knn

October 2, 2025

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

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-10-01 11:55:45--  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  49.1MB/s    in 3.5s

2025-10-01 11:55:49 (46.0 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
--2025-10-01 11:55:53--  http://cs231n.stanford.edu/imagenet_val_25.npz
Resolving cs231n.stanford.edu (cs231n.stanford.edu)… 171.64.64.64
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.64.64|:80…
connected.
HTTP request sent, awaiting response… 301 Moved Permanently
Location: https://cs231n.stanford.edu/imagenet_val_25.npz [following]
--2025-10-01 11:55:53--  https://cs231n.stanford.edu/imagenet_val_25.npz
Connecting to cs231n.stanford.edu (cs231n.stanford.edu)|171.64.64.64|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 3940548 (3.8M)
Saving to: ‘imagenet_val_25.npz’

imagenet_val_25.npz 100%[===================>]   3.76M  5.93MB/s    in 0.6s

2025-10-01 11:55:54 (5.93 MB/s) - ‘imagenet_val_25.npz’ saved [3940548/3940548]

/content/drive/My Drive/cs231n/assignments/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

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

The kNN classifier consists of two stages:

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

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

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

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

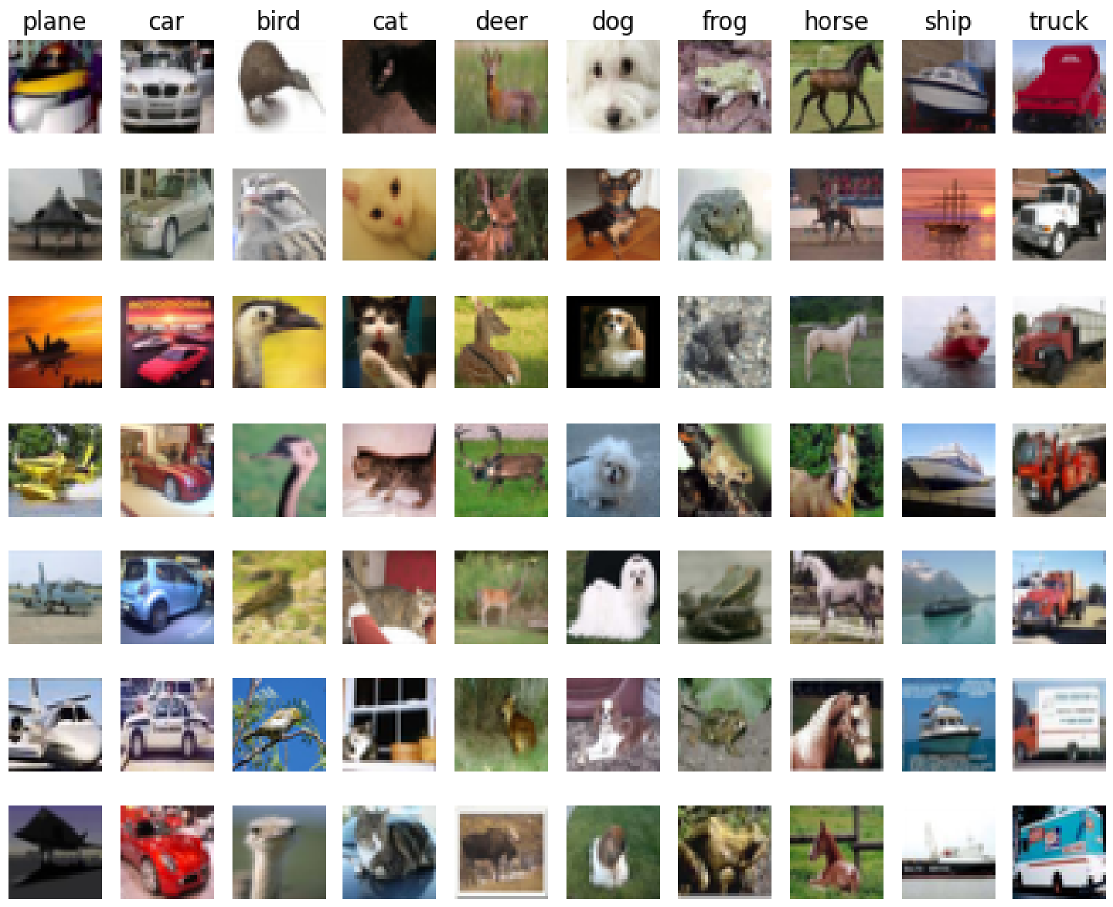
```
# This is a bit of magic to make matplotlib figures appear inline in the␣
 ↪notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

import sys, types, importlib
imp = types.ModuleType("imp")
imp.reload = importlib.reload
sys.modules["imp"] = imp

%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.

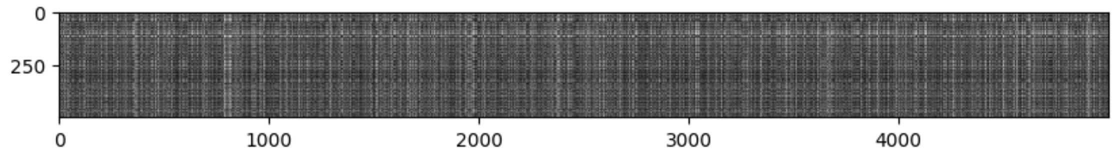
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

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)
```

[ ]:
```python
# 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 visibly brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

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

YourAnswer : *

What in the data is the cause behind the distinctly bright rows?

First, we notice that there are 500 samples in y coordinate and 5000 samples in x coordinate, which we understand that columns are related to a single training image compared to all the test set while rows are related to a single test image compared to all the training set.

A bright row is a result of similarity mismatch between a single test image with almost all the training images. To clarify more, when that test image's pixel vlaues are compared to the pixel values of the all images in the training set (using a chosen distance metric, in our example it's L1, euclidean distance), the calculated distance is consistently high, making the test image likely to be an outlier or asserting it to be a difficult example to classify given the training set. Which in turn,makes the entire row appear bright.

What causes the columns?

As we already discussed above, patterns in the columns relate to specific training images.

A bright column is caused by a training image that is an outlier and is very dissimilar to most of the test images, indicating that this training images is not a good reference for the test set as it is unusual for the whole test set.

[ ]:
```python
# 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. To clarify, both training and test examples are preprocessed in the same way.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

$\mathit{Your Answer:}$

1,2,3 preprocessing steps will not change the performance.

