knn

October 20, 2020

[ ]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'dlvis2020-entregables/assignment1_colab/cs231n/'
FOLDERNAME = 'dlvis2020-entregables/assignment1_colab/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-10-18 15:01:47--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  68.4MB/s    in 2.4s

2020-10-18 15:01:49 (68.4 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

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.

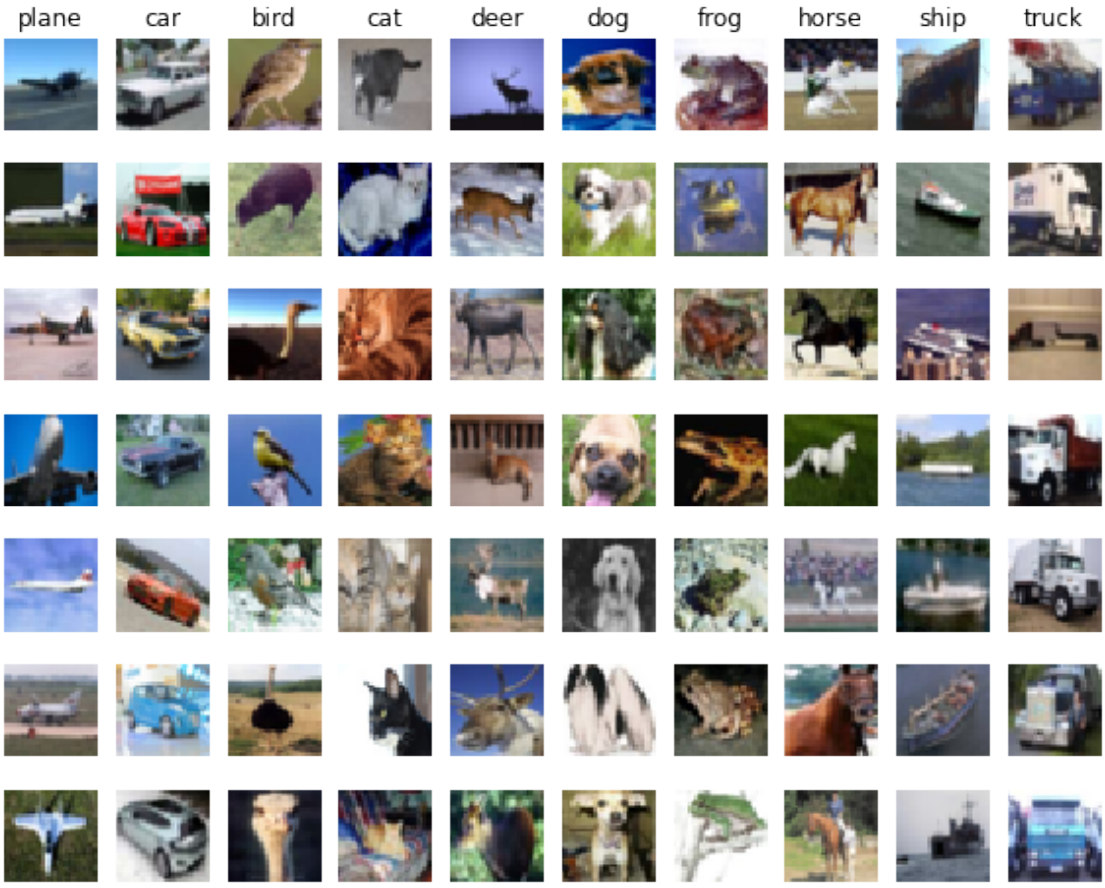
[ ]:
```
# Run some setup code for this notebook.

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
```

[ ]:
```
# 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,)
```

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



[ ]:
```
# 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)
```

[ ]:
```
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.

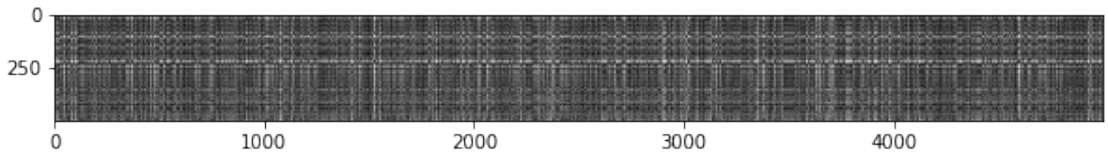
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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.

[ ]:
```
# 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)
```

[ ]:
```
# 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?

Your Answer :

- 1.1 una fila más brillantes implica un ejemplo de test distante de la gran mayoría de los ejemplos de entrenamiento. Esto se define como dato outlier o atípico, es decir, que está por fuera de los rangos estadísticos.
- 1.2 Para el caso de las columnas, decimos que el ejemplo de entrenamiento representado por dicha columna se encuentra lejos de la gran mayoría de los ejemplos de test. En este caso el dato atípico se encuentra en el entrenamiento.
- 1.3 El rango de la matriz es el número de filas (o columnas) linealmente independientes. Si no hubiera variación intra-clase, todos los ejemplos de una misma clase representarían al mismo punto en el espacio, por lo que todas las filas de una misma clase serían iguales. En este caso el rango de la matriz dist sería C, siendo C el número de clases. A demás tenemos el dato que la separación inter-clases es la misma para todo par de clases. suponiendo que dicha separación vale X, los ejemplos tendrían un 0 en la columna correspondiente a la clase a la que pertenecen, y en el resto de las columnas tendrían el valor X, ya que la distancia es la misma hacia el resto de las clases.

[ ]:
```
# 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`:

[ ]:
```
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 :

1,2,3 y 5

YourExplanation :

Para el caso 1, si a todos los pixels de todas las imágenes se le substrae el mismo valor, la distancia entre las imágenes no varía. De forma similar para el caso de la normalización considerada en el punto 3, la proporción de la distancia no cambia ya que a todos los pixeles de todas las imágenes las divido por el mismo valor.

Para el caso 2, a cada imagen se le resta una matriz (los valores del pixel wise para cada pixel). Al hacer la distancia entre dos imágenes, para la cual se calcula la sumatoria del valor absoluto de la diferencia entre cada pixel, los valores del pixel wise mean se anulan, por lo que la distancia L1 se mantiene.

Para el caso 5, la rotación 90 grados se utiliza la matriz de rotación:[[0 -1], [1 0]]. para obtener las nuevas coordenadas basta multiplicar dicha matriz por las coordenadas iniciales. para un punto genérico (x,y), las nuevas coordenadas serán: (-y,x). Tomando la norma L1 vemos que el valor de la norma se mantiene ya que los valores |x|+|y| = |-y| + |x|

Para el caso 4 se divide a la diferencia pixel a pixel por un valor diferente para cada pixel, por lo el valor de la distancia utilizando la norma L1 no se mantendrá.

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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 37.288851 seconds
One loop version took 26.054924 seconds
No loop version took 0.555064 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.

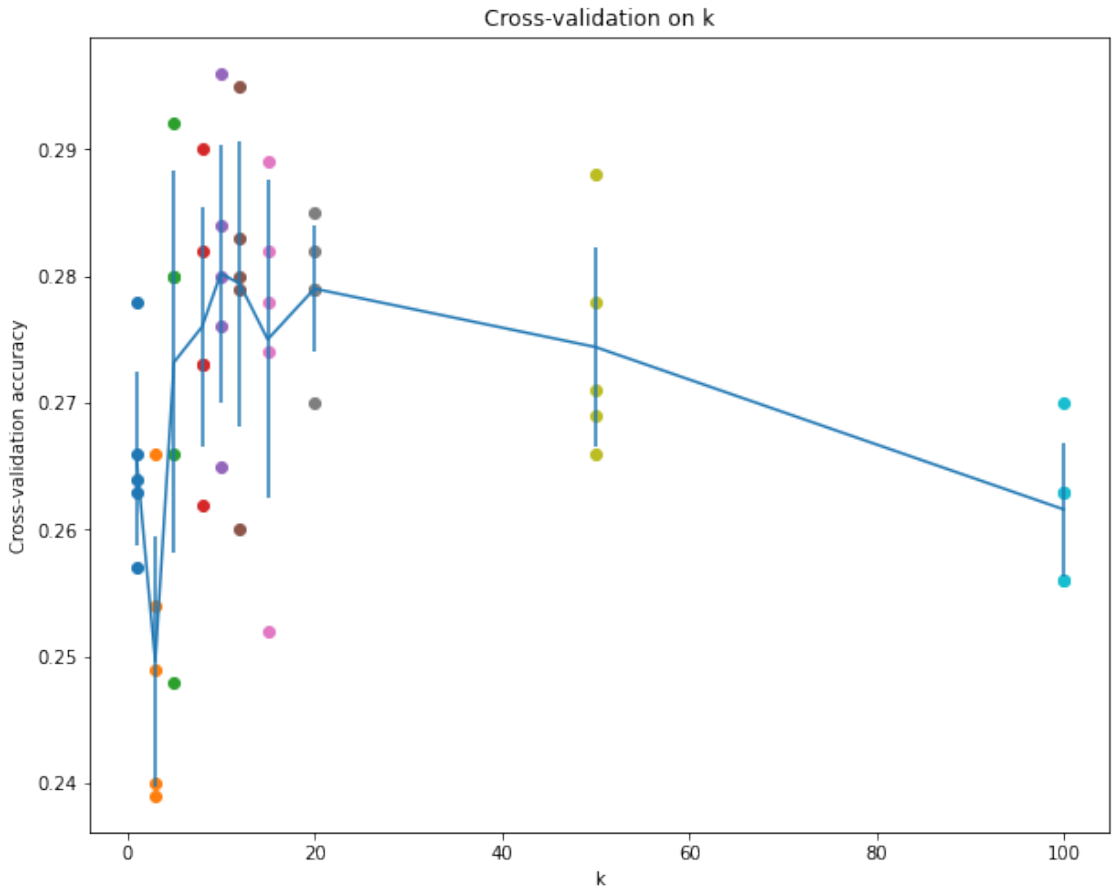
[ ]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)*****

# para cada opción de K
for k in k_choices:
  # para cada fold
  k_to_accuracies[k] = []
  for i in range(num_folds):
    # genero conjuntos de entrenamiento y test
    X_testAux = X_train_folds[i]
    y_testAux = y_train_folds[i]
    X_trainAux = np.concatenate(X_train_folds[:i] + X_train_folds[i + 1:])
    y_trainAux = np.concatenate(y_train_folds[:i] + y_train_folds[i + 1:])

