300 / 1500: loss 2.134297
iteration 400 / 1500: loss 2.147732
iteration 500 / 1500: loss 2.095060
iteration 600 / 1500: loss 2.143273
iteration 700 / 1500: loss 2.170388
iteration 800 / 1500: loss 2.159840
iteration 900 / 1500: loss 2.130325
iteration 1000 / 1500: loss 2.213742
iteration 1100 / 1500: loss 2.118226
iteration 1200 / 1500: loss 2.129317
iteration 1300 / 1500: loss 2.150088
iteration 1400 / 1500: loss 2.167909
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.331878 val accuracy: 0.351000
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.319020 val accuracy: 0.331000
lr 1.000000e-06 reg 2.000000e+04 train accuracy: 0.333429 val accuracy: 0.356000
lr 1.000000e-06 reg 4.000000e+04 train accuracy: 0.320531 val accuracy: 0.329000
best validation accuracy achieved during cross-validation: 0.356000
```

[36]:
```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: 0.349000
```

Inline Question 2 - *Softmax loss with Temperature*

Suppose we want to use a temperature parameter T for the softmax distribution:

$$P(i) = \frac{e^{\frac{f_i}{T}}}{\sum_j e^{\frac{f_j}{T}}}$$

where f_i is the score for class i - What values of T would make the model more confident about its predictions? - How would it affect the training process? You may test this experimentally.

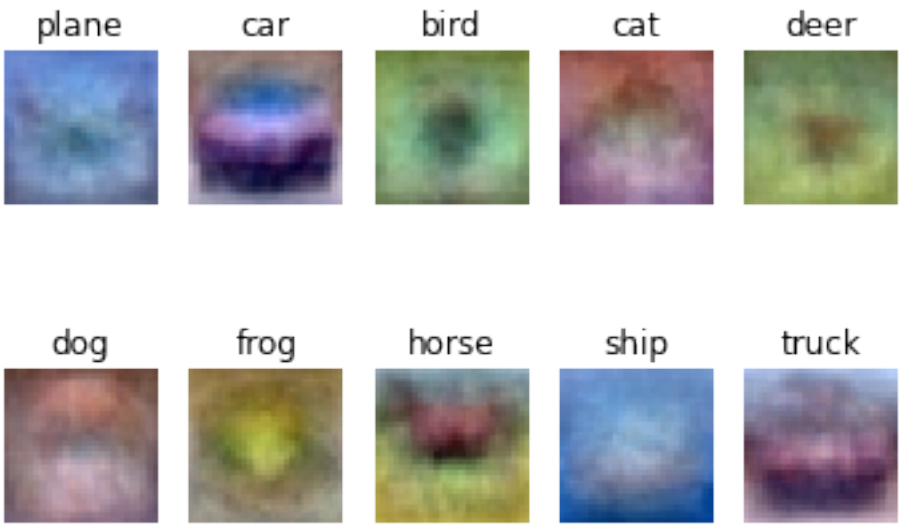
YourAnswer :

para $T = 1$ se obtiene la misma discrepancia. Para $T < 1$ el modelo tendrá mayor "confianza" en sus predicciones. Cuanto mayor sea la temperatura mas suave será el clasificador, esto quiere decir que tendrá menos confianza. Por ejemplo, si con $T = 1$, el clasificador saca probabilidades (0.01 , 0.01 , 0.98), si aumentamos T se podría obtener algo de la forma (0.2,0.2,0.6)

Al entrenar usando temperatura por ejemplo menor a 1 (modelo con mayor confianza en sus predicciones), se computa softmax utilizando valores mayores. El resultado es un clasificador mas propenso a errores. Se prueba (en otro notebook) y se observa que al entrenar con un valor menor a 1 para la temperatura, el término de loss disminuye mas rapidamente durante las iteraciones que con temperatura igual a 1. Se observa la tabla con los resultados en donde el accuracy en validacion se observa menor al de temperatura 1, coherente a lo expresado anteriormente:

T	Loss (última iter)	Training acc	Val acc
0.1	1.91	0.29	0.3
1	2.11	0.31	0.32

[37]:
```
# 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])
```


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

two_layer_net

October 20, 2020

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

[1]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%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))))
```

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

[2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

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

```
num_inputs = 5

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

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

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

2 Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the Softmax exercise: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

[3]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

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

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
```

```
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720496109664e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization loss.

[4]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

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

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

[8]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

```
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
↪verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
↪grads[param_name])))
```