$\mathit{Your Explanation:}$

The performance of a K-NN classifier depends on the relative ordering of distances between data points. If a preprocessing step keeps this order the same for all points, the classifier's predictions won't change. The L1 distance between two images, I_a and I_b, is

$$D_1(I_a, I_b) = \sum_{i,j} \left| p_{ij}^{(a)} - p_{ij}^{(b)} \right|$$

###**CASE BY CASE EXPLANATION:**

1.0.1 1. Mean subtraction

- **Formula Proof**: $\sum_{i,j} \left| \left(p_{ij}^{(a)} - \mu\right) - \left(p_{ij}^{(b)} - \mu\right) \right| = \sum_{i,j} \left| p_{ij}^{(a)} - p_{ij}^{(b)} \right|$
- **Explanation**: It can be seen that μ will cancel out during distance computation because both pixel values are modified by the same constant.

1.0.2 3. Standardization

- **Formula Proof**: $\sum_{i,j} \left| \frac{p_{ij}^{(a)} - \mu}{\sigma} - \frac{p_{ij}^{(b)} - \mu}{\sigma} \right|$
- **Explanation**: Since is a global constant, the entire L1 distance is simply scaled by the same factor, σ^{-1}. This uniform scaling does not change the relative ordering of distances, so the nearest neighbors for any point remain the same. Therefore, the classifier's performance is unaffected.

1.0.3 5. Rotation

- **Formula**: $||R\mathbf{I}_a - R\mathbf{I}_b||_1$

Suppose that we rotate the pixel vectors of the images by rotation matrix R - **Explanation**: The L1 (Manhattan) distance is sensitive to the coordinate system, so rotating the data changes the distances. In contrast, the L2 (Euclidean) distance is rotationally invariant, meaning distances are preserved after rotation. The core reason is that L1 distance sums the components along the axes ($|x_1 - x_2| + |y_1 - y_2|$), and rotation mixes these x and y components. L2 distance squares the components, which removes their orientation before summing, making it insensitive to how the axes are aligned. - **Example**

Let's use three simple points and rotate them by 45 degrees. - **Points**: $\mathbf{I}_a = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{I}_b = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \mathbf{I}_c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- **Rotation Matrix (45°)**: $R = \begin{bmatrix} \cos(45^\circ) & -\sin(45^\circ) \\ \sin(45^\circ) & \cos(45^\circ) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$

1.0.4 Before Rotation

First, let's calculate the distances from the origin point $\mathbf{I}_a$: - **L1 Distances**: - $d_1(\mathbf{I}_a, \mathbf{I}_b) = |2 - 0| + |0 - 0| = 2$ - $d_1(\mathbf{I}_a, \mathbf{I}_c) = |1 - 0| + |1 - 0| = 2$ - *Result*: In L1 space, $\mathbf{I}_b$ and $\mathbf{I}_c$ are **equidistant** from $\mathbf{I}_a$.

- **L2 Distances**:
  - $d_2(\mathbf{I}_a, \mathbf{I}_b) = \sqrt{(2-0)^2 + (0-0)^2} = \sqrt{4} = 2$
  - $d_2(\mathbf{I}_a, \mathbf{I}_c) = \sqrt{(1-0)^2 + (1-0)^2} = \sqrt{2} \approx 1.414$
  - *Result*: In L2 space, $\mathbf{I}_c$ is **closer** to $\mathbf{I}_a$ than $\mathbf{I}_b$ is.

1.0.5 Applying the Rotation

Now we find the new coordinates by multiplying the original vectors by the rotation matrix R. - **Rotating $\mathbf{I}_b$**:

$$\mathbf{I}'_b = R \cdot \mathbf{I}_b = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} (2)\frac{\sqrt{2}}{2} + (0) \\ (2)\frac{\sqrt{2}}{2} + (0) \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ \sqrt{2} \end{bmatrix}$$

- **Rotating $\mathbf{I}_c$**:

$$\mathbf{I}'_c = R \cdot \mathbf{I}_c = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} (1)\frac{\sqrt{2}}{2} - (1)\frac{\sqrt{2}}{2} \\ (1)\frac{\sqrt{2}}{2} + (1)\frac{\sqrt{2}}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{2} \end{bmatrix}$$

1.0.6 After Rotation

Let's re-calculate the distances using the new rotated points ($\mathbf{I}_a$ remains at the origin, it does not change with the rotation matrix). - **L1 Distances**: - $d_1(\mathbf{I}_a, \mathbf{I}'_b) = |\sqrt{2} - 0| + |\sqrt{2} - 0| = 2\sqrt{2} \approx 2.828$ - $d_1(\mathbf{I}_a, \mathbf{I}'_c) = |0 - 0| + |\sqrt{2} - 0| = \sqrt{2} \approx 1.414$ - *Result*: The L1 distances have changed significantly. Now $\mathbf{I}'_c$ is much **closer** than $\mathbf{I}'_b$, breaking the original tie.

- **L2 Distances**:
  - $d_2(\mathbf{I}_a, \mathbf{I}'_b) = \sqrt{(\sqrt{2} - 0)^2 + (\sqrt{2} - 0)^2} = \sqrt{2 + 2} = \sqrt{4} = 2$
  - $d_2(\mathbf{I}_a, \mathbf{I}'_c) = \sqrt{(0 - 0)^2 + (\sqrt{2} - 0)^2} = \sqrt{2}$
  - *Result*: The L2 distances are **exactly the same** as before the rotation.

[ ]:
```
# 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 36.774678 seconds
One loop version took 37.762415 seconds
No loop version took 0.521999 seconds
```