    # creo clasificador y cargo datos de entrenamiento
    classifierAux = KNearestNeighbor()
    classifierAux.train(X_trainAux, y_trainAux)
    # calculo las distancias con el método más performante
    dists_Aux = classifierAux.compute_distances_no_loops(X_testAux)
    # hago las predicciones
    y_test_predAux = classifierAux.predict_labels(dists_Aux, k)
    #calculo accurancy
    num_correct = np.sum(y_test_predAux == y_testAux)
    accuracy = float(num_correct) / len(y_testAux)
    #agrego dato a diccionario
    k_to_accuracies[k] = k_to_accuracies.get(k) + [accuracy]


# *****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
```

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



[ ]:
```
# 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 = 20

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 136 / 500 correct => accuracy: 0.272000
```

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 : Sentencias 2 y 4 son verdaderas.

YourExplanation :

1. Falso. La función de distancia no suele ser lineal, por lo que la barrera de decisión tampoco lo es.
2. Verdadero. Cuanto más pequeño es k, más flexible es el método, por lo que se ajusta mejor al conjunto de entrenamiento. Esto reduce el error del estimador en dichas instancias (sesgo), pero son más propensos al sobreajuste.
3. Falso. Dado que 1-NN es más propenso al sobreajuste, es esperable que la performance en un conjunto de test sea peor que para un método menos flexible, por ejemplo KNN que considere mayor cantidad de vecinos.
4. Verdadero. Para clasificar una instancia es necesario calcular la distancia a todas las instancias de entrenamiento, por lo que si dicho conjunto es más grande, se requerirá de más tiempo para la clasificación.

Inline Question 4

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

YourAnswer :

Si. La regresión local ponderada es una generalización de KNN. para clasificar un punto aproximamos una función objetivo y luego clasificamos el punto. La función objetivo puede ser una función lineal o cuadrática normalmente. Para estimar la función podemos usar solo los k vecinos más cercanos, todos los ejemplos, o ponderar con más peso los k vecinos.

Con este método generamos un clasificador para etiquetar cada nuevo punto. Luego el clasificador es descartado.

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[ ]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/k_nearest_neighbor.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```

[ ]:
```

```

softmax

October 20, 2020

[15]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'dlvis2020-entregables/assignment1_colab/cs231n/'
FOLDERNAME = 'dlvis2020-entregables/assignment1_colab/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-10-18 15:14:53--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  15.9MB/s    in 11s

2020-10-18 15:15:05 (14.4 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

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

[16]:
```
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
```

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

[17]:
```
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`.

[18]:
```
# 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
%reload_ext autoreload

# 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: 2.397708
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?

Your Answer :

- 1.1 Dado que se cuenta con 10 clases, la probabilidad que la clasificación sea de una determinada clase es 0.1. La función de entropía cruzada es el promedio de -log(probabilidad ejemplo sea de clase objetivo). Por lo que se espera que en un conjunto importante de ejemplos, la entropía cruzada se aproxime a -log(0.1)
- 1.2 si la probabilidad calculada por softmax para cada ejemplo fuera 1 para para la clase objetivo del ejemplo (probabilidad ideal a alcanzar), la entropía cruzada valdría -log(1)=0. Por el contrario, si para todos los ejemplos la probabilidad calculada por softmax fuera 0 para la clase objetivo del ejemplo, entonces la entropía cruzada valdría -log(0). Dicho valor no existe pero al aproximarse a cero tinde a infinito.

[19]:
```
# 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: -1.815699 analytic: -1.815699, relative error: 3.931774e-08
numerical: -0.150983 analytic: -0.150983, relative error: 1.027425e-07
numerical: 1.789852 analytic: 1.789852, relative error: 5.088352e-09
numerical: -0.390115 analytic: -0.390115, relative error: 4.330402e-08
numerical: -3.318749 analytic: -3.318749, relative error: 9.103284e-09
numerical: -1.697004 analytic: -1.697004, relative error: 2.519338e-08
numerical: 0.883139 analytic: 0.883139, relative error: 6.723554e-08
numerical: 0.211946 analytic: 0.211946, relative error: 1.009199e-08
numerical: -0.640016 analytic: -0.640016, relative error: 8.386218e-08
numerical: -0.581901 analytic: -0.581901, relative error: 6.142262e-08
numerical: 2.570007 analytic: 2.570007, relative error: 1.556551e-08
numerical: 0.001246 analytic: 0.001246, relative error: 3.074566e-05
numerical: 0.504610 analytic: 0.504609, relative error: 2.122397e-07
numerical: -1.943112 analytic: -1.943112, relative error: 1.317034e-08
numerical: -3.616700 analytic: -3.616700, relative error: 5.413228e-09
```

```
numerical: 0.091103 analytic: 0.091103, relative error: 6.349030e-07
numerical: -0.691369 analytic: -0.691369, relative error: 3.919634e-08
numerical: 0.970920 analytic: 0.970920, relative error: 6.109424e-08
numerical: 0.929903 analytic: 0.929903, relative error: 5.697553e-09
numerical: 2.104541 analytic: 2.104541, relative error: 6.000966e-09
```

[20]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[21]:
```
# 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: 2.397708e+00 computed in 0.166404s
vectorized loss: 2.397708e+00 computed in 0.013198s
Loss difference: 0.000000
Gradient difference: 0.000000
```

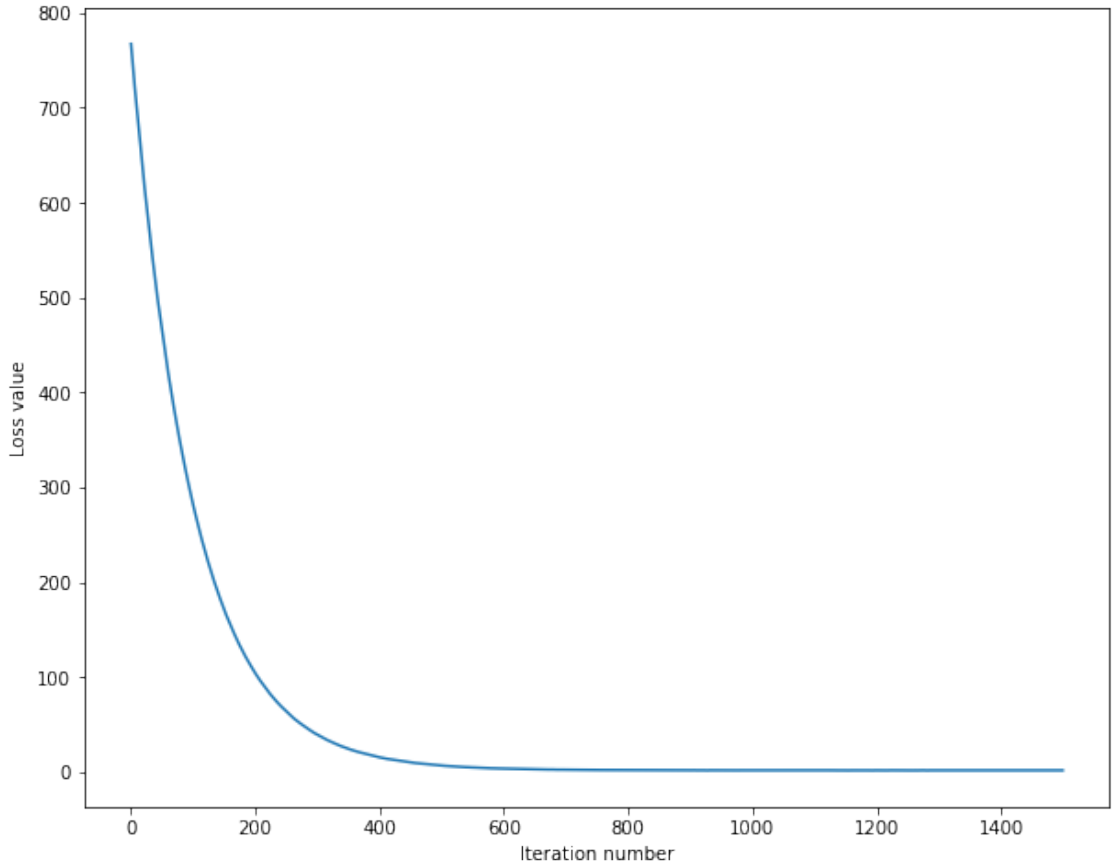
1.1.1 Stochastic Gradient Descent

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`.

[22]:
```
# 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 767.293195
iteration 100 / 1500: loss 281.673702
iteration 200 / 1500: loss 104.417729
iteration 300 / 1500: loss 39.561488
iteration 400 / 1500: loss 15.797611
iteration 500 / 1500: loss 7.158671
iteration 600 / 1500: loss 3.962647
iteration 700 / 1500: loss 2.739818
iteration 800 / 1500: loss 2.317414
iteration 900 / 1500: loss 2.200133
iteration 1000 / 1500: loss 2.119450
iteration 1100 / 1500: loss 2.105190
iteration 1200 / 1500: loss 2.094114
iteration 1300 / 1500: loss 2.053705
iteration 1400 / 1500: loss 2.085243
That took 7.476996s
```

[23]:
```
# 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()
```



[24]:
```
# 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: 0.329286
validation accuracy: 0.338000
```

[25]:
```
# 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)*****

modelo = Softmax()

for i in learning_rates:
    for j in regularization_strengths:
        loss_hist = modelo.train(X_train, y_train, i, j, num_iters=1500)

        prediccionesTrain = modelo.predict(X_train)
        prediccionesValidacion = modelo.predict(X_val)

        accurancy_train = np.mean(y_train == prediccionesTrain)
        accurancy_validation = np.mean(y_val == prediccionesValidacion)

        results[(i, j)] = (accurancy_train, accurancy_validation)

        if accurancy_validation > best_val:
            best_val = accurancy_validation
            best_softmax = modelo

# *****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)
```

```
lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.350408 val accuracy: 0.370000
```