```
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
```

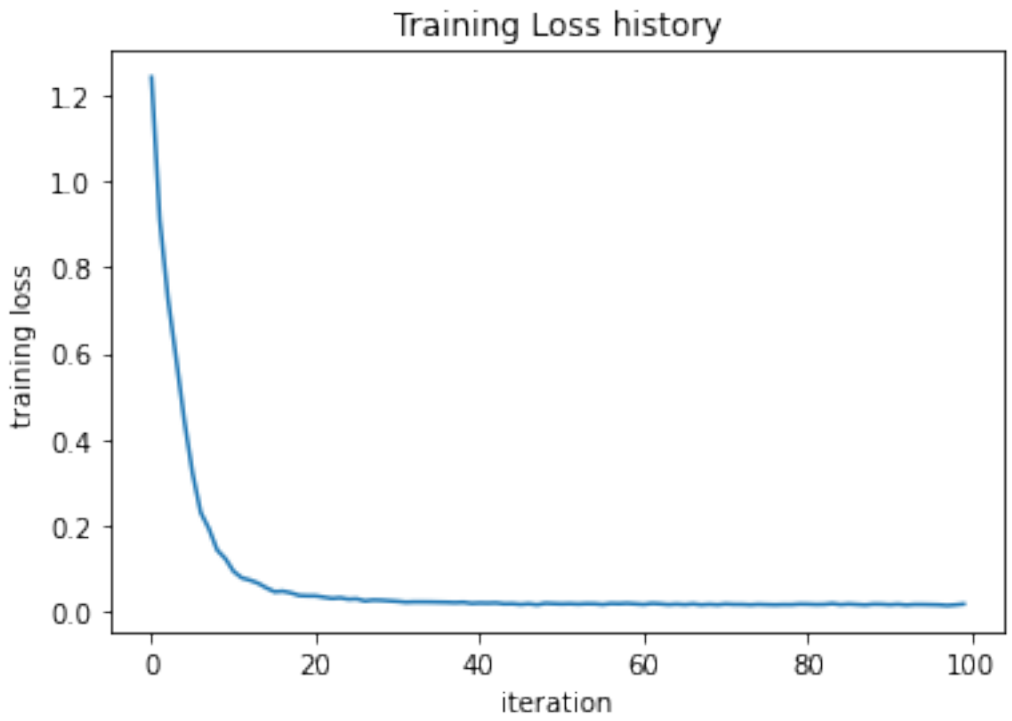
5 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the Softmax classifier. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the Softmax classifier. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

[9]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

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

```
Final training loss:  0.017149607938732093
```



6 Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

[10]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier.
    """
    # 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]

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

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

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


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

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

7 Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

[11]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

8 Debug the training

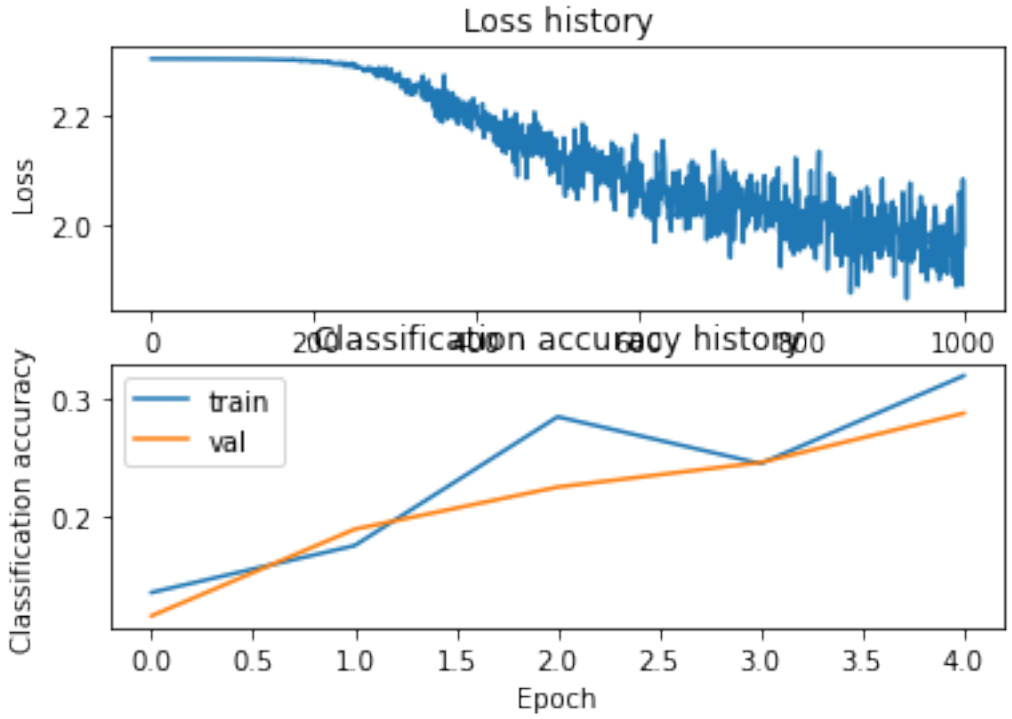
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

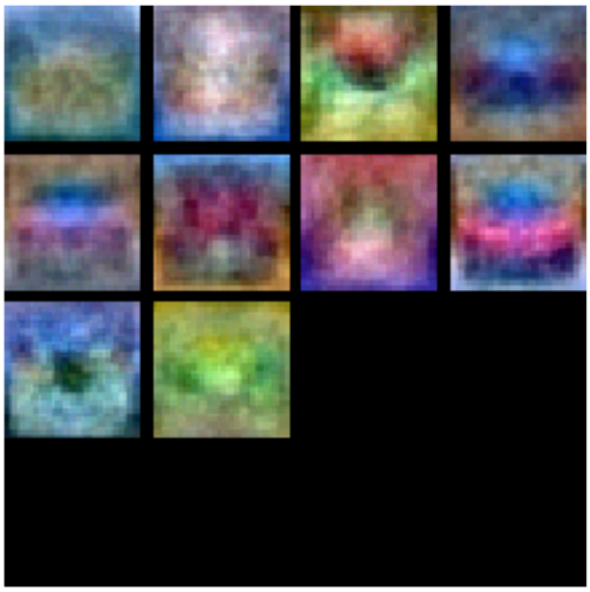
[12]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
```

```
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



[28]:
```
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(net)
```



9 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.).

Explain your hyperparameter tuning process below.

YourAnswer :

Para tunear los hiperparámetros utilizo la técnica de grid search que consiste en, dados varios valores para los parámetros sobre los cuales quiero probar el desempeño, creo todas las permutaciones posibles de los mismos. Luego entreno la red sobre cada una de estas permutaciones testeando el resultado sobre el conjunto de validación. Finalmente me quedo con la red con cuyos parametros obtuve un mayor accuracy en validación.

[29]:
```
best_net = None # store the best model into this