1.0.7 Cross-validation

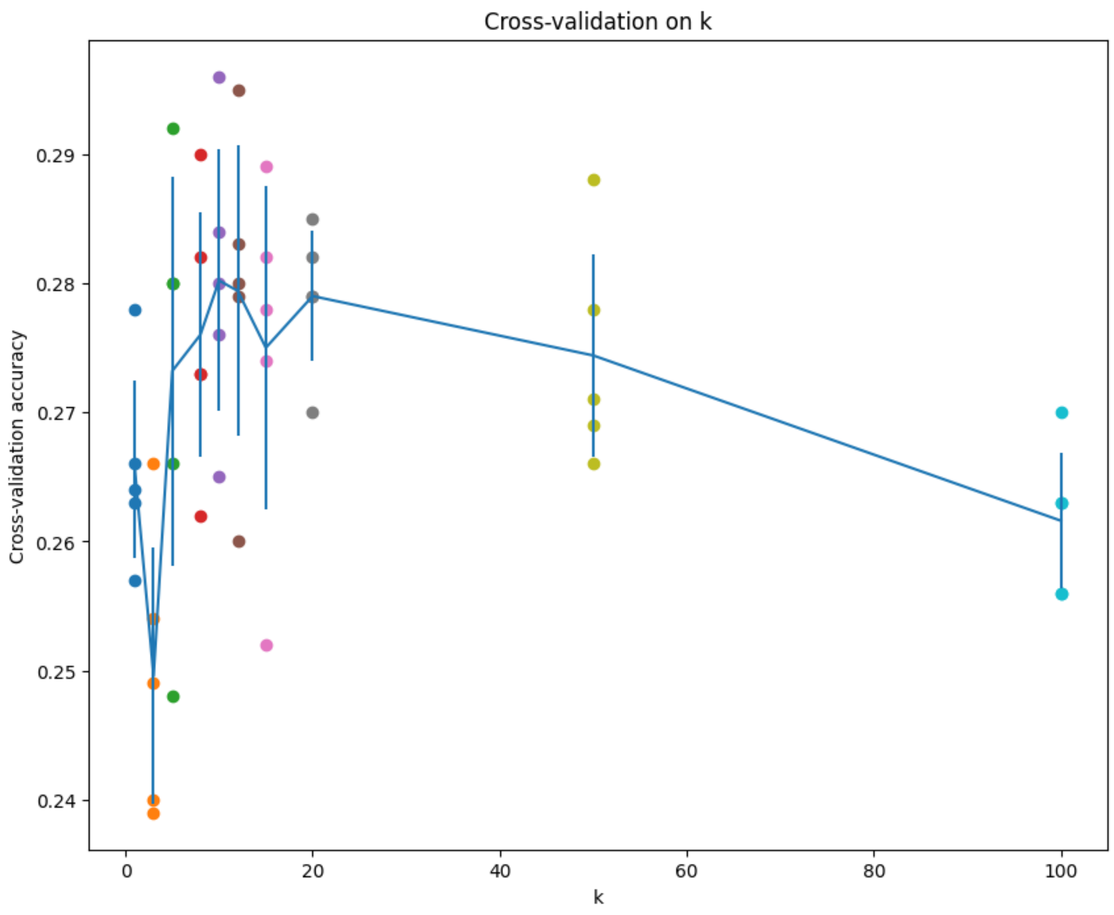
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

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



for k in k_choices:
    k_to_accuracies[k] = [] # initialize the dictionary with empty lists for␣
 ↪each k value
    for fold in range(num_folds):
        # creating validation set (the current fold)
        X_val_fold = X_train_folds[fold]
        y_val_fold = y_train_folds[fold]

        # creating training set (all other remaining folds)
        X_train_cv = np.concatenate([X_train_folds[i] for i in range(num_folds)␣
 ↪if i != fold])
        y_train_cv = np.concatenate([y_train_folds[i] for i in range(num_folds)␣
 ↪if i != fold])

        classifier.train(X_train_cv, y_train_cv)

        dists_cv = classifier.compute_distances_no_loops(X_val_fold)
        y_val_pred = classifier.predict_labels(dists_cv, k=k)

        accuracy = np.mean(y_val_pred == y_val_fold)
        k_to_accuracies[k].append(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 = 1

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

Inline Question 3

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

Your Answer :

2, 4 are true.

Your Explanation :

1. The decision boundary of a K-NN classifier is linear.

FALSE: The decision boundary for K-NN is highly non-linear. i.e. there is no a single hyperplane to seperate classes , instead it's formed by many small, linear segments (pieces of hyperplanes) that locally separate neighboring points of different classes.

---

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

TRUE: The training error for a `1-NN` classifier is always `zero`. This is because when you ask the model to classify a point that is already in the training set, its single nearest neighbor is the point itself, guaranteeing a correct prediction every time. A `5-NN`, however, considers the point itself plus four other neighbors. These other four points could potentially "outvote" the correct label, leading to a non-zero training error. Thus, `1-NN`'s training error (0%) is always less than or equal to `5-NN`'s.

---

3. The test error of a 1-NN is always lower than or equal to that of a 5-NN.

FALSE: The choice of `k` in a `K-NN` classifier directly controls the model's complexity and its tendency to overfit. It is maybe true that the performance on the classifier is better on the training set, because it basically memorizes it with `low bias` and but `high variance`. But by increasing the `k`, we effectively smooths out the decision boundary and making the model simpler. The classifier is no longer sensitive to a single noisy point because a majority vote from five neighbors is required. This introduces `bias`, but reduces the model's `variance` and makes it generalize better to the test set, which often leads to a lower test error. Thus, increasing `k` reduces the tendency to overfit, indicating that `5-NN` classifier is more robust, hence could be more accurate from the `1-NN` classifier.

---

4. The test time of a K-NN classifier increases with the size of the training set.

TRUE: `K-NN` is often called a `lazy learner` because it does no computation during training; it simply stores the entire training set. Instead, all the work happens at test time, which makes it ineffienct to classify a single test image, as the algorithm must compute the distance to every single image in the training set. Therefore, a larger training set means linearly more distance calculations, which directly results in a longer test time.

svm

October 2, 2025

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

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Multiclass Support Vector Machine exercise

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

In this exercise you will:

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

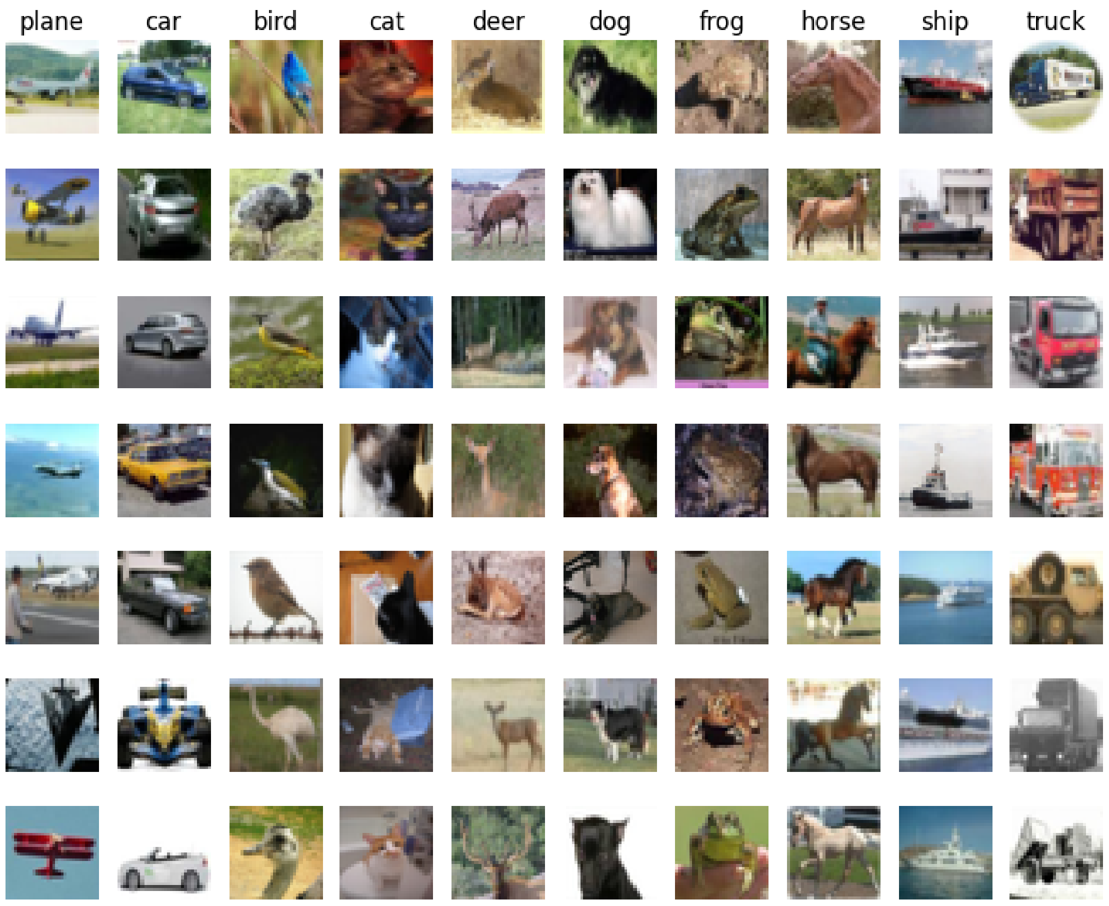
[3]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 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