```
lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.333571 val accuracy: 0.353000
lr 1.000000e-06 reg 2.000000e+04 train accuracy: 0.344878 val accuracy: 0.353000
lr 1.000000e-06 reg 4.000000e+04 train accuracy: 0.332102 val accuracy: 0.343000
best validation accuracy achieved during cross-validation: 0.370000
```

[26]:
```
# 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: 0.343000
```

Inline Question 2 - *Softmax loss with Temperature*
Suppose we want to use a temperature parameter T for the softmax distribution:

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

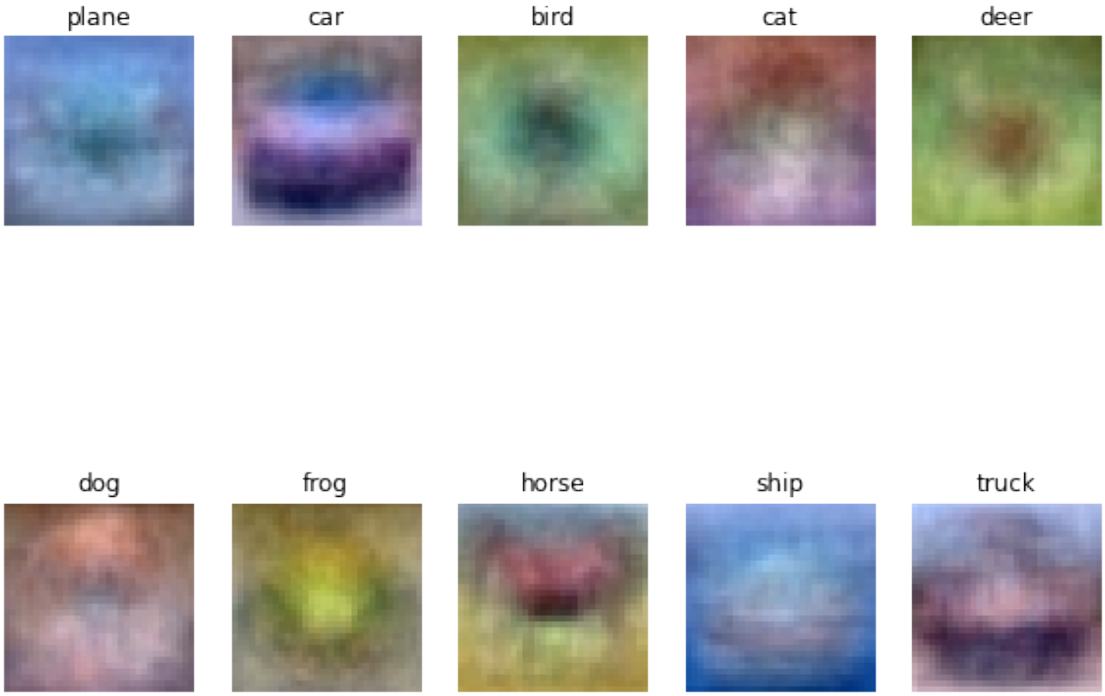
where f_i is the score for class i - What values of T would make the model more confident about its predictions? - How would it affect the training process? You may test this experimentally.

YourAnswer : - El uso del parámetro T se utiliza para suavizar las probabilidades estimadas por el modelo. A mayores valores de T, las probabilidades de cada clase para un ejemplo tenderán a ser parecidas, teniendo menos confianza en las predicciones. A mayores valores de T, se penaliza más a las probabilidades mayores, sobre las menores, debido a que exp es una función creciente. Si T es 1, las probabilidades calculadas serán las originales (sin tener en cuenta T), y será el caso en el que el modelo es más confiable.

- En el entrenamiento, a mayor valor de T (con T=10), la pérdida se acerca a -log(0.1). Esto quiere decir que a todas las clases le está asignando probabilidades cercanas a 0.1 (todas las clases con la misma probabilidad). En particular un clasificador así no sirve porque no tiene confianza al momento de predecir una clase para un ejemplo. Con T más bajo, se pueden suavizar las probabilidades, para llegar a un compromiso entre aleatoriedad y confianza.

[27]:
```
# 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])
```



[27]:

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[28]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/softmax.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
```

```
    f.write(''.join(open(files).readlines()))
```

two_layer_net

October 20, 2020

[ ]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'dlvis2020-entregables/assignment1_colab/cs231n/'
FOLDERNAME = 'dlvis2020-entregables/assignment1_colab/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-10-18 15:29:37--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  46.1MB/s    in 3.7s

2020-10-18 15:29:41 (44.0 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

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.

[ ]:
```
# 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.

[ ]:
```
# 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.

[ ]:
```
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:
3.6802720745909845e-08
```

3 Forward pass: compute loss

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

[ ]:
```
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:
0.018965419606062905
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

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:

[ ]:
```
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])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865070e-11
W1 max relative error: 4.090896e-09
b1 max relative error: 2.738421e-09
```

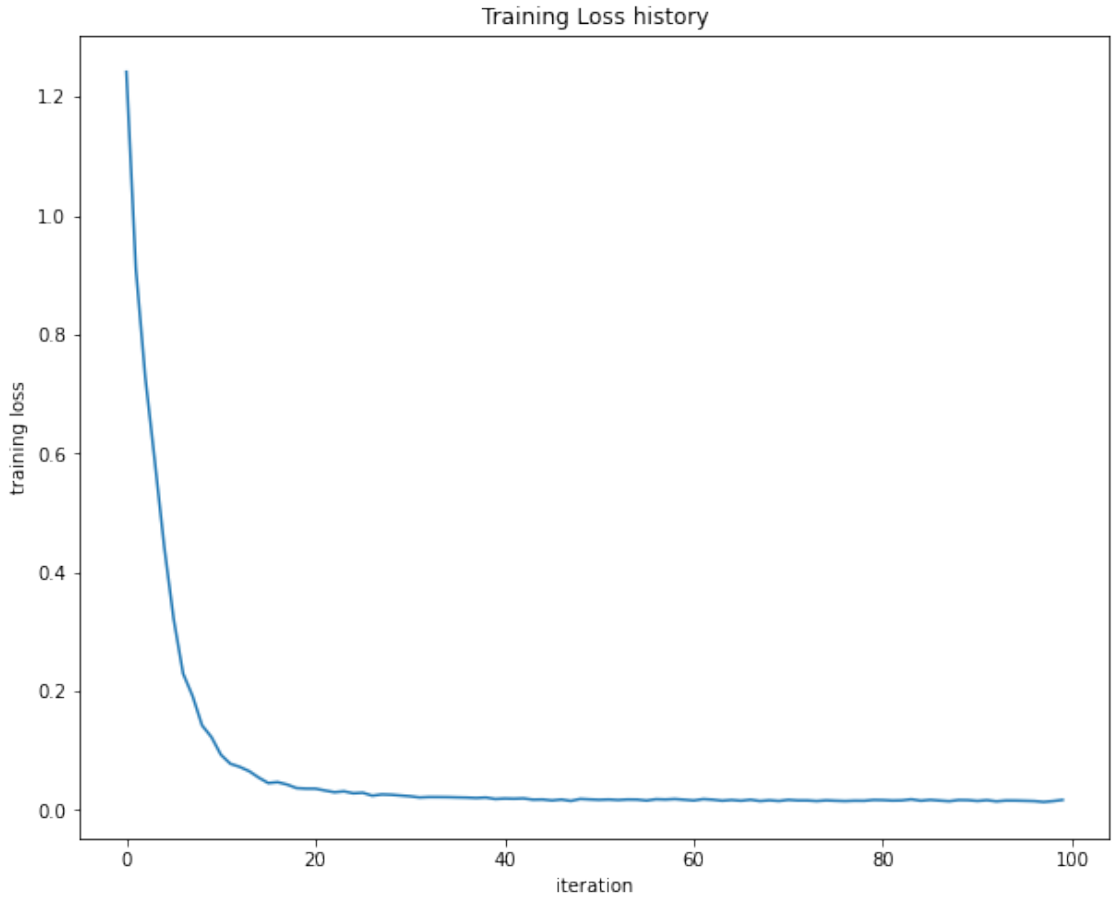
5 Train the network

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.

[ ]:
```
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:  0.017143673679355275
```



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.

[ ]:
```
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.

[ ]:
```
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 2.302762
iteration 100 / 1000: loss 2.302358
iteration 200 / 1000: loss 2.297404
iteration 300 / 1000: loss 2.258897
iteration 400 / 1000: loss 2.202975
iteration 500 / 1000: loss 2.116816
iteration 600 / 1000: loss 2.049789
iteration 700 / 1000: loss 1.985711
iteration 800 / 1000: loss 2.003727
iteration 900 / 1000: loss 1.948076
Validation accuracy:  0.287
```

8 Debug the training

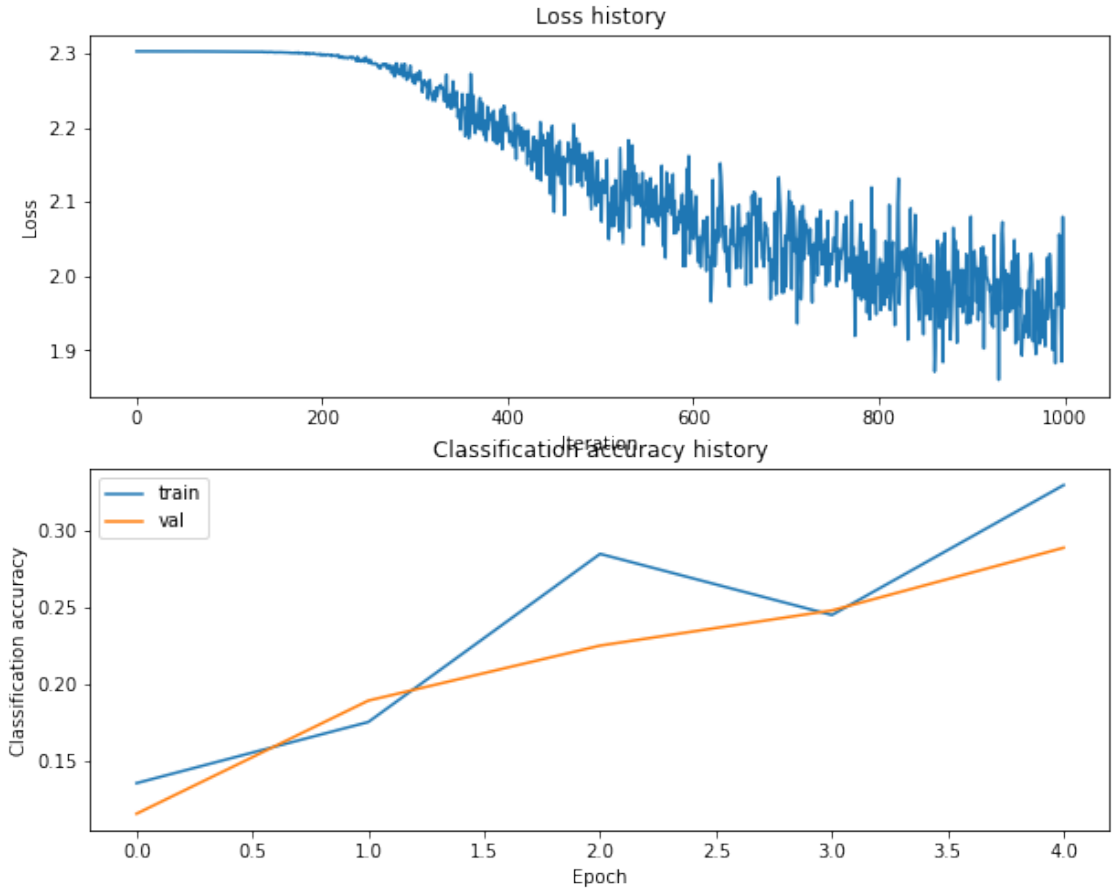
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