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained #
# model in best_net.                                                           #
#                                                                              #
# 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 the previous exercises.                         #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

#results = {}
best_val = -1

input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = [10, 50, 100]
learning_rates= [1e-3, 1e-4]
num_iters = [500, 1500, 2000]
regs = [0.0, 0.1, 0.25, 0.40, 0.6]

grid_search = [[hs, lr, ni, reg] for hs in hidden_sizes \
```

```
                                    for lr in learning_rates \
                                    for ni in num_iters \
                                    for reg in regs
                ]

for hidden_size, learning_rate, num_iters, reg in grid_search:

    net = TwoLayerNet(input_size, hidden_size, num_classes)

    stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=num_iters, batch_size=200,
            learning_rate=learning_rate, learning_rate_decay=0.95,
            reg=reg, verbose=False)

    val_acc = (net.predict(X_val) == y_val).mean()

    print('hidden_size:', hidden_size, 'learning_rate:', learning_rate,
          'num_iters:', num_iters, 'reg:', reg,
          '----> Validation accuracy: ', val_acc)

    if val_acc > best_val:
        best_val = val_acc
        best_net = net
```

```
hidden_size: 10 learning_rate: 0.001 num_iters: 500 reg: 0.0 ----> Validation
accuracy:  0.411
hidden_size: 10 learning_rate: 0.001 num_iters: 500 reg: 0.1 ----> Validation
accuracy:  0.401
hidden_size: 10 learning_rate: 0.001 num_iters: 500 reg: 0.25 ----> Validation
accuracy:  0.407
hidden_size: 10 learning_rate: 0.001 num_iters: 500 reg: 0.4 ----> Validation
accuracy:  0.399
hidden_size: 10 learning_rate: 0.001 num_iters: 500 reg: 0.6 ----> Validation
accuracy:  0.366
hidden_size: 10 learning_rate: 0.001 num_iters: 1500 reg: 0.0 ----> Validation
accuracy:  0.427
hidden_size: 10 learning_rate: 0.001 num_iters: 1500 reg: 0.1 ----> Validation
accuracy:  0.412
hidden_size: 10 learning_rate: 0.001 num_iters: 1500 reg: 0.25 ----> Validation
accuracy:  0.424
hidden_size: 10 learning_rate: 0.001 num_iters: 1500 reg: 0.4 ----> Validation
accuracy:  0.401
hidden_size: 10 learning_rate: 0.001 num_iters: 1500 reg: 0.6 ----> Validation
accuracy:  0.393
hidden_size: 10 learning_rate: 0.001 num_iters: 2000 reg: 0.0 ----> Validation
accuracy:  0.407
hidden_size: 10 learning_rate: 0.001 num_iters: 2000 reg: 0.1 ----> Validation
```