import sys, types, importlib
imp = types.ModuleType("imp")
imp.reload = importlib.reload
sys.modules["imp"] = imp
%load_ext autoreload
%autoreload 2
```

1.1 CIFAR-10 Data Loading and Preprocessing

[4]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

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

[5]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



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

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

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

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

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

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

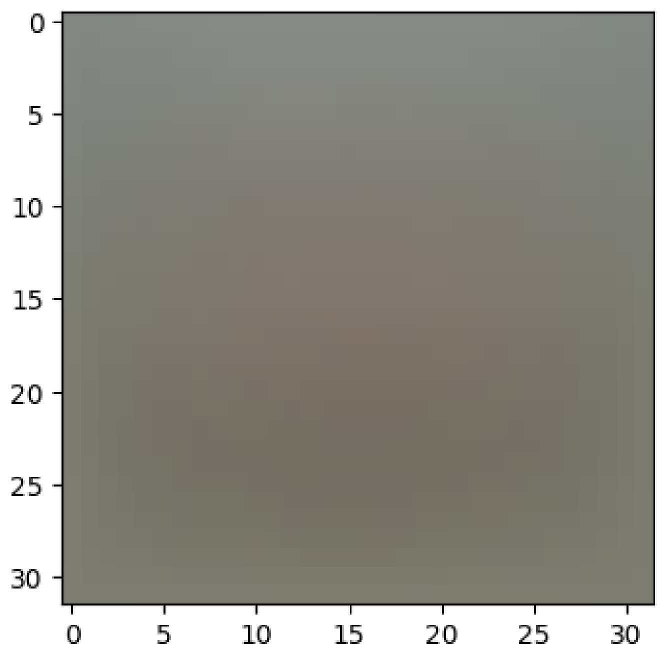
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

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

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

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

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

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

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

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

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



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

1.2 SVM Classifier

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

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

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

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

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

```
loss: 8.959278
```

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

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

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

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

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

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

```
numerical: 4.745658 analytic: 4.745658, relative error: 6.428574e-11
numerical: 1.950249 analytic: 1.950249, relative error: 7.331809e-11
numerical: -11.346741 analytic: -11.341605, relative error: 2.263932e-04
numerical: 3.052235 analytic: 3.052235, relative error: 7.860625e-11
numerical: -19.700703 analytic: -19.697260, relative error: 8.738299e-05
numerical: -9.899252 analytic: -9.899252, relative error: 2.236378e-11
numerical: 11.374831 analytic: 11.374831, relative error: 2.806217e-12
numerical: 13.291117 analytic: 13.291117, relative error: 5.804751e-12
numerical: -11.335709 analytic: -11.335709, relative error: 3.394847e-11
numerical: -8.377870 analytic: -8.377870, relative error: 1.832530e-11
numerical: -4.930677 analytic: -4.930677, relative error: 4.086303e-11
numerical: -30.831222 analytic: -30.831222, relative error: 6.924003e-12
numerical: -18.696890 analytic: -18.696890, relative error: 5.649089e-12
numerical: 18.166121 analytic: 18.166121, relative error: 1.148640e-11
numerical: 0.878660 analytic: 0.878660, relative error: 4.693929e-12
numerical: 11.102937 analytic: 11.102937, relative error: 2.581916e-12
numerical: 3.050364 analytic: 3.050364, relative error: 1.720496e-10
numerical: -23.663121 analytic: -23.651155, relative error: 2.528999e-04
numerical: 12.033370 analytic: 12.033370, relative error: 6.411915e-12
numerical: 17.636304 analytic: 17.636304, relative error: 1.332124e-11
```

Inline Question 1

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

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

SVM loss uses the hinge loss $\max(0, -)$, which looks smooth but actually has some tricky spots. For a simple 1D case, we think of $f(x) = \max(0, x)$ where $x = \hat{y}_i - \hat{y}_c + \Delta$.

$\hat{y}_i$ - predicted score for class i

$\hat{y}_c$ - predicted score for the correct class c

Δ - the margin (typically set to 1)

h - small value used for numerical gradient approximation

$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$ - definition of derivative/gradient

---

Why mismatches happen:

There are basically two reasons numerical and analytical gradients might not match:

1. `Numerical approximation error` - When we compute the numerical gradient, we pick some small h value and calculate the gradient based on the formula: $\frac{f(x+h)-f(x)}{h}$. But of course this is just an approximation of the true derivative (which is a limit as $h \to 0$). The analytical gradient is exact, so there's always going to be some small difference between analytical and numerical gradient.

2. `The kink at zero` - This is the bigger issue. At $x = 0$, the max function isn't actually differentiable - it has a sharp corner. So that the gradient is undefined, not smooth. The gradient jumps from 0 to 1. If we're unlucky and our numerical gradient check happens to straddle this point due to arbitrary choice of `h`, we can get weird results that don't match the analytical gradient.

Should we worry?

The approximation errors are usually tiny and not a big deal, the gradient update is still smooth. The kink issue is more concerning because it can give misleading gradient values with unintentional/not smooth updates on gradient.

What about changing the margin?

Intuitively, increasing Δ would reduce kinks because more examples would have positive margins. However, the network will just adjust and learn to make the score differences larger too. i.e. we would still have a change of landing on the edge where $x \approx 0$. The margin mostly affects how `confident` we want our classifications to be, not really the gradient check issues.

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

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

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

```
Naive loss: 8.959278e+00 computed in 0.080485s
Vectorized loss: 8.959278e+00 computed in 0.015859s
difference: 0.000000
```

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

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

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

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

```
Naive loss and gradient: computed in 0.066963s
Vectorized loss and gradient: computed in 0.009547s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