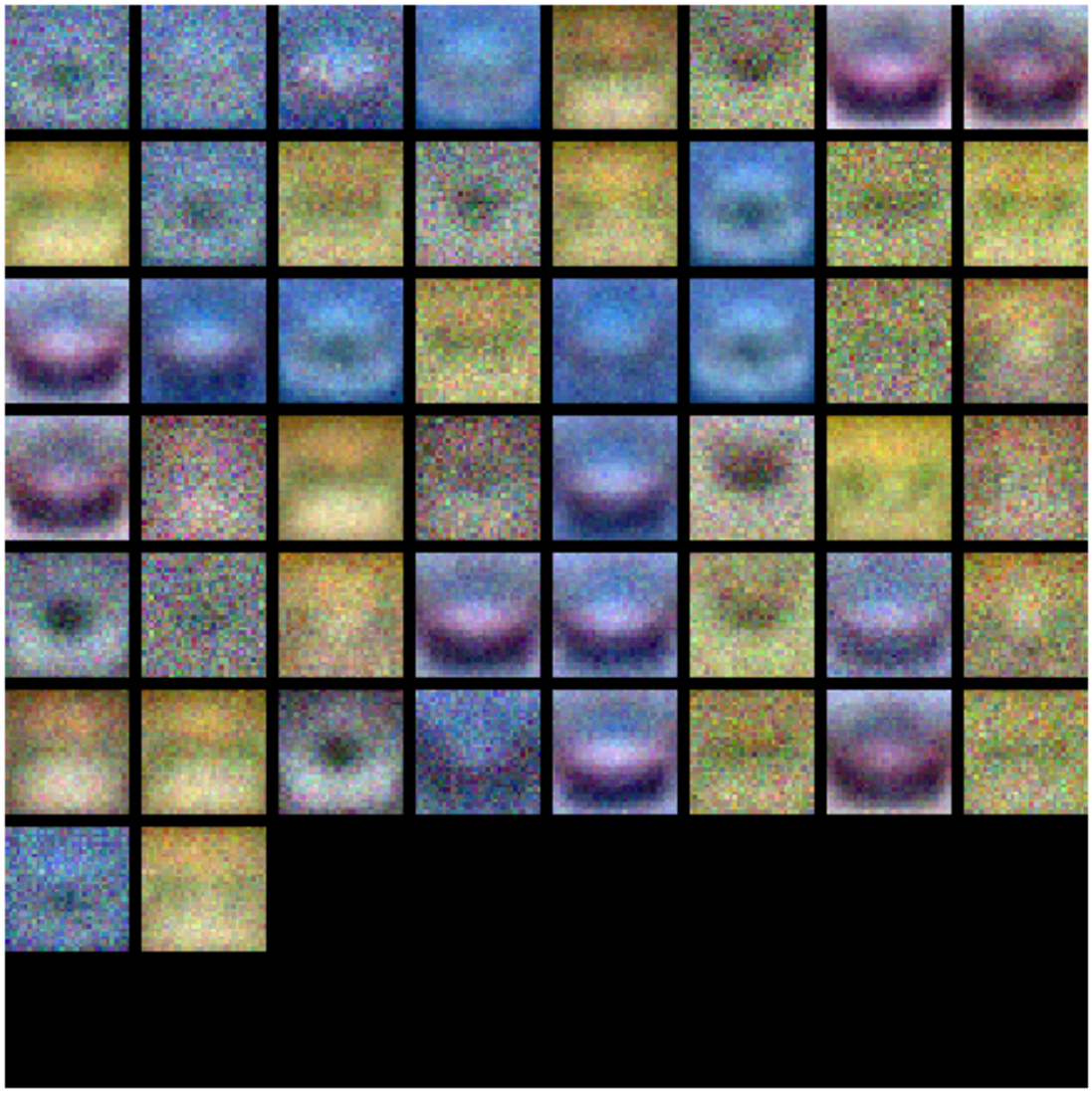
[ ]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
```

```
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()
```



[ ]:
```
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.

Your Answer :

Para el tuneo de parámetros se eligió variar la cantidad de unidades de la capa oculta, el parámetro de aprendizaje, la tasa de enfriamiento del parámetro de aprendizaje y el parámetro de regularización.

Al inicio se tomaron valores mayores del parámetro de aprendizaje (0.1 o 0.01) pero el algoritmo no converge, ya que hace que el paso avance tanto que se pase del mínimo, y por ende no converja. existen varios métodos para hallar el paso óptimo en cada iteración, como line Search o Armijo, pero en este ejercicio estamos utilizando el paso decreciente.

Con respecto a esto último, al inicio se utilizaron 3 valores diferentes de tasas de decrecimiento del paso. Fonalmente la tasa de decrecimiento 0.85 se quitó ya que disminuímos mucho la tasa de aprendizaje en cada iteración, y corremos riesgo que el entrenamiento se enfríe demasiado.

Se probó también con diferentes parámetros de regularización para evitar sobreajuste. tasas de regularización bajas servirán mucho para los casos en que probamos con muchas unidades en la capa interna de la red. Vemos que con una tasa alta (0.8) el algoritmo no coverge.

[ ]:
```
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


hidden_sizes = [100, 1500]
num_classes = 10
learning_rates = [0.001, 0.005]
learning_rate_decay = [0.95, 0.90]
regularization_strengths = [0.8, 0.1]

input_size = 32 * 32 * 3
num_classes = 10


for i in learning_rates:
    for j in regularization_strengths:
        for k in hidden_sizes:
            for l in learning_rate_decay:
                    #Modelo
                    net = TwoLayerNet(input_size, k, num_classes)

                    # Train the network
                    stats = net.train(X_train, y_train, X_val, y_val,
                        num_iters=2000, batch_size=200,
                        learning_rate=i, learning_rate_decay= l,
                        reg=j, verbose=True)

                    prediccionesTrain = net.predict(X_train)
                    prediccionesValidacion = net.predict(X_val)

                    accurancy_train = np.mean(y_train == prediccionesTrain)
                    accurancy_validation = np.mean(y_val ==␣
 ↪prediccionesValidacion)

                    results[(i, j)] = (accurancy_train, accurancy_validation)
                    show_net_weights(net)
```

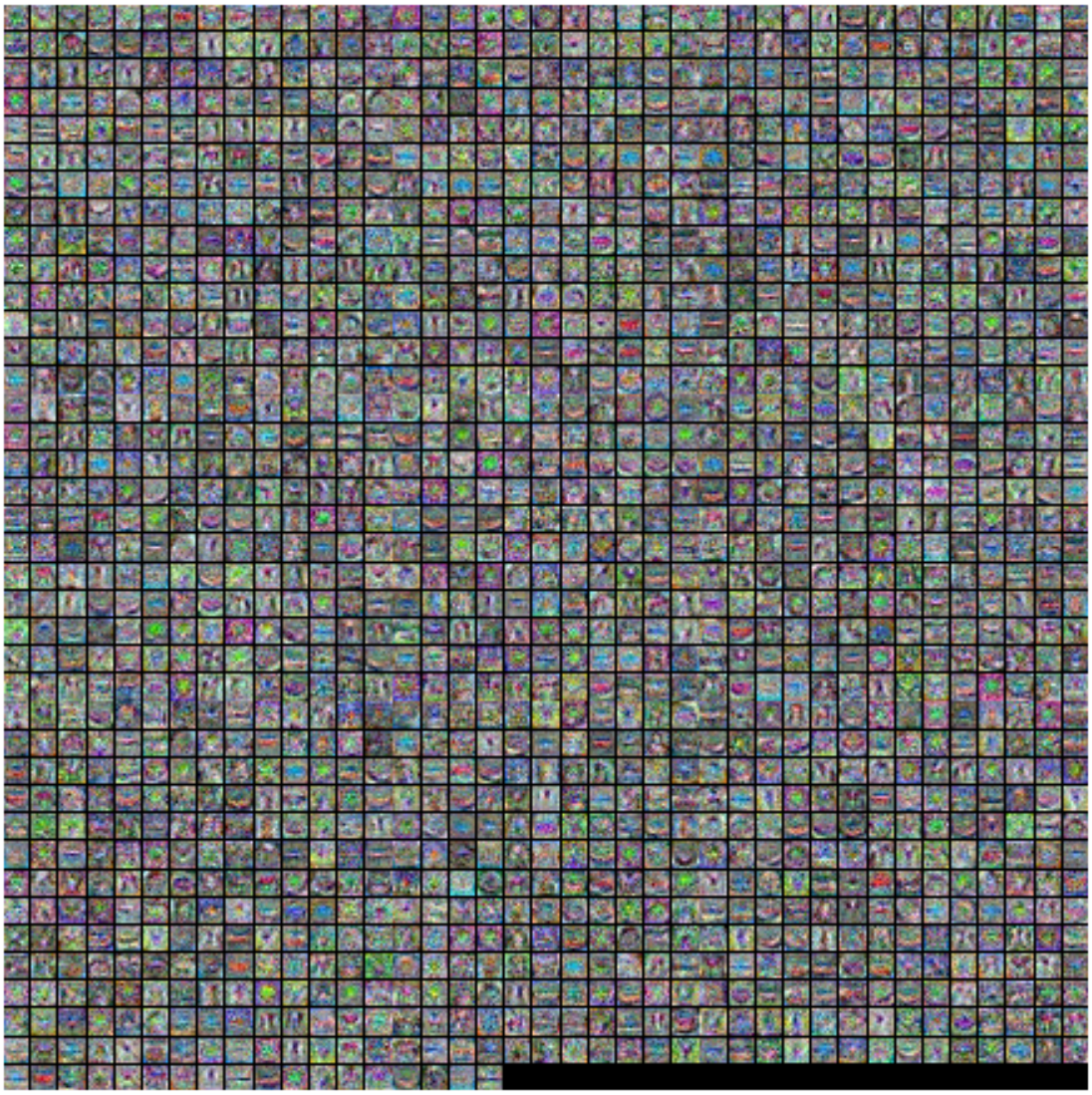
```
                    print('lr %e reg %e hs %e lrd %e train accuracy: %f val␣
↪accuracy: %f' % (i, j, k, l, accurancy_train, accurancy_validation))

                    if accurancy_validation > best_val:
                        best_val = accurancy_validation
                        best_net = net

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

Output hidden; open in https://colab.research.google.com to view.

[ ]:
```
# 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: 0.541

[ ]:
```
# 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%.

[ ]:
```
# 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:  0.537
```

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 : 1
YourExplanation :
las opciones 2 y 4 no ayudan a disminuir el sobreajuste ya que al agregar más capas o más unidades en una capa, implicará mayor cantidad de parámetros a ajustar. En esos caso se aumenta el riesgo del sobreajuste a los datos de entrenamiento, sobretodo si tenemos pocos.

aumentar el dataset de entrenamiento siempre es una buena idea, ya que el algoritmo podrá generalizar más facilmente.

decrementar el parámetro de regularización no ayudará a tener parámetros de la red pequeños, sino que al contrario no los penalizará. En el caso que fuera aumentar la regularización, ahí si se disminuye el sobreajuste a los datos.

---

11 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[ ]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['cs231n/classifiers/neural_net.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```

features

October 20, 2020

[1]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'dlvis2020-entregables/assignment1_colab/cs231n/'
FOLDERNAME = 'dlvis2020-entregables/assignment1_colab/cs231n/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd cs231n/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-10-18 14:18:53--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[===================>] 162.60M  37.6MB/s    in 4.8s

2020-10-18 14:18:58 (34.0 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]

cifar-10-batches-py/
```

```
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content
```

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.