```
accuracy:  0.422
hidden_size: 10 learning_rate: 0.001 num_iters: 2000 reg: 0.25 ----> Validation
accuracy:  0.425
hidden_size: 10 learning_rate: 0.001 num_iters: 2000 reg: 0.4 ----> Validation
accuracy:  0.425
hidden_size: 10 learning_rate: 0.001 num_iters: 2000 reg: 0.6 ----> Validation
accuracy:  0.421
hidden_size: 10 learning_rate: 0.0001 num_iters: 500 reg: 0.0 ----> Validation
accuracy:  0.197
hidden_size: 10 learning_rate: 0.0001 num_iters: 500 reg: 0.1 ----> Validation
accuracy:  0.168
hidden_size: 10 learning_rate: 0.0001 num_iters: 500 reg: 0.25 ----> Validation
accuracy:  0.189
hidden_size: 10 learning_rate: 0.0001 num_iters: 500 reg: 0.4 ----> Validation
accuracy:  0.184
hidden_size: 10 learning_rate: 0.0001 num_iters: 500 reg: 0.6 ----> Validation
accuracy:  0.175
hidden_size: 10 learning_rate: 0.0001 num_iters: 1500 reg: 0.0 ----> Validation
accuracy:  0.314
hidden_size: 10 learning_rate: 0.0001 num_iters: 1500 reg: 0.1 ----> Validation
accuracy:  0.29
hidden_size: 10 learning_rate: 0.0001 num_iters: 1500 reg: 0.25 ----> Validation
accuracy:  0.29
hidden_size: 10 learning_rate: 0.0001 num_iters: 1500 reg: 0.4 ----> Validation
accuracy:  0.31
hidden_size: 10 learning_rate: 0.0001 num_iters: 1500 reg: 0.6 ----> Validation
accuracy:  0.286
hidden_size: 10 learning_rate: 0.0001 num_iters: 2000 reg: 0.0 ----> Validation
accuracy:  0.337
hidden_size: 10 learning_rate: 0.0001 num_iters: 2000 reg: 0.1 ----> Validation
accuracy:  0.336
hidden_size: 10 learning_rate: 0.0001 num_iters: 2000 reg: 0.25 ----> Validation
accuracy:  0.332
hidden_size: 10 learning_rate: 0.0001 num_iters: 2000 reg: 0.4 ----> Validation
accuracy:  0.32
hidden_size: 10 learning_rate: 0.0001 num_iters: 2000 reg: 0.6 ----> Validation
accuracy:  0.329
hidden_size: 50 learning_rate: 0.001 num_iters: 500 reg: 0.0 ----> Validation
accuracy:  0.428
hidden_size: 50 learning_rate: 0.001 num_iters: 500 reg: 0.1 ----> Validation
accuracy:  0.446
hidden_size: 50 learning_rate: 0.001 num_iters: 500 reg: 0.25 ----> Validation
accuracy:  0.445
hidden_size: 50 learning_rate: 0.001 num_iters: 500 reg: 0.4 ----> Validation
accuracy:  0.429
hidden_size: 50 learning_rate: 0.001 num_iters: 500 reg: 0.6 ----> Validation
accuracy:  0.449
hidden_size: 50 learning_rate: 0.001 num_iters: 1500 reg: 0.0 ----> Validation
```