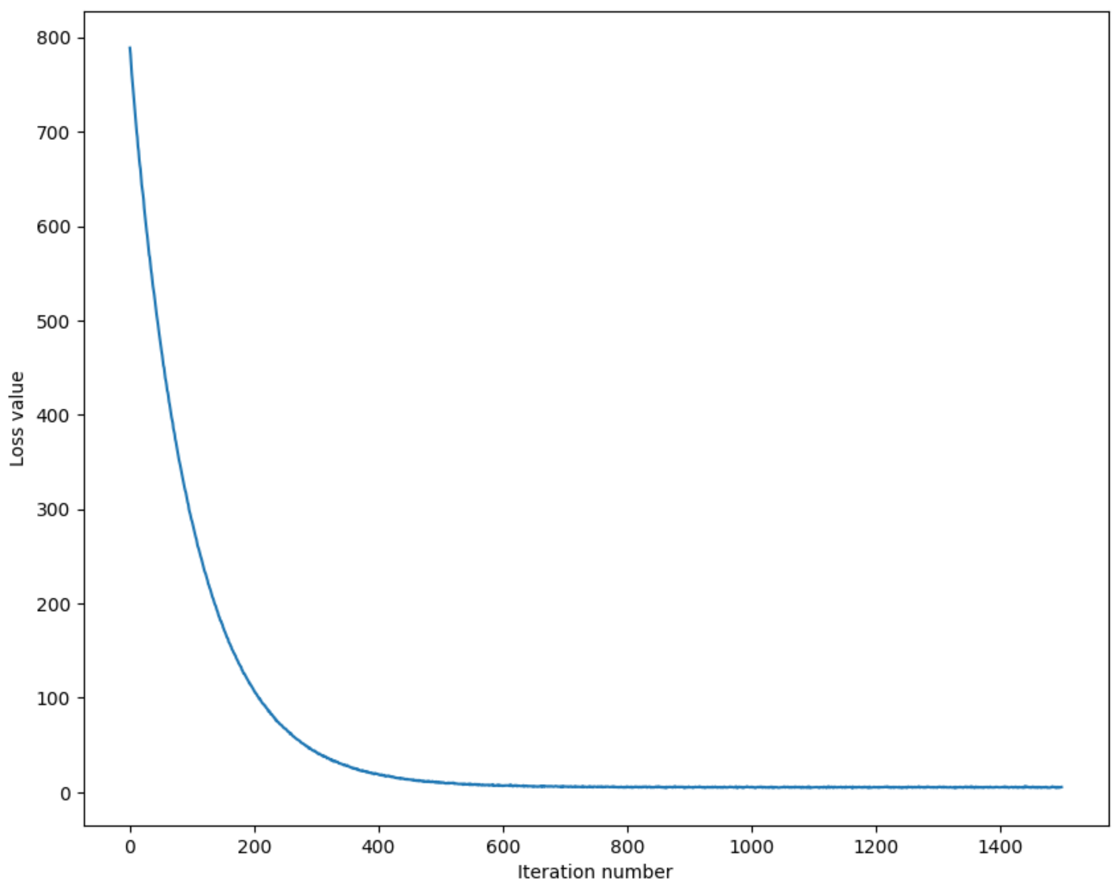
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

[13]:
```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
```

```
from cs231n.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 788.791719
iteration 100 / 1500: loss 287.143631
iteration 200 / 1500: loss 107.089680
iteration 300 / 1500: loss 42.652660
iteration 400 / 1500: loss 18.860900
iteration 500 / 1500: loss 10.026153
iteration 600 / 1500: loss 6.656764
iteration 700 / 1500: loss 6.057818
iteration 800 / 1500: loss 6.104131
iteration 900 / 1500: loss 5.355947
iteration 1000 / 1500: loss 5.715910
iteration 1100 / 1500: loss 5.292486
iteration 1200 / 1500: loss 5.230937
iteration 1300 / 1500: loss 5.486939
iteration 1400 / 1500: loss 5.239034
That took 7.319694s
```

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



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

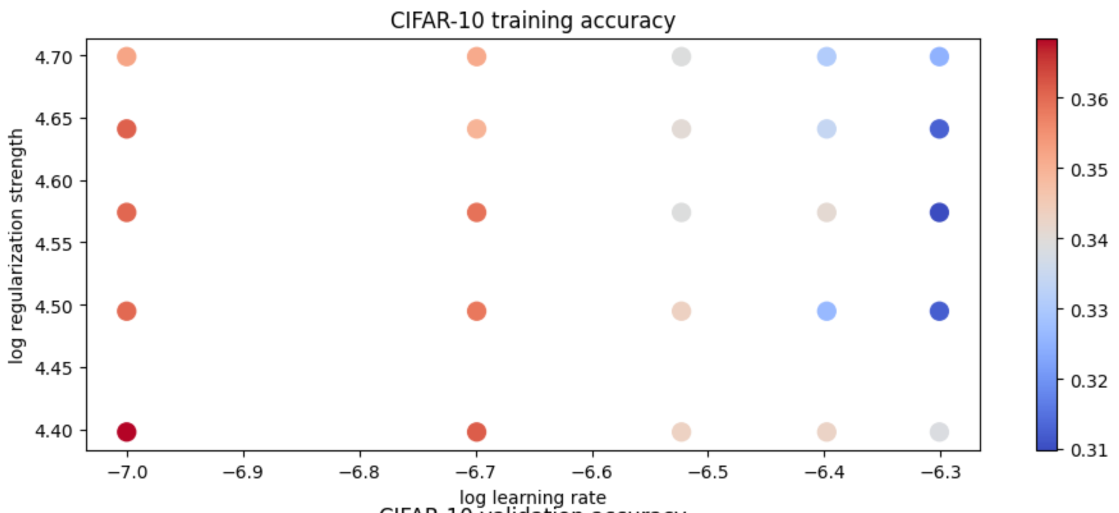
```
training accuracy: 0.372653
validation accuracy: 0.385000
```