[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
```

1.1 Load data

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

[3]:
```
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.

[4]:
```
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.   ␣
 ↪#
################################################################################
modelo = Softmax()

for i in learning_rates:
    for j in regularization_strengths:
        loss_hist = modelo.train(X_train_feats, y_train, i, j, num_iters=1500)

        prediccionesTrain = modelo.predict(X_train_feats)
        prediccionesValidacion = modelo.predict(X_val_feats)

        accurancy_train = np.mean(y_train == prediccionesTrain)
        accurancy_validation = np.mean(y_val == prediccionesValidacion)

        results[(i, j)] = (accurancy_train, accurancy_validation)

        if accurancy_validation > best_val:
            best_val = accurancy_validation
            best_softmax = modelo
################################################################################
#                              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)
```

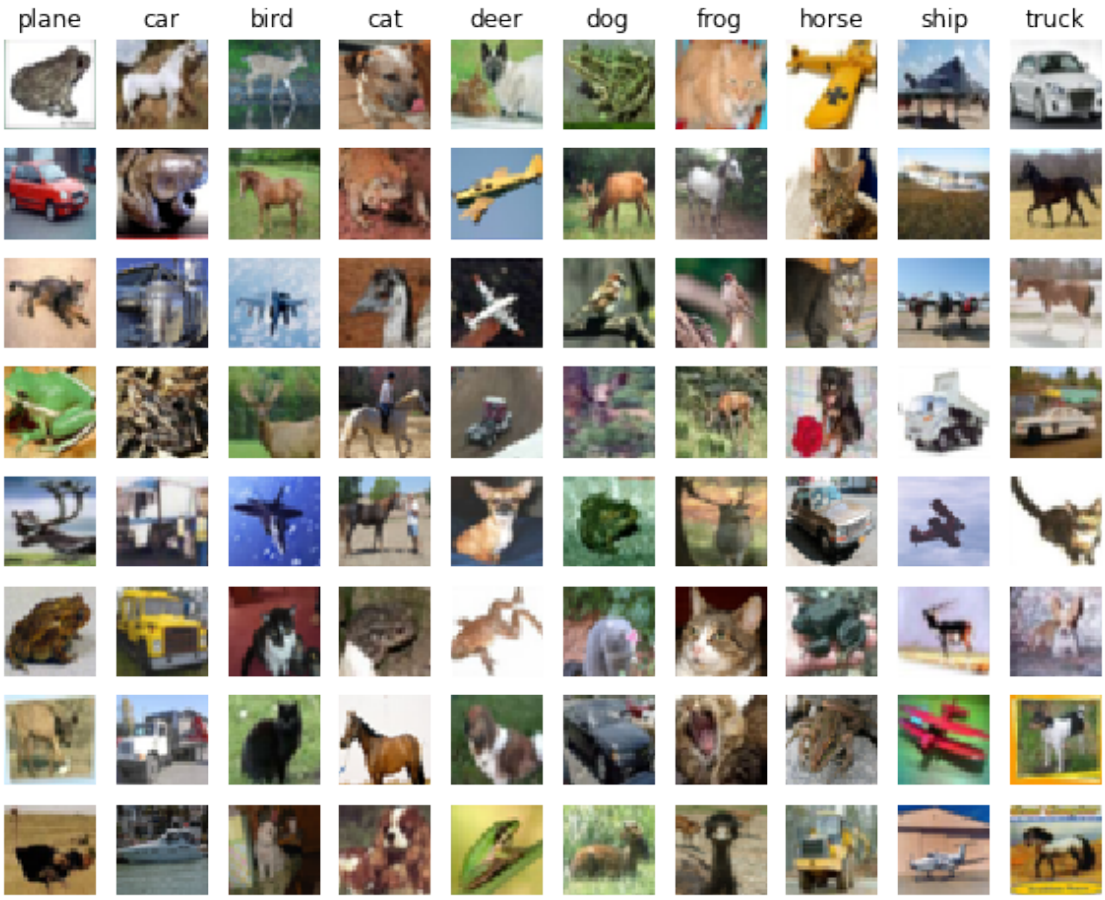
```
lr 1.000000e-09 reg 1.000000e+05 train accuracy: 0.081429 val accuracy: 0.098000
lr 1.000000e-09 reg 1.000000e+06 train accuracy: 0.081449 val accuracy: 0.098000
lr 1.000000e-09 reg 1.000000e+07 train accuracy: 0.413367 val accuracy: 0.417000
lr 1.000000e-08 reg 1.000000e+05 train accuracy: 0.415857 val accuracy: 0.413000
lr 1.000000e-08 reg 1.000000e+06 train accuracy: 0.413510 val accuracy: 0.424000
lr 1.000000e-08 reg 1.000000e+07 train accuracy: 0.407714 val accuracy: 0.407000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.411959 val accuracy: 0.424000
lr 1.000000e-07 reg 1.000000e+06 train accuracy: 0.391265 val accuracy: 0.380000
lr 1.000000e-07 reg 1.000000e+07 train accuracy: 0.294204 val accuracy: 0.313000
```

```
best validation accuracy achieved during cross-validation: 0.424000
```

[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)
```

```
0.315
```

[15]:
```
# 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 :

Algunas de las imágenes mal clasificadas para una determinada categoría comparten colores similares o fondos similares.

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.

[8]:
```
# 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)
```

[14]:
```
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)*****

results = {}
best_val = -1

hidden_sizes = [100, 500, 1500]
num_classes = 10
learning_rates = [1.0, 0.001]
learning_rate_decay = [0.95, 0.90]
regularization_strengths = [0.001,0.01, 0.1]


for i in learning_rates:
    for j in regularization_strengths:
            for l in learning_rate_decay:
                for m in hidden_sizes:
                    net = TwoLayerNet(input_dim, m, num_classes)
                    # Train the network
                    stats = net.train(X_train_feats, y_train, X_val_feats,␣
 ↪y_val, num_iters=2000, batch_size=200, learning_rate=i, learning_rate_decay=␣
 ↪l, reg=j, verbose=True)
```

```
                    prediccionesTrain = net.predict(X_train_feats)
                    prediccionesValidacion = net.predict(X_val_feats)

                    accurancy_train = np.mean(y_train == prediccionesTrain)
                    accurancy_validation = np.mean(y_val ==␣
↪prediccionesValidacion)

                    results[(i, j)] = (accurancy_train, accurancy_validation)

                    if accurancy_validation > best_val:
                        best_val = accurancy_validation
                        best_net = net

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

```
iteration 0 / 2000: loss 2.302585
iteration 100 / 2000: loss 1.594095
iteration 200 / 2000: loss 1.319682
iteration 300 / 2000: loss 1.330499
iteration 400 / 2000: loss 1.180809
iteration 500 / 2000: loss 1.322113
iteration 600 / 2000: loss 1.337980
iteration 700 / 2000: loss 1.182316
iteration 800 / 2000: loss 1.250871
iteration 900 / 2000: loss 1.081953
iteration 1000 / 2000: loss 1.112479
iteration 1100 / 2000: loss 1.133966
iteration 1200 / 2000: loss 1.089409
iteration 1300 / 2000: loss 1.083531
iteration 1400 / 2000: loss 1.238367
iteration 1500 / 2000: loss 1.151462
iteration 1600 / 2000: loss 1.072772
iteration 1700 / 2000: loss 1.125352
iteration 1800 / 2000: loss 1.205759
iteration 1900 / 2000: loss 1.018749
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 1.500098
iteration 200 / 2000: loss 1.407162
iteration 300 / 2000: loss 1.352567
iteration 400 / 2000: loss 1.218611
iteration 500 / 2000: loss 1.128755
iteration 600 / 2000: loss 1.173262
iteration 700 / 2000: loss 1.112104
iteration 800 / 2000: loss 1.146828
iteration 900 / 2000: loss 1.148005
iteration 1000 / 2000: loss 1.052561
```

```
iteration 1100 / 2000: loss 1.232930
iteration 1200 / 2000: loss 1.079229
iteration 1300 / 2000: loss 1.110201
iteration 1400 / 2000: loss 1.176803
iteration 1500 / 2000: loss 0.970384
iteration 1600 / 2000: loss 0.979806
iteration 1700 / 2000: loss 1.051372
iteration 1800 / 2000: loss 1.050216
iteration 1900 / 2000: loss 1.039516
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 1.455561
iteration 200 / 2000: loss 1.374139
iteration 300 / 2000: loss 1.443867
iteration 400 / 2000: loss 1.310661
iteration 500 / 2000: loss 1.202844
iteration 600 / 2000: loss 1.332734
iteration 700 / 2000: loss 1.037384
iteration 800 / 2000: loss 1.093347
iteration 900 / 2000: loss 1.150489
iteration 1000 / 2000: loss 1.188897
iteration 1100 / 2000: loss 1.098955
iteration 1200 / 2000: loss 1.222502
iteration 1300 / 2000: loss 1.172052
iteration 1400 / 2000: loss 1.077102
iteration 1500 / 2000: loss 1.157958
iteration 1600 / 2000: loss 1.023222
iteration 1700 / 2000: loss 1.070654
iteration 1800 / 2000: loss 0.971836
iteration 1900 / 2000: loss 0.895799
iteration 0 / 2000: loss 2.302585
iteration 100 / 2000: loss 1.548917
iteration 200 / 2000: loss 1.345128
iteration 300 / 2000: loss 1.260652
iteration 400 / 2000: loss 1.207966
iteration 500 / 2000: loss 1.321213
iteration 600 / 2000: loss 1.232447
iteration 700 / 2000: loss 1.291392
iteration 800 / 2000: loss 1.160774
iteration 900 / 2000: loss 1.158706
iteration 1000 / 2000: loss 1.277868
iteration 1100 / 2000: loss 1.104657
iteration 1200 / 2000: loss 1.098066
iteration 1300 / 2000: loss 1.206643
iteration 1400 / 2000: loss 1.238818
iteration 1500 / 2000: loss 1.273333
iteration 1600 / 2000: loss 0.980040
iteration 1700 / 2000: loss 1.149733
iteration 1800 / 2000: loss 1.047433
```