```
accuracy:  0.489
hidden_size: 50 learning_rate: 0.001 num_iters: 1500 reg: 0.1 ----> Validation
accuracy:  0.495
hidden_size: 50 learning_rate: 0.001 num_iters: 1500 reg: 0.25 ----> Validation
accuracy:  0.476
hidden_size: 50 learning_rate: 0.001 num_iters: 1500 reg: 0.4 ----> Validation
accuracy:  0.475
hidden_size: 50 learning_rate: 0.001 num_iters: 1500 reg: 0.6 ----> Validation
accuracy:  0.479
hidden_size: 50 learning_rate: 0.001 num_iters: 2000 reg: 0.0 ----> Validation
accuracy:  0.5
hidden_size: 50 learning_rate: 0.001 num_iters: 2000 reg: 0.1 ----> Validation
accuracy:  0.501
hidden_size: 50 learning_rate: 0.001 num_iters: 2000 reg: 0.25 ----> Validation
accuracy:  0.477
hidden_size: 50 learning_rate: 0.001 num_iters: 2000 reg: 0.4 ----> Validation
accuracy:  0.49
hidden_size: 50 learning_rate: 0.001 num_iters: 2000 reg: 0.6 ----> Validation
accuracy:  0.489
hidden_size: 50 learning_rate: 0.0001 num_iters: 500 reg: 0.0 ----> Validation
accuracy:  0.222
hidden_size: 50 learning_rate: 0.0001 num_iters: 500 reg: 0.1 ----> Validation
accuracy:  0.208
hidden_size: 50 learning_rate: 0.0001 num_iters: 500 reg: 0.25 ----> Validation
accuracy:  0.217
hidden_size: 50 learning_rate: 0.0001 num_iters: 500 reg: 0.4 ----> Validation
accuracy:  0.228
hidden_size: 50 learning_rate: 0.0001 num_iters: 500 reg: 0.6 ----> Validation
accuracy:  0.228
hidden_size: 50 learning_rate: 0.0001 num_iters: 1500 reg: 0.0 ----> Validation
accuracy:  0.332
hidden_size: 50 learning_rate: 0.0001 num_iters: 1500 reg: 0.1 ----> Validation
accuracy:  0.331
hidden_size: 50 learning_rate: 0.0001 num_iters: 1500 reg: 0.25 ----> Validation
accuracy:  0.317
hidden_size: 50 learning_rate: 0.0001 num_iters: 1500 reg: 0.4 ----> Validation
accuracy:  0.329
hidden_size: 50 learning_rate: 0.0001 num_iters: 1500 reg: 0.6 ----> Validation
accuracy:  0.331
hidden_size: 50 learning_rate: 0.0001 num_iters: 2000 reg: 0.0 ----> Validation
accuracy:  0.363
hidden_size: 50 learning_rate: 0.0001 num_iters: 2000 reg: 0.1 ----> Validation
accuracy:  0.361
hidden_size: 50 learning_rate: 0.0001 num_iters: 2000 reg: 0.25 ----> Validation
accuracy:  0.367
hidden_size: 50 learning_rate: 0.0001 num_iters: 2000 reg: 0.4 ----> Validation
accuracy:  0.367
hidden_size: 50 learning_rate: 0.0001 num_iters: 2000 reg: 0.6 ----> Validation
```

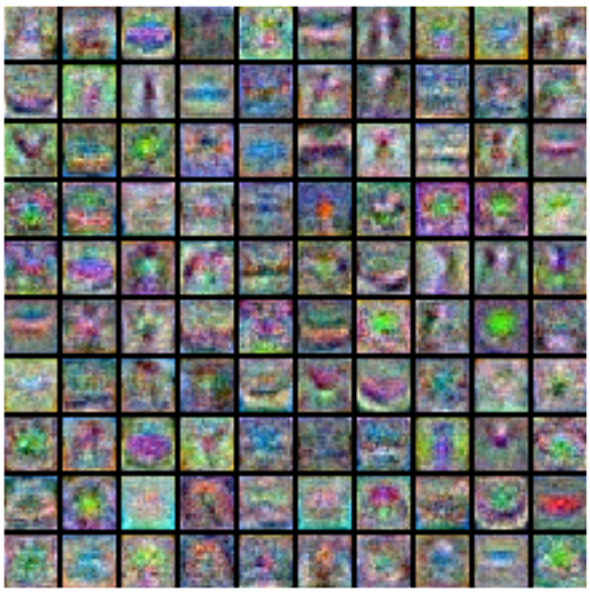
```
accuracy:  0.36
hidden_size: 100 learning_rate: 0.001 num_iters: 500 reg: 0.0 ----> Validation
accuracy:  0.438
hidden_size: 100 learning_rate: 0.001 num_iters: 500 reg: 0.1 ----> Validation
accuracy:  0.475
hidden_size: 100 learning_rate: 0.001 num_iters: 500 reg: 0.25 ----> Validation
accuracy:  0.439
hidden_size: 100 learning_rate: 0.001 num_iters: 500 reg: 0.4 ----> Validation
accuracy:  0.453
hidden_size: 100 learning_rate: 0.001 num_iters: 500 reg: 0.6 ----> Validation
accuracy:  0.436
hidden_size: 100 learning_rate: 0.001 num_iters: 1500 reg: 0.0 ----> Validation
accuracy:  0.504
hidden_size: 100 learning_rate: 0.001 num_iters: 1500 reg: 0.1 ----> Validation
accuracy:  0.507
hidden_size: 100 learning_rate: 0.001 num_iters: 1500 reg: 0.25 ----> Validation
accuracy:  0.504
hidden_size: 100 learning_rate: 0.001 num_iters: 1500 reg: 0.4 ----> Validation
accuracy:  0.473
hidden_size: 100 learning_rate: 0.001 num_iters: 1500 reg: 0.6 ----> Validation
accuracy:  0.463
hidden_size: 100 learning_rate: 0.001 num_iters: 2000 reg: 0.0 ----> Validation
accuracy:  0.528
hidden_size: 100 learning_rate: 0.001 num_iters: 2000 reg: 0.1 ----> Validation
accuracy:  0.515
hidden_size: 100 learning_rate: 0.001 num_iters: 2000 reg: 0.25 ----> Validation
accuracy:  0.494
hidden_size: 100 learning_rate: 0.001 num_iters: 2000 reg: 0.4 ----> Validation
accuracy:  0.495
hidden_size: 100 learning_rate: 0.001 num_iters: 2000 reg: 0.6 ----> Validation
accuracy:  0.493
hidden_size: 100 learning_rate: 0.0001 num_iters: 500 reg: 0.0 ----> Validation
accuracy:  0.235
hidden_size: 100 learning_rate: 0.0001 num_iters: 500 reg: 0.1 ----> Validation
accuracy:  0.231
hidden_size: 100 learning_rate: 0.0001 num_iters: 500 reg: 0.25 ----> Validation
accuracy:  0.23
hidden_size: 100 learning_rate: 0.0001 num_iters: 500 reg: 0.4 ----> Validation
accuracy:  0.238
hidden_size: 100 learning_rate: 0.0001 num_iters: 500 reg: 0.6 ----> Validation
accuracy:  0.236
hidden_size: 100 learning_rate: 0.0001 num_iters: 1500 reg: 0.0 ----> Validation
accuracy:  0.332
hidden_size: 100 learning_rate: 0.0001 num_iters: 1500 reg: 0.1 ----> Validation
accuracy:  0.336
hidden_size: 100 learning_rate: 0.0001 num_iters: 1500 reg: 0.25 ---->
Validation accuracy:  0.335
hidden_size: 100 learning_rate: 0.0001 num_iters: 1500 reg: 0.4 ----> Validation
```