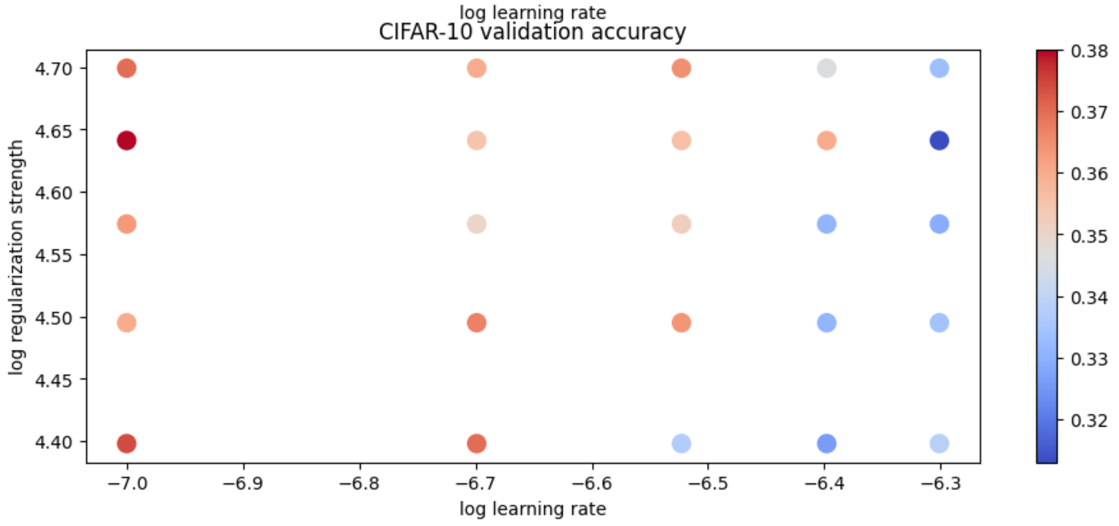
[16]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 (> 0.385) on the validation set.

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

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

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = np.linspace(1e-7, 5e-7, 5)
regularization_strengths = np.linspace(2.5e4, 5e4, 5)


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


for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()

        # train the model with the current set of hyperparameters
        # we use 1500 iterations
        svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                  num_iters=1000, verbose=False)

        # predict on the training data and validation data, then calculate␣
 ↪accuracy
        y_train_pred, y_val_pred = svm.predict(X_train), svm.predict(X_val)
        train_accuracy, val_accuracy = np.mean(y_train == y_train_pred), np.
 ↪mean(y_val == y_val_pred)
```

```
        # store the results in the dictionary
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # if this model has the best validation accuracy so far, save it
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm


# *****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 2.500000e+04 train accuracy: 0.368531 val accuracy: 0.374000
lr 1.000000e-07 reg 3.125000e+04 train accuracy: 0.360265 val accuracy: 0.360000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.360449 val accuracy: 0.363000
lr 1.000000e-07 reg 4.375000e+04 train accuracy: 0.361041 val accuracy: 0.380000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.352204 val accuracy: 0.370000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.361612 val accuracy: 0.370000
lr 2.000000e-07 reg 3.125000e+04 train accuracy: 0.358265 val accuracy: 0.367000
lr 2.000000e-07 reg 3.750000e+04 train accuracy: 0.359061 val accuracy: 0.350000
lr 2.000000e-07 reg 4.375000e+04 train accuracy: 0.349510 val accuracy: 0.355000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.351224 val accuracy: 0.360000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.343102 val accuracy: 0.337000
lr 3.000000e-07 reg 3.125000e+04 train accuracy: 0.343469 val accuracy: 0.364000
lr 3.000000e-07 reg 3.750000e+04 train accuracy: 0.339041 val accuracy: 0.352000
lr 3.000000e-07 reg 4.375000e+04 train accuracy: 0.340367 val accuracy: 0.356000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.338939 val accuracy: 0.365000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.342633 val accuracy: 0.326000
lr 4.000000e-07 reg 3.125000e+04 train accuracy: 0.326408 val accuracy: 0.331000
lr 4.000000e-07 reg 3.750000e+04 train accuracy: 0.340898 val accuracy: 0.331000
lr 4.000000e-07 reg 4.375000e+04 train accuracy: 0.334245 val accuracy: 0.360000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.330857 val accuracy: 0.346000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.338592 val accuracy: 0.338000
lr 5.000000e-07 reg 3.125000e+04 train accuracy: 0.312143 val accuracy: 0.334000
lr 5.000000e-07 reg 3.750000e+04 train accuracy: 0.309776 val accuracy: 0.329000
lr 5.000000e-07 reg 4.375000e+04 train accuracy: 0.312694 val accuracy: 0.313000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.324980 val accuracy: 0.333000
best validation accuracy achieved during cross-validation: 0.380000
```

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

# pdb.set_trace()

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

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

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



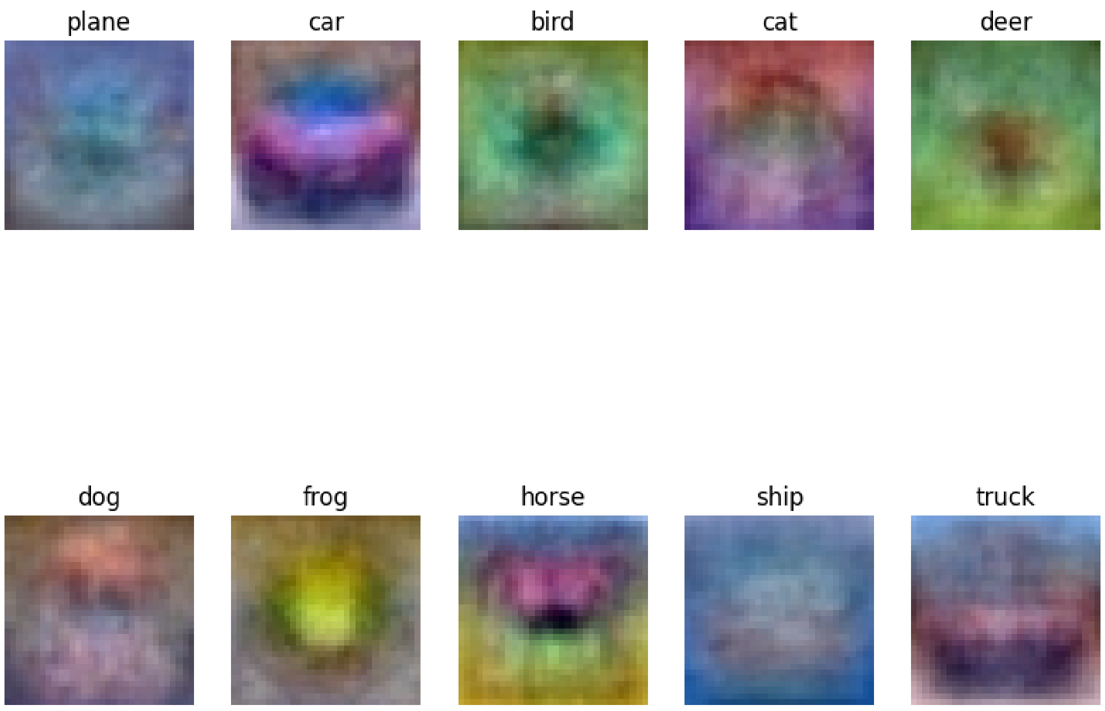


[18]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.367000
```

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

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

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



Inline question 2

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

Your Answer :