```
iteration 1900 / 2000: loss 1.007124
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 1.501713
iteration 200 / 2000: loss 1.412421
iteration 300 / 2000: loss 1.399428
iteration 400 / 2000: loss 1.252926
iteration 500 / 2000: loss 1.174830
iteration 600 / 2000: loss 1.225055
iteration 700 / 2000: loss 1.219286
iteration 800 / 2000: loss 1.106991
iteration 900 / 2000: loss 1.091808
iteration 1000 / 2000: loss 1.199259
iteration 1100 / 2000: loss 1.007512
iteration 1200 / 2000: loss 1.035636
iteration 1300 / 2000: loss 1.169394
iteration 1400 / 2000: loss 0.997379
iteration 1500 / 2000: loss 1.111152
iteration 1600 / 2000: loss 1.001614
iteration 1700 / 2000: loss 1.086847
iteration 1800 / 2000: loss 0.970292
iteration 1900 / 2000: loss 0.980488
iteration 0 / 2000: loss 2.302587
iteration 100 / 2000: loss 1.429326
iteration 200 / 2000: loss 1.486555
iteration 300 / 2000: loss 1.314358
iteration 400 / 2000: loss 1.317811
iteration 500 / 2000: loss 1.151657
iteration 600 / 2000: loss 1.077906
iteration 700 / 2000: loss 1.179777
iteration 800 / 2000: loss 1.177926
iteration 900 / 2000: loss 1.239934
iteration 1000 / 2000: loss 1.009088
iteration 1100 / 2000: loss 0.986343
iteration 1200 / 2000: loss 1.000127
iteration 1300 / 2000: loss 0.992884
iteration 1400 / 2000: loss 1.063573
iteration 1500 / 2000: loss 1.100285
iteration 1600 / 2000: loss 0.845145
iteration 1700 / 2000: loss 1.189986
iteration 1800 / 2000: loss 0.941449
iteration 1900 / 2000: loss 0.968299
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 1.527152
iteration 200 / 2000: loss 1.638221
iteration 300 / 2000: loss 1.609925
iteration 400 / 2000: loss 1.608385
iteration 500 / 2000: loss 1.675178
iteration 600 / 2000: loss 1.475267
```

```
iteration 700 / 2000: loss 1.595601
iteration 800 / 2000: loss 1.536235
iteration 900 / 2000: loss 1.418021
iteration 1000 / 2000: loss 1.558502
iteration 1100 / 2000: loss 1.569961
iteration 1200 / 2000: loss 1.506768
iteration 1300 / 2000: loss 1.540763
iteration 1400 / 2000: loss 1.501186
iteration 1500 / 2000: loss 1.491476
iteration 1600 / 2000: loss 1.457507
iteration 1700 / 2000: loss 1.578162
iteration 1800 / 2000: loss 1.519565
iteration 1900 / 2000: loss 1.694071
iteration 0 / 2000: loss 2.302589
iteration 100 / 2000: loss 1.695344
iteration 200 / 2000: loss 1.494289
iteration 300 / 2000: loss 1.573029
iteration 400 / 2000: loss 1.687618
iteration 500 / 2000: loss 1.553333
iteration 600 / 2000: loss 1.552902
iteration 700 / 2000: loss 1.593451
iteration 800 / 2000: loss 1.542792
iteration 900 / 2000: loss 1.502627
iteration 1000 / 2000: loss 1.575454
iteration 1100 / 2000: loss 1.478252
iteration 1200 / 2000: loss 1.726128
iteration 1300 / 2000: loss 1.513227
iteration 1400 / 2000: loss 1.487055
iteration 1500 / 2000: loss 1.472936
iteration 1600 / 2000: loss 1.479030
iteration 1700 / 2000: loss 1.485005
iteration 1800 / 2000: loss 1.513594
iteration 1900 / 2000: loss 1.512913
iteration 0 / 2000: loss 2.302597
iteration 100 / 2000: loss 1.701214
iteration 200 / 2000: loss 1.566827
iteration 300 / 2000: loss 1.516775
iteration 400 / 2000: loss 1.502525
iteration 500 / 2000: loss 1.457378
iteration 600 / 2000: loss 1.568149
iteration 700 / 2000: loss 1.511786
iteration 800 / 2000: loss 1.529483
iteration 900 / 2000: loss 1.615509
iteration 1000 / 2000: loss 1.481321
iteration 1100 / 2000: loss 1.486413
iteration 1200 / 2000: loss 1.699685
iteration 1300 / 2000: loss 1.588727
iteration 1400 / 2000: loss 1.398008
```

```
iteration 1500 / 2000: loss 1.535601
iteration 1600 / 2000: loss 1.597931
iteration 1700 / 2000: loss 1.555523
iteration 1800 / 2000: loss 1.362057
iteration 1900 / 2000: loss 1.446846
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 1.633558
iteration 200 / 2000: loss 1.565330
iteration 300 / 2000: loss 1.554808
iteration 400 / 2000: loss 1.592494
iteration 500 / 2000: loss 1.396563
iteration 600 / 2000: loss 1.673161
iteration 700 / 2000: loss 1.596976
iteration 800 / 2000: loss 1.402896
iteration 900 / 2000: loss 1.489500
iteration 1000 / 2000: loss 1.521294
iteration 1100 / 2000: loss 1.590885
iteration 1200 / 2000: loss 1.457180
iteration 1300 / 2000: loss 1.593633
iteration 1400 / 2000: loss 1.476995
iteration 1500 / 2000: loss 1.402411
iteration 1600 / 2000: loss 1.478511
iteration 1700 / 2000: loss 1.456848
iteration 1800 / 2000: loss 1.544008
iteration 1900 / 2000: loss 1.469709
iteration 0 / 2000: loss 2.302589
iteration 100 / 2000: loss 1.709936
iteration 200 / 2000: loss 1.688606
iteration 300 / 2000: loss 1.592377
iteration 400 / 2000: loss 1.458271
iteration 500 / 2000: loss 1.613887
iteration 600 / 2000: loss 1.563234
iteration 700 / 2000: loss 1.394473
iteration 800 / 2000: loss 1.533512
iteration 900 / 2000: loss 1.626786
iteration 1000 / 2000: loss 1.532540
iteration 1100 / 2000: loss 1.460074
iteration 1200 / 2000: loss 1.573920
iteration 1300 / 2000: loss 1.473331
iteration 1400 / 2000: loss 1.478597
iteration 1500 / 2000: loss 1.442866
iteration 1600 / 2000: loss 1.560126
iteration 1700 / 2000: loss 1.367486
iteration 1800 / 2000: loss 1.444156
iteration 1900 / 2000: loss 1.401491
iteration 0 / 2000: loss 2.302597
iteration 100 / 2000: loss 1.568193
iteration 200 / 2000: loss 1.588568
```

```
iteration 300 / 2000: loss 1.648481
iteration 400 / 2000: loss 1.587336
iteration 500 / 2000: loss 1.402979
iteration 600 / 2000: loss 1.615876
iteration 700 / 2000: loss 1.523931
iteration 800 / 2000: loss 1.509974
iteration 900 / 2000: loss 1.565609
iteration 1000 / 2000: loss 1.603817
iteration 1100 / 2000: loss 1.598388
iteration 1200 / 2000: loss 1.498348
iteration 1300 / 2000: loss 1.430763
iteration 1400 / 2000: loss 1.480830
iteration 1500 / 2000: loss 1.530409
iteration 1600 / 2000: loss 1.369012
iteration 1700 / 2000: loss 1.412750
iteration 1800 / 2000: loss 1.419303
iteration 1900 / 2000: loss 1.496891
iteration 0 / 2000: loss 2.302593
iteration 100 / 2000: loss 2.032063
iteration 200 / 2000: loss 2.003798
iteration 300 / 2000: loss 2.059112
iteration 400 / 2000: loss 2.063692
iteration 500 / 2000: loss 1.973652
iteration 600 / 2000: loss 1.972341
iteration 700 / 2000: loss 1.922593
iteration 800 / 2000: loss 2.013016
iteration 900 / 2000: loss 2.127975
iteration 1000 / 2000: loss 1.923780
iteration 1100 / 2000: loss 1.996703
iteration 1200 / 2000: loss 2.034242
iteration 1300 / 2000: loss 2.082058
iteration 1400 / 2000: loss 1.921790
iteration 1500 / 2000: loss 1.949842
iteration 1600 / 2000: loss 1.985932
iteration 1700 / 2000: loss 2.015440
iteration 1800 / 2000: loss 2.049519
iteration 1900 / 2000: loss 1.978433
iteration 0 / 2000: loss 2.302626
iteration 100 / 2000: loss 2.047506
iteration 200 / 2000: loss 2.006311
iteration 300 / 2000: loss 2.104067
iteration 400 / 2000: loss 2.024046
iteration 500 / 2000: loss 2.041884
iteration 600 / 2000: loss 2.040862
iteration 700 / 2000: loss 2.038583
iteration 800 / 2000: loss 1.937999
iteration 900 / 2000: loss 1.974519
iteration 1000 / 2000: loss 2.057323
```