```
accuracy:  0.338
hidden_size: 100 learning_rate: 0.0001 num_iters: 1500 reg: 0.6 ----> Validation
accuracy:  0.333
hidden_size: 100 learning_rate: 0.0001 num_iters: 2000 reg: 0.0 ----> Validation
accuracy:  0.36
hidden_size: 100 learning_rate: 0.0001 num_iters: 2000 reg: 0.1 ----> Validation
accuracy:  0.371
hidden_size: 100 learning_rate: 0.0001 num_iters: 2000 reg: 0.25 ---->
Validation accuracy:  0.368
hidden_size: 100 learning_rate: 0.0001 num_iters: 2000 reg: 0.4 ----> Validation
accuracy:  0.371
hidden_size: 100 learning_rate: 0.0001 num_iters: 2000 reg: 0.6 ----> Validation
accuracy:  0.367
```

[30]:
```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.528
```

[31]:
```
# Visualize the weights of the best network
show_net_weights(best_net)
```



10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

[32]:
```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.49
```

Inline Question

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. Decrease the regularization strength.
4. Add another hidden layer.
5. None of the above.

YourAnswer :

Solo la 1 aplica

YourExplanation :

1:(SI) Al agregar mayor puntos de entrenamiento la distribucion de estos se asemeja más a la distribución real de los datos, provocando que el clasificador entrenado generalice mejor.

2:(NO) Agregar mas neuronas le da mas libertad al clasificador, permitiendole definir fronteras de decision mas complejas, esto provocará un error menor en training pero error mayor en testing ya que al tener mas libertad, el clasificador se puede ajustar mas a los datos provocando overfitting

3:(NO) Si se disminuye el factor de regularización se penaliza menos el overfitting por lo que este aumenta (generalizando peor)

4:(NO) Mismo que en 2

features

October 20, 2020

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.

[1]:
```
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.

[2]:
```
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.

[3]:
```
from cs231n.features import *

num_color_bins = 10 # 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 Softmax on features

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

[5]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import Softmax

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [1e5, 1e6, 1e7]

results = {}
best_val = -1
best_softmax = None

pass
################################################################################
# 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 Softmax;     #
# save the best trained classifer in best_softmax. 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 [0.42] on the validation set.     #
################################################################################

# de la misma forma que en "softmax.ipynb":

grid_search = [[lr, rs] for lr in learning_rates for rs in␣
 ↪regularization_strengths]

for learning_rate, regularization_strength in grid_search:

    softmax = Softmax()

    softmax.train(X_train_feats, y_train, learning_rate=learning_rate,
                      reg=regularization_strength, num_iters=1500, verbose=True)

    y_train_pred = softmax.predict(X_train_feats)
    y_val_pred = softmax.predict(X_val_feats)

    train_accuracy = np.mean(y_train_pred == y_train)
    val_accuracy = np.mean(y_val_pred == y_val)

    results[(learning_rate, regularization_strength)] = \
                            (train_accuracy, val_accuracy)

    if val_accuracy > best_val:
        best_val = val_accuracy
        best_softmax = softmax

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

# 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)
```

```
iteration 0 / 1500: loss 149.550663
iteration 100 / 1500: loss 143.777450
```

```
iteration 200 / 1500: loss 138.229865
iteration 300 / 1500: loss 132.898242
iteration 400 / 1500: loss 127.778811
iteration 500 / 1500: loss 122.857539
iteration 600 / 1500: loss 118.128846
iteration 700 / 1500: loss 113.587832
iteration 800 / 1500: loss 109.222708
iteration 900 / 1500: loss 105.030953
iteration 1000 / 1500: loss 101.002793
iteration 1100 / 1500: loss 97.132525
iteration 1200 / 1500: loss 93.413940
iteration 1300 / 1500: loss 89.839614
iteration 1400 / 1500: loss 86.407772
iteration 0 / 1500: loss 1479.501900
iteration 100 / 1500: loss 992.101743
iteration 200 / 1500: loss 665.518770
iteration 300 / 1500: loss 446.691622
iteration 400 / 1500: loss 300.066022
iteration 500 / 1500: loss 201.820163
iteration 600 / 1500: loss 135.989265
iteration 700 / 1500: loss 91.879816
iteration 800 / 1500: loss 62.323744
iteration 900 / 1500: loss 42.520092
iteration 1000 / 1500: loss 29.250134
iteration 1100 / 1500: loss 20.358814
iteration 1200 / 1500: loss 14.401311
iteration 1300 / 1500: loss 10.409354
iteration 1400 / 1500: loss 7.734531
iteration 0 / 1500: loss 15051.610343
iteration 100 / 1500: loss 266.989161
iteration 200 / 1500: loss 6.957855
iteration 300 / 1500: loss 2.384463
iteration 400 / 1500: loss 2.304025
iteration 500 / 1500: loss 2.302610
iteration 600 / 1500: loss 2.302586
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 156.809128
iteration 100 / 1500: loss 105.828312
iteration 200 / 1500: loss 71.671518
iteration 300 / 1500: loss 48.782743
iteration 400 / 1500: loss 33.447139
```