- Each image visualizes the learned “template” for one of the ten classes. The SVM classifier works by taking an input image and matching it against each of these ten templates; the class whose template results in the highest “match score” (dot product) is the predicted label, unlike the K-NN classifier which compares every test image with training samples.
- These templates look like blurry, averaged pictures because the linear classifier is forced to create a single template that must work for all the different variations within a class. For example, to classify both horses facing left and horses facing right, the learned template ends up being a ghostly superposition of both, which is why the “horse” image appears to have two heads. Similarly, the “car” template has a reddish-bluish blob in the middle because it

has learned a general shape that is reddish for red cars and bluish for blue cars. That's the templates happen to be symmetric visualizing the average we can say to generalize the best for each class/category.

softmax

October 2, 2025

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

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Softmax exercise

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

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

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

```
[ ]: 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

     import sys, types, importlib
     imp = types.ModuleType("imp")
     imp.reload = importlib.reload
     sys.modules["imp"] = imp

     %load_ext autoreload
     %autoreload 2
```

```
[ ]: def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
      ↪num_dev=500):
         """
         Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
         it for the linear classifier. These are the same steps as we used for the
         SVM, but condensed to a single function.
         """
         # Load the raw CIFAR-10 data
         cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

         X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

         # subsample the data
         mask = list(range(num_training, num_training + num_validation))
         X_val = X_train[mask]
         y_val = y_train[mask]
```

```
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

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

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

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


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

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

1.1 Softmax Classifier

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

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

     from cs231n.classifiers.softmax import softmax_loss_naive
     import time

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

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

```
loss: 2.350699
sanity check: 2.302585
```

Inline Question 1

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

$YourAnswer:$

Before the training, in the code the weights W are initialized with small random numbers, meaning that the initial scores for all `10` classes will also be small and very similar to each other.

When the softmax function processes these nearly identical scores, it produces a probability distribution that is close to uniform. With `10` classes, each class gets a probability of roughly `1/10 = 0.1`.

As the loss is defined as the negative log of the probability assigned to the correct class, we expect the initial loss to be close to `-log(0.1)`

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

     # As we did for the SVM, use numeric gradient checking as a debugging tool.
     # The numeric gradient should be close to the analytic gradient.
     from cs231n.gradient_check import grad_check_sparse
```

```
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

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

```
numerical: 1.449644 analytic: 1.449644, relative error: 7.192972e-09
numerical: -1.150992 analytic: -1.150992, relative error: 5.087368e-09
numerical: 0.157880 analytic: 0.157880, relative error: 1.505929e-07
numerical: 0.434982 analytic: 0.434982, relative error: 2.811588e-08
numerical: 0.364082 analytic: 0.364082, relative error: 4.561422e-08
numerical: 0.011960 analytic: 0.011960, relative error: 1.351624e-06
numerical: 0.625386 analytic: 0.625386, relative error: 8.855834e-08
numerical: 0.122785 analytic: 0.122785, relative error: 3.031538e-07
numerical: -2.890379 analytic: -2.890379, relative error: 2.195968e-08
numerical: 0.661024 analytic: 0.661024, relative error: 7.788115e-08
numerical: -0.946829 analytic: -0.946829, relative error: 4.711387e-08
numerical: -1.865720 analytic: -1.865720, relative error: 1.779374e-08
numerical: 1.700665 analytic: 1.700665, relative error: 4.754563e-09
numerical: 1.933553 analytic: 1.933553, relative error: 1.029153e-08
numerical: -4.851960 analytic: -4.851960, relative error: 7.311115e-09
numerical: -2.327838 analytic: -2.327838, relative error: 1.776929e-09
numerical: -2.099263 analytic: -2.099263, relative error: 1.161158e-08
numerical: 1.318757 analytic: 1.318756, relative error: 3.183206e-08
numerical: 0.508669 analytic: 0.508669, relative error: 1.527923e-07
numerical: 0.513752 analytic: 0.513752, relative error: 1.058933e-07
```

[9]:
```
# Now that we have a naive implementation of the softmax loss function and its␣
 ↪gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version␣
 ↪should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

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

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

```
naive loss: 2.350699e+00 computed in 0.572578s
vectorized loss: 2.350699e+00 computed in 0.051285s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[11]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

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

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

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = np.linspace(1e-7, 5e-7, 5)
regularization_strengths = np.linspace(2.5e4, 5e4, 5)

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

# Iterate through each combination of hyperparameters
for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()

        # train the model with the current hyperparameters
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500, verbose=False)

        # predict on the training and validation sets, then calculate accuricies
        y_train_pred, y_val_pred = softmax.predict(X_train), softmax.
 ↪predict(X_val)
```

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

        # store in the dictionary
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # check if this is the best model so far and save it if it is
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax

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

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

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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.323449 val accuracy: 0.338000
lr 1.000000e-07 reg 3.125000e+04 train accuracy: 0.316408 val accuracy: 0.330000
lr 1.000000e-07 reg 3.750000e+04 train accuracy: 0.315367 val accuracy: 0.323000
lr 1.000000e-07 reg 4.375000e+04 train accuracy: 0.310429 val accuracy: 0.325000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.301469 val accuracy: 0.324000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.319796 val accuracy: 0.343000
lr 2.000000e-07 reg 3.125000e+04 train accuracy: 0.314776 val accuracy: 0.331000
lr 2.000000e-07 reg 3.750000e+04 train accuracy: 0.312224 val accuracy: 0.327000
lr 2.000000e-07 reg 4.375000e+04 train accuracy: 0.305286 val accuracy: 0.318000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.302857 val accuracy: 0.308000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.321224 val accuracy: 0.343000
lr 3.000000e-07 reg 3.125000e+04 train accuracy: 0.323796 val accuracy: 0.346000
lr 3.000000e-07 reg 3.750000e+04 train accuracy: 0.317163 val accuracy: 0.321000
lr 3.000000e-07 reg 4.375000e+04 train accuracy: 0.307102 val accuracy: 0.322000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.303776 val accuracy: 0.322000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.323673 val accuracy: 0.332000
lr 4.000000e-07 reg 3.125000e+04 train accuracy: 0.320082 val accuracy: 0.330000
lr 4.000000e-07 reg 3.750000e+04 train accuracy: 0.313020 val accuracy: 0.320000
lr 4.000000e-07 reg 4.375000e+04 train accuracy: 0.305265 val accuracy: 0.324000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.310490 val accuracy: 0.326000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.328000 val accuracy: 0.322000
lr 5.000000e-07 reg 3.125000e+04 train accuracy: 0.300816 val accuracy: 0.317000
lr 5.000000e-07 reg 3.750000e+04 train accuracy: 0.315857 val accuracy: 0.329000
lr 5.000000e-07 reg 4.375000e+04 train accuracy: 0.316204 val accuracy: 0.331000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.300469 val accuracy: 0.316000
```