```
iteration 1100 / 2000: loss 2.041272
iteration 1200 / 2000: loss 2.013146
iteration 1300 / 2000: loss 2.066549
iteration 1400 / 2000: loss 1.977177
iteration 1500 / 2000: loss 2.042154
iteration 1600 / 2000: loss 1.973227
iteration 1700 / 2000: loss 1.996339
iteration 1800 / 2000: loss 1.989787
iteration 1900 / 2000: loss 1.929979
iteration 0 / 2000: loss 2.302709
iteration 100 / 2000: loss 2.058607
iteration 200 / 2000: loss 2.101922
iteration 300 / 2000: loss 2.067820
iteration 400 / 2000: loss 2.019848
iteration 500 / 2000: loss 2.029207
iteration 600 / 2000: loss 2.046820
iteration 700 / 2000: loss 2.064479
iteration 800 / 2000: loss 2.030643
iteration 900 / 2000: loss 1.926744
iteration 1000 / 2000: loss 2.055342
iteration 1100 / 2000: loss 2.028316
iteration 1200 / 2000: loss 2.064307
iteration 1300 / 2000: loss 1.951410
iteration 1400 / 2000: loss 2.045541
iteration 1500 / 2000: loss 1.987181
iteration 1600 / 2000: loss 2.002051
iteration 1700 / 2000: loss 2.087262
iteration 1800 / 2000: loss 1.976477
iteration 1900 / 2000: loss 1.966825
iteration 0 / 2000: loss 2.302594
iteration 100 / 2000: loss 2.040548
iteration 200 / 2000: loss 1.975489
iteration 300 / 2000: loss 1.961792
iteration 400 / 2000: loss 2.017594
iteration 500 / 2000: loss 1.983897
iteration 600 / 2000: loss 2.093269
iteration 700 / 2000: loss 2.072599
iteration 800 / 2000: loss 2.021475
iteration 900 / 2000: loss 2.045098
iteration 1000 / 2000: loss 1.935592
iteration 1100 / 2000: loss 1.987170
iteration 1200 / 2000: loss 2.003551
iteration 1300 / 2000: loss 1.875396
iteration 1400 / 2000: loss 2.137768
iteration 1500 / 2000: loss 2.003221
iteration 1600 / 2000: loss 2.017448
iteration 1700 / 2000: loss 2.001868
iteration 1800 / 2000: loss 2.064519
```

```
iteration 1900 / 2000: loss 1.962909
iteration 0 / 2000: loss 2.302626
iteration 100 / 2000: loss 2.050206
iteration 200 / 2000: loss 2.042782
iteration 300 / 2000: loss 1.967822
iteration 400 / 2000: loss 2.061283
iteration 500 / 2000: loss 2.001167
iteration 600 / 2000: loss 2.095020
iteration 700 / 2000: loss 2.033245
iteration 800 / 2000: loss 1.962740
iteration 900 / 2000: loss 2.043260
iteration 1000 / 2000: loss 2.032410
iteration 1100 / 2000: loss 2.002650
iteration 1200 / 2000: loss 2.043144
iteration 1300 / 2000: loss 2.040607
iteration 1400 / 2000: loss 1.994280
iteration 1500 / 2000: loss 2.033936
iteration 1600 / 2000: loss 1.999925
iteration 1700 / 2000: loss 1.964221
iteration 1800 / 2000: loss 1.966402
iteration 1900 / 2000: loss 1.998836
iteration 0 / 2000: loss 2.302708
iteration 100 / 2000: loss 2.021867
iteration 200 / 2000: loss 2.109879
iteration 300 / 2000: loss 2.115465
iteration 400 / 2000: loss 2.055408
iteration 500 / 2000: loss 2.030922
iteration 600 / 2000: loss 2.061638
iteration 700 / 2000: loss 1.970454
iteration 800 / 2000: loss 1.989540
iteration 900 / 2000: loss 1.970189
iteration 1000 / 2000: loss 2.041396
iteration 1100 / 2000: loss 1.993972
iteration 1200 / 2000: loss 2.044402
iteration 1300 / 2000: loss 1.948542
iteration 1400 / 2000: loss 2.041466
iteration 1500 / 2000: loss 1.983767
iteration 1600 / 2000: loss 2.032241
iteration 1700 / 2000: loss 2.013086
iteration 1800 / 2000: loss 2.014254
iteration 1900 / 2000: loss 1.959954
iteration 0 / 2000: loss 2.302585
iteration 100 / 2000: loss 2.302592
iteration 200 / 2000: loss 2.302632
iteration 300 / 2000: loss 2.302573
iteration 400 / 2000: loss 2.302517
iteration 500 / 2000: loss 2.302641
iteration 600 / 2000: loss 2.302654
```

```
iteration 700 / 2000: loss 2.302602
iteration 800 / 2000: loss 2.302566
iteration 900 / 2000: loss 2.302653
iteration 1000 / 2000: loss 2.302663
iteration 1100 / 2000: loss 2.302596
iteration 1200 / 2000: loss 2.302620
iteration 1300 / 2000: loss 2.302594
iteration 1400 / 2000: loss 2.302599
iteration 1500 / 2000: loss 2.302583
iteration 1600 / 2000: loss 2.302592
iteration 1700 / 2000: loss 2.302497
iteration 1800 / 2000: loss 2.302555
iteration 1900 / 2000: loss 2.302601
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 2.302587
iteration 200 / 2000: loss 2.302607
iteration 300 / 2000: loss 2.302614
iteration 400 / 2000: loss 2.302590
iteration 500 / 2000: loss 2.302563
iteration 600 / 2000: loss 2.302572
iteration 700 / 2000: loss 2.302504
iteration 800 / 2000: loss 2.302540
iteration 900 / 2000: loss 2.302616
iteration 1000 / 2000: loss 2.302630
iteration 1100 / 2000: loss 2.302613
iteration 1200 / 2000: loss 2.302611
iteration 1300 / 2000: loss 2.302591
iteration 1400 / 2000: loss 2.302541
iteration 1500 / 2000: loss 2.302585
iteration 1600 / 2000: loss 2.302580
iteration 1700 / 2000: loss 2.302603
iteration 1800 / 2000: loss 2.302678
iteration 1900 / 2000: loss 2.302594
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 2.302587
iteration 200 / 2000: loss 2.302559
iteration 300 / 2000: loss 2.302572
iteration 400 / 2000: loss 2.302590
iteration 500 / 2000: loss 2.302625
iteration 600 / 2000: loss 2.302600
iteration 700 / 2000: loss 2.302609
iteration 800 / 2000: loss 2.302551
iteration 900 / 2000: loss 2.302565
iteration 1000 / 2000: loss 2.302585
iteration 1100 / 2000: loss 2.302628
iteration 1200 / 2000: loss 2.302638
iteration 1300 / 2000: loss 2.302601
iteration 1400 / 2000: loss 2.302533
```

```
iteration 1500 / 2000: loss 2.302560
iteration 1600 / 2000: loss 2.302602
iteration 1700 / 2000: loss 2.302537
iteration 1800 / 2000: loss 2.302530
iteration 1900 / 2000: loss 2.302540
iteration 0 / 2000: loss 2.302585
iteration 100 / 2000: loss 2.302587
iteration 200 / 2000: loss 2.302578
iteration 300 / 2000: loss 2.302593
iteration 400 / 2000: loss 2.302628
iteration 500 / 2000: loss 2.302602
iteration 600 / 2000: loss 2.302627
iteration 700 / 2000: loss 2.302596
iteration 800 / 2000: loss 2.302594
iteration 900 / 2000: loss 2.302629
iteration 1000 / 2000: loss 2.302538
iteration 1100 / 2000: loss 2.302511
iteration 1200 / 2000: loss 2.302516
iteration 1300 / 2000: loss 2.302578
iteration 1400 / 2000: loss 2.302580
iteration 1500 / 2000: loss 2.302600
iteration 1600 / 2000: loss 2.302627
iteration 1700 / 2000: loss 2.302598
iteration 1800 / 2000: loss 2.302535
iteration 1900 / 2000: loss 2.302581
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 2.302600
iteration 200 / 2000: loss 2.302597
iteration 300 / 2000: loss 2.302662
iteration 400 / 2000: loss 2.302532
iteration 500 / 2000: loss 2.302513
iteration 600 / 2000: loss 2.302502
iteration 700 / 2000: loss 2.302527
iteration 800 / 2000: loss 2.302547
iteration 900 / 2000: loss 2.302618
iteration 1000 / 2000: loss 2.302580
iteration 1100 / 2000: loss 2.302606
iteration 1200 / 2000: loss 2.302602
iteration 1300 / 2000: loss 2.302579
iteration 1400 / 2000: loss 2.302594
iteration 1500 / 2000: loss 2.302547
iteration 1600 / 2000: loss 2.302596
iteration 1700 / 2000: loss 2.302614
iteration 1800 / 2000: loss 2.302597
iteration 1900 / 2000: loss 2.302582
iteration 0 / 2000: loss 2.302587
iteration 100 / 2000: loss 2.302580
iteration 200 / 2000: loss 2.302615
```

```
iteration 300 / 2000: loss 2.302575
iteration 400 / 2000: loss 2.302586
iteration 500 / 2000: loss 2.302568
iteration 600 / 2000: loss 2.302559
iteration 700 / 2000: loss 2.302582
iteration 800 / 2000: loss 2.302621
iteration 900 / 2000: loss 2.302544
iteration 1000 / 2000: loss 2.302599
iteration 1100 / 2000: loss 2.302592
iteration 1200 / 2000: loss 2.302568
iteration 1300 / 2000: loss 2.302635
iteration 1400 / 2000: loss 2.302502
iteration 1500 / 2000: loss 2.302552
iteration 1600 / 2000: loss 2.302551
iteration 1700 / 2000: loss 2.302563
iteration 1800 / 2000: loss 2.302579
iteration 1900 / 2000: loss 2.302545
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 2.302571
iteration 200 / 2000: loss 2.302632
iteration 300 / 2000: loss 2.302592
iteration 400 / 2000: loss 2.302549
iteration 500 / 2000: loss 2.302599
iteration 600 / 2000: loss 2.302694
iteration 700 / 2000: loss 2.302636
iteration 800 / 2000: loss 2.302596
iteration 900 / 2000: loss 2.302559
iteration 1000 / 2000: loss 2.302574
iteration 1100 / 2000: loss 2.302508
iteration 1200 / 2000: loss 2.302568
iteration 1300 / 2000: loss 2.302609
iteration 1400 / 2000: loss 2.302528
iteration 1500 / 2000: loss 2.302586
iteration 1600 / 2000: loss 2.302610
iteration 1700 / 2000: loss 2.302608
iteration 1800 / 2000: loss 2.302538
iteration 1900 / 2000: loss 2.302590
iteration 0 / 2000: loss 2.302589
iteration 100 / 2000: loss 2.302600
iteration 200 / 2000: loss 2.302567
iteration 300 / 2000: loss 2.302568
iteration 400 / 2000: loss 2.302600
iteration 500 / 2000: loss 2.302545
iteration 600 / 2000: loss 2.302584
iteration 700 / 2000: loss 2.302563
iteration 800 / 2000: loss 2.302573
iteration 900 / 2000: loss 2.302607
iteration 1000 / 2000: loss 2.302517
```