```
iteration 500 / 1500: loss 23.170613
iteration 600 / 1500: loss 16.285388
iteration 700 / 1500: loss 11.671762
iteration 800 / 1500: loss 8.580223
iteration 900 / 1500: loss 6.509075
iteration 1000 / 1500: loss 5.121116
iteration 1100 / 1500: loss 4.190921
iteration 1200 / 1500: loss 3.568099
iteration 1300 / 1500: loss 3.150425
iteration 1400 / 1500: loss 2.870717
iteration 0 / 1500: loss 1678.959810
iteration 100 / 1500: loss 31.791647
iteration 200 / 1500: loss 2.821255
iteration 300 / 1500: loss 2.311705
iteration 400 / 1500: loss 2.302745
iteration 500 / 1500: loss 2.302588
iteration 600 / 1500: loss 2.302585
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 15669.732195
iteration 100 / 1500: loss 2.302585
iteration 200 / 1500: loss 2.302585
iteration 300 / 1500: loss 2.302585
iteration 400 / 1500: loss 2.302585
iteration 500 / 1500: loss 2.302585
iteration 600 / 1500: loss 2.302585
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 160.791511
iteration 100 / 1500: loss 5.090107
iteration 200 / 1500: loss 2.351604
iteration 300 / 1500: loss 2.303443
iteration 400 / 1500: loss 2.302599
iteration 500 / 1500: loss 2.302584
iteration 600 / 1500: loss 2.302583
iteration 700 / 1500: loss 2.302583
```

```
iteration 800 / 1500: loss 2.302583
iteration 900 / 1500: loss 2.302584
iteration 1000 / 1500: loss 2.302583
iteration 1100 / 1500: loss 2.302583
iteration 1200 / 1500: loss 2.302584
iteration 1300 / 1500: loss 2.302583
iteration 1400 / 1500: loss 2.302583
iteration 0 / 1500: loss 1530.635291
iteration 100 / 1500: loss 2.302585
iteration 200 / 1500: loss 2.302585
iteration 300 / 1500: loss 2.302585
iteration 400 / 1500: loss 2.302585
iteration 500 / 1500: loss 2.302585
iteration 600 / 1500: loss 2.302585
iteration 700 / 1500: loss 2.302585
iteration 800 / 1500: loss 2.302585
iteration 900 / 1500: loss 2.302585
iteration 1000 / 1500: loss 2.302585
iteration 1100 / 1500: loss 2.302585
iteration 1200 / 1500: loss 2.302585
iteration 1300 / 1500: loss 2.302585
iteration 1400 / 1500: loss 2.302585
iteration 0 / 1500: loss 15486.215044
iteration 100 / 1500: loss 15486.250317
iteration 200 / 1500: loss 15486.276848
iteration 300 / 1500: loss 15486.300586
iteration 400 / 1500: loss 15486.333561
iteration 500 / 1500: loss 15486.357835
iteration 600 / 1500: loss 15486.398622
iteration 700 / 1500: loss 15486.444732
iteration 800 / 1500: loss 15486.489778
iteration 900 / 1500: loss 15486.536283
iteration 1000 / 1500: loss 15486.574254
iteration 1100 / 1500: loss 15486.614661
iteration 1200 / 1500: loss 15486.653629
iteration 1300 / 1500: loss 15486.675352
iteration 1400 / 1500: loss 15486.704345
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.075796 val accuracy: 0.058000
lr 1.000000e-09 reg 1.000000e+06 train accuracy: 0.107714 val accuracy: 0.112000
lr 1.000000e-09 reg 1.000000e+07 train accuracy: 0.411347 val accuracy: 0.404000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.106102 val accuracy: 0.097000
lr 1.000000e-08 reg 1.000000e+06 train accuracy: 0.412796 val accuracy: 0.419000
lr 1.000000e-08 reg 1.000000e+07 train accuracy: 0.405327 val accuracy: 0.401000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.413347 val accuracy: 0.415000
lr 1.000000e-07 reg 1.000000e+06 train accuracy: 0.395163 val accuracy: 0.391000
lr 1.000000e-07 reg 1.000000e+07 train accuracy: 0.096694 val accuracy: 0.105000
best validation accuracy achieved during cross-validation: 0.419000
```

[6]:
```
# Evaluate your trained Softmax on the test set
y_test_pred = best_softmax.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.416
```

[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?

Your Answer :

Intentando darle un sentido a los errores de clasificacion me lleva a pensar en que al estar clasificando en funcion de texturas y colores, y no valores de pixeles,se observa que el clasificador determina como imagenes de aviones a imagenes con fondos uniformes (como la ultima de la columna en la que esta un ave con un fondo azul) ya que la mayoria de las fotos de aviones sacadas de dia tienen un fondo de un solo color. En cuanto a la textura proveniente de la caracteristica HoG, se observa que en su mayoria las imagenes mal clasificadas contienen figuras bien definidas sin demasiados detalles (como por eje las penúltima y la antepenúltima). Esto no se cumple para las imagenes del alce o del gato por ejemplo.

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]:
```
# 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, 155)
(49000, 154)
```

[20]:
```
from cs231n.classifiers.neural_net import TwoLayerNet