```
best validation accuracy achieved during cross-validation: 0.346000
```

[12]:
```
# 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: 0.341000
```

Inline Question 2 - *True or False*

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

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

True.

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

We can add a "well-classified" data point that has zero loss for an `SVM` (only cares for the margin), but any data point will always have a non-zero loss for `Softmax`.

-

1.1.1 Why Softmax Loss is Never Zero

The Softmax classifier's goal is different. It aims to make the **probability** of the correct class, P_c, as close to **1** as possible.

- The loss is

  $L_n = -\log(P_c)$, where $P_c = \frac{e^{s_c}}{\sum_j e^{s_j}}$.

For the loss to be zero ($-\log(P_c) = 0$), the probability P_c would have to be exactly 1. For P_c to be 1, the denominator ($\sum_j e^{s_j}$) would have to equal the numerator (e^{s_c}). This would require the scores for all incorrect classes (s_i) to be negative infinity ($e^{-\infty} = 0$), which is a mathematical impossibility for a linear classifier that produces finite scores.

No matter how well-classified a data point is, its probability for the correct class will always be slightly less than 1 (e.g., 0.9999). Therefore, its loss, $-\log(P_c)$, will always be a small positive number, never exactly zero.

e.g.

again assume the score function used in above.

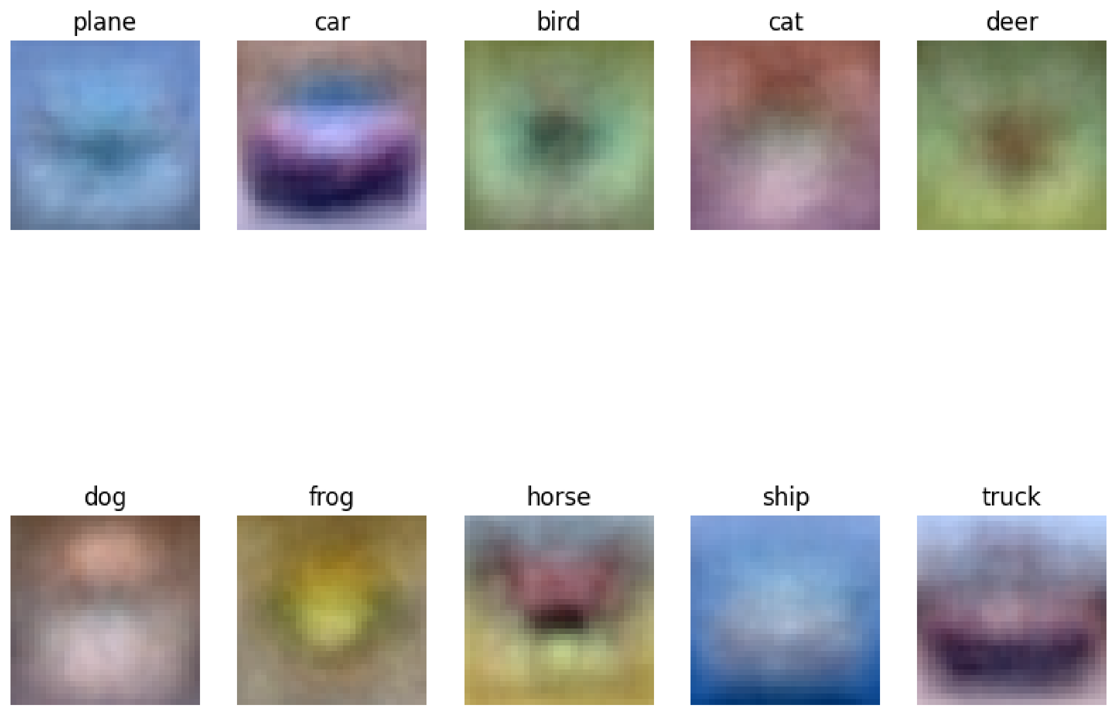
Softmax Calculation:

1. **Calculate exponentiated scores:**
   - $e^{10} \approx 22026.47$
   - $e^{-15} \approx 3.06 \times 10^{-7}$
   - $e^{-20} \approx 2.06 \times 10^{-9}$

2. **Calculate the probability of the correct class:**
   - $P_{correct} = \frac{e^{10}}{e^{10}+e^{-15}+e^{-20}} = \frac{22026.47}{22026.4700003} \approx 0.999999986$
3. **Calculate the loss:**
   - $L = -\log(0.999999986) \approx 1.4 \times 10^{-8}$

The Softmax loss for this data point is a very small positive number, but it is not zero. Adding it to the training set will slightly increase the total loss.

[13]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

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

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

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



two_layer_net

October 2, 2025

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

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

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

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

1 Fully-Connected Neural Nets

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

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

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

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

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

  return dx, dw
```

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

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

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

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

import sys, types, importlib
imp = types.ModuleType("imp")
imp.reload = importlib.reload
sys.modules["imp"] = imp
%load_ext autoreload
%autoreload 2

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

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

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

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

2 Affine layer: forward

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

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

```
[ ]: # Test the affine_forward function

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

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

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

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

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

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

3 Affine layer: backward

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

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

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

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

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

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

4 ReLU activation: forward

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

```
[ ]: # Test the relu_forward function

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

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

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

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

5 ReLU activation: backward

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

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

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

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

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

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

5.1 Inline Question 1:

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

5.2 Answer:

`Sigmoid` and `ReLU` activation functions cause this `zero-gradient problem`, on contrast to Leaky ReLU which is introduced to solve Dying ReLU problem, that is going to be explained below.

A neuron in a neural network might stop learning when updating of the weights in earlier layers became vanishingly small because of the small gradients getting multiplied down the network, over the course of backpropagation.

1. Sigmoid

   As not generally used in hidden layers, the Sigmoid function has vanishing gradient problem. This problem occurs when the inputs are either very large positive numbers or they are very large negative numbers. The shape of this activation function actually gives clue about these problem, that it flattens out at both ends.

When a neuron's output is in these mentioned flat regions, it becomes `saturated`, and its gradient is nearly zero, effectively halting learning for that neuron.

2. ReLU (Rectified Linear Unit)

   Unlike sigmoid function. ReLu activation function has this problem on one side, known as `Dying ReLu` problem.

   The gradient is exactly zero for any negative input (x<0). While the gradient is a constant 1 for all positive inputs (which is great for learning), if a neuron consistently receives negative inputs, its weights will never be updated because the gradient passed back through it will always be zero. The neuron becomes "stuck" and stops participating in the learning process. So that the neuron becomes `dead`.

   This problem is solved by the `Leaky ReLU` function, as for negative inputs, instead of being zero, the gradient is a small, non-zero constant (e.g., 0.01). Because the gradient is never zero, a small but consistent gradient flow is always maintained, preventing the neuron