```
iteration 1100 / 2000: loss 2.302605
iteration 1200 / 2000: loss 2.302569
iteration 1300 / 2000: loss 2.302611
iteration 1400 / 2000: loss 2.302680
iteration 1500 / 2000: loss 2.302609
iteration 1600 / 2000: loss 2.302545
iteration 1700 / 2000: loss 2.302626
iteration 1800 / 2000: loss 2.302592
iteration 1900 / 2000: loss 2.302580
iteration 0 / 2000: loss 2.302597
iteration 100 / 2000: loss 2.302620
iteration 200 / 2000: loss 2.302600
iteration 300 / 2000: loss 2.302674
iteration 400 / 2000: loss 2.302601
iteration 500 / 2000: loss 2.302607
iteration 600 / 2000: loss 2.302621
iteration 700 / 2000: loss 2.302609
iteration 800 / 2000: loss 2.302598
iteration 900 / 2000: loss 2.302550
iteration 1000 / 2000: loss 2.302564
iteration 1100 / 2000: loss 2.302588
iteration 1200 / 2000: loss 2.302639
iteration 1300 / 2000: loss 2.302653
iteration 1400 / 2000: loss 2.302545
iteration 1500 / 2000: loss 2.302636
iteration 1600 / 2000: loss 2.302685
iteration 1700 / 2000: loss 2.302691
iteration 1800 / 2000: loss 2.302654
iteration 1900 / 2000: loss 2.302523
iteration 0 / 2000: loss 2.302586
iteration 100 / 2000: loss 2.302593
iteration 200 / 2000: loss 2.302598
iteration 300 / 2000: loss 2.302600
iteration 400 / 2000: loss 2.302599
iteration 500 / 2000: loss 2.302581
iteration 600 / 2000: loss 2.302567
iteration 700 / 2000: loss 2.302603
iteration 800 / 2000: loss 2.302571
iteration 900 / 2000: loss 2.302575
iteration 1000 / 2000: loss 2.302695
iteration 1100 / 2000: loss 2.302602
iteration 1200 / 2000: loss 2.302558
iteration 1300 / 2000: loss 2.302578
iteration 1400 / 2000: loss 2.302601
iteration 1500 / 2000: loss 2.302602
iteration 1600 / 2000: loss 2.302583
iteration 1700 / 2000: loss 2.302584
iteration 1800 / 2000: loss 2.302638
```

```
iteration 1900 / 2000: loss 2.302610
iteration 0 / 2000: loss 2.302589
iteration 100 / 2000: loss 2.302579
iteration 200 / 2000: loss 2.302589
iteration 300 / 2000: loss 2.302593
iteration 400 / 2000: loss 2.302585
iteration 500 / 2000: loss 2.302587
iteration 600 / 2000: loss 2.302635
iteration 700 / 2000: loss 2.302604
iteration 800 / 2000: loss 2.302623
iteration 900 / 2000: loss 2.302527
iteration 1000 / 2000: loss 2.302584
iteration 1100 / 2000: loss 2.302547
iteration 1200 / 2000: loss 2.302591
iteration 1300 / 2000: loss 2.302617
iteration 1400 / 2000: loss 2.302586
iteration 1500 / 2000: loss 2.302585
iteration 1600 / 2000: loss 2.302550
iteration 1700 / 2000: loss 2.302617
iteration 1800 / 2000: loss 2.302570
iteration 1900 / 2000: loss 2.302530
iteration 0 / 2000: loss 2.302598
iteration 100 / 2000: loss 2.302595
iteration 200 / 2000: loss 2.302595
iteration 300 / 2000: loss 2.302608
iteration 400 / 2000: loss 2.302613
iteration 500 / 2000: loss 2.302600
iteration 600 / 2000: loss 2.302615
iteration 700 / 2000: loss 2.302587
iteration 800 / 2000: loss 2.302597
iteration 900 / 2000: loss 2.302609
iteration 1000 / 2000: loss 2.302641
iteration 1100 / 2000: loss 2.302532
iteration 1200 / 2000: loss 2.302580
iteration 1300 / 2000: loss 2.302654
iteration 1400 / 2000: loss 2.302595
iteration 1500 / 2000: loss 2.302666
iteration 1600 / 2000: loss 2.302582
iteration 1700 / 2000: loss 2.302595
iteration 1800 / 2000: loss 2.302575
iteration 1900 / 2000: loss 2.302540
iteration 0 / 2000: loss 2.302593
iteration 100 / 2000: loss 2.302596
iteration 200 / 2000: loss 2.302576
iteration 300 / 2000: loss 2.302603
iteration 400 / 2000: loss 2.302639
iteration 500 / 2000: loss 2.302598
iteration 600 / 2000: loss 2.302585
```

```
iteration 700 / 2000: loss 2.302537
iteration 800 / 2000: loss 2.302633
iteration 900 / 2000: loss 2.302589
iteration 1000 / 2000: loss 2.302537
iteration 1100 / 2000: loss 2.302615
iteration 1200 / 2000: loss 2.302620
iteration 1300 / 2000: loss 2.302554
iteration 1400 / 2000: loss 2.302502
iteration 1500 / 2000: loss 2.302611
iteration 1600 / 2000: loss 2.302561
iteration 1700 / 2000: loss 2.302687
iteration 1800 / 2000: loss 2.302583
iteration 1900 / 2000: loss 2.302480
iteration 0 / 2000: loss 2.302626
iteration 100 / 2000: loss 2.302624
iteration 200 / 2000: loss 2.302636
iteration 300 / 2000: loss 2.302583
iteration 400 / 2000: loss 2.302645
iteration 500 / 2000: loss 2.302570
iteration 600 / 2000: loss 2.302625
iteration 700 / 2000: loss 2.302496
iteration 800 / 2000: loss 2.302606
iteration 900 / 2000: loss 2.302400
iteration 1000 / 2000: loss 2.302535
iteration 1100 / 2000: loss 2.302694
iteration 1200 / 2000: loss 2.302724
iteration 1300 / 2000: loss 2.302538
iteration 1400 / 2000: loss 2.302679
iteration 1500 / 2000: loss 2.302644
iteration 1600 / 2000: loss 2.302497
iteration 1700 / 2000: loss 2.302531
iteration 1800 / 2000: loss 2.302628
iteration 1900 / 2000: loss 2.302548
iteration 0 / 2000: loss 2.302708
iteration 100 / 2000: loss 2.302726
iteration 200 / 2000: loss 2.302711
iteration 300 / 2000: loss 2.302704
iteration 400 / 2000: loss 2.302707
iteration 500 / 2000: loss 2.302741
iteration 600 / 2000: loss 2.302681
iteration 700 / 2000: loss 2.302715
iteration 800 / 2000: loss 2.302661
iteration 900 / 2000: loss 2.302590
iteration 1000 / 2000: loss 2.302659
iteration 1100 / 2000: loss 2.302703
iteration 1200 / 2000: loss 2.302714
iteration 1300 / 2000: loss 2.302633
iteration 1400 / 2000: loss 2.302741
```

```
iteration 1500 / 2000: loss 2.302672
iteration 1600 / 2000: loss 2.302708
iteration 1700 / 2000: loss 2.302629
iteration 1800 / 2000: loss 2.302672
iteration 1900 / 2000: loss 2.302662
iteration 0 / 2000: loss 2.302593
iteration 100 / 2000: loss 2.302596
iteration 200 / 2000: loss 2.302584
iteration 300 / 2000: loss 2.302628
iteration 400 / 2000: loss 2.302614
iteration 500 / 2000: loss 2.302591
iteration 600 / 2000: loss 2.302599
iteration 700 / 2000: loss 2.302662
iteration 800 / 2000: loss 2.302590
iteration 900 / 2000: loss 2.302630
iteration 1000 / 2000: loss 2.302560
iteration 1100 / 2000: loss 2.302602
iteration 1200 / 2000: loss 2.302554
iteration 1300 / 2000: loss 2.302614
iteration 1400 / 2000: loss 2.302581
iteration 1500 / 2000: loss 2.302558
iteration 1600 / 2000: loss 2.302619
iteration 1700 / 2000: loss 2.302516
iteration 1800 / 2000: loss 2.302662
iteration 1900 / 2000: loss 2.302627
iteration 0 / 2000: loss 2.302626
iteration 100 / 2000: loss 2.302631
iteration 200 / 2000: loss 2.302668
iteration 300 / 2000: loss 2.302639
iteration 400 / 2000: loss 2.302604
iteration 500 / 2000: loss 2.302613
iteration 600 / 2000: loss 2.302586
iteration 700 / 2000: loss 2.302621
iteration 800 / 2000: loss 2.302657
iteration 900 / 2000: loss 2.302578
iteration 1000 / 2000: loss 2.302640
iteration 1100 / 2000: loss 2.302613
iteration 1200 / 2000: loss 2.302603
iteration 1300 / 2000: loss 2.302600
iteration 1400 / 2000: loss 2.302665
iteration 1500 / 2000: loss 2.302606
iteration 1600 / 2000: loss 2.302603
iteration 1700 / 2000: loss 2.302630
iteration 1800 / 2000: loss 2.302645
iteration 1900 / 2000: loss 2.302615
iteration 0 / 2000: loss 2.302708
iteration 100 / 2000: loss 2.302705
iteration 200 / 2000: loss 2.302731
```

```
iteration 300 / 2000: loss 2.302714
iteration 400 / 2000: loss 2.302685
iteration 500 / 2000: loss 2.302689
iteration 600 / 2000: loss 2.302651
iteration 700 / 2000: loss 2.302724
iteration 800 / 2000: loss 2.302703
iteration 900 / 2000: loss 2.302717
iteration 1000 / 2000: loss 2.302671
iteration 1100 / 2000: loss 2.302645
iteration 1200 / 2000: loss 2.302665
iteration 1300 / 2000: loss 2.302651
iteration 1400 / 2000: loss 2.302671
iteration 1500 / 2000: loss 2.302668
iteration 1600 / 2000: loss 2.302680
iteration 1700 / 2000: loss 2.302666
iteration 1800 / 2000: loss 2.302705
iteration 1900 / 2000: loss 2.302669
```

[16]:
```
# 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)
```

```
0.578
```

---

2 IMPORTANT

This is the end of this question. Please do the following:

1. Click `File -> Save` to make sure the latest checkpoint of this notebook is saved to your Drive.
2. Execute the cell below to download the modified `.py` files back to your drive.

[17]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = []

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
 ↪as f:
    f.write(''.join(open(files).readlines()))
```