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)*****

# tomando la solucion planteada en "two_layer_net.ipynb":

best_val = -1

# learning_rates= [1e-3, 1e-4] la red no aprende
learning_rates= [4e-1, 4e-1, 3e-1, 2e-1]
num_iters = [500, 1500, 2000]
# regs = [0.0, 0.1, 0.25, 0.40, 0.6] restringe mucho el clasificador
regs = [0.0, 0.001, 0.01]

grid_search = [[lr, ni, reg] for lr in learning_rates \
                                for ni in num_iters \
                                for reg in regs
               ]

for learning_rate, num_iters, reg in grid_search:

    stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
            num_iters=num_iters, batch_size=200,
            learning_rate=learning_rate, learning_rate_decay=0.95,
            reg=reg, verbose=False)

    val_acc = (net.predict(X_val_feats) == y_val).mean()

    print('learning_rate:', learning_rate,
          'num_iters:', num_iters, 'reg:', reg,
          '----> Validation accuracy: ', val_acc)

    if val_acc > best_val:
        best_val = val_acc
        best_net = net

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

```
learning_rate: 0.4 num_iters: 500 reg: 0.0 ----> Validation accuracy:  0.523
learning_rate: 0.4 num_iters: 500 reg: 0.001 ----> Validation accuracy:  0.56
learning_rate: 0.4 num_iters: 500 reg: 0.01 ----> Validation accuracy:  0.502
learning_rate: 0.4 num_iters: 1500 reg: 0.0 ----> Validation accuracy:  0.589
learning_rate: 0.4 num_iters: 1500 reg: 0.001 ----> Validation accuracy:  0.592
learning_rate: 0.4 num_iters: 1500 reg: 0.01 ----> Validation accuracy:  0.535
```

```
learning_rate: 0.4 num_iters: 2000 reg: 0.0 ----> Validation accuracy:  0.576
learning_rate: 0.4 num_iters: 2000 reg: 0.001 ----> Validation accuracy:  0.582
learning_rate: 0.4 num_iters: 2000 reg: 0.01 ----> Validation accuracy:  0.519
learning_rate: 0.4 num_iters: 500 reg: 0.0 ----> Validation accuracy:  0.577
learning_rate: 0.4 num_iters: 500 reg: 0.001 ----> Validation accuracy:  0.573
learning_rate: 0.4 num_iters: 500 reg: 0.01 ----> Validation accuracy:  0.505
learning_rate: 0.4 num_iters: 1500 reg: 0.0 ----> Validation accuracy:  0.605
learning_rate: 0.4 num_iters: 1500 reg: 0.001 ----> Validation accuracy:  0.599
learning_rate: 0.4 num_iters: 1500 reg: 0.01 ----> Validation accuracy:  0.513
learning_rate: 0.4 num_iters: 2000 reg: 0.0 ----> Validation accuracy:  0.586
learning_rate: 0.4 num_iters: 2000 reg: 0.001 ----> Validation accuracy:  0.597
learning_rate: 0.4 num_iters: 2000 reg: 0.01 ----> Validation accuracy:  0.516
learning_rate: 0.3 num_iters: 500 reg: 0.0 ----> Validation accuracy:  0.585
learning_rate: 0.3 num_iters: 500 reg: 0.001 ----> Validation accuracy:  0.584
learning_rate: 0.3 num_iters: 500 reg: 0.01 ----> Validation accuracy:  0.531
learning_rate: 0.3 num_iters: 1500 reg: 0.0 ----> Validation accuracy:  0.59
learning_rate: 0.3 num_iters: 1500 reg: 0.001 ----> Validation accuracy:  0.599
learning_rate: 0.3 num_iters: 1500 reg: 0.01 ----> Validation accuracy:  0.516
learning_rate: 0.3 num_iters: 2000 reg: 0.0 ----> Validation accuracy:  0.594
learning_rate: 0.3 num_iters: 2000 reg: 0.001 ----> Validation accuracy:  0.584
learning_rate: 0.3 num_iters: 2000 reg: 0.01 ----> Validation accuracy:  0.528
learning_rate: 0.2 num_iters: 500 reg: 0.0 ----> Validation accuracy:  0.588
learning_rate: 0.2 num_iters: 500 reg: 0.001 ----> Validation accuracy:  0.578
learning_rate: 0.2 num_iters: 500 reg: 0.01 ----> Validation accuracy:  0.539
learning_rate: 0.2 num_iters: 1500 reg: 0.0 ----> Validation accuracy:  0.608
learning_rate: 0.2 num_iters: 1500 reg: 0.001 ----> Validation accuracy:  0.587
learning_rate: 0.2 num_iters: 1500 reg: 0.01 ----> Validation accuracy:  0.522
learning_rate: 0.2 num_iters: 2000 reg: 0.0 ----> Validation accuracy:  0.6
learning_rate: 0.2 num_iters: 2000 reg: 0.001 ----> Validation accuracy:  0.603
learning_rate: 0.2 num_iters: 2000 reg: 0.01 ----> Validation accuracy:  0.516
```

[21]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.525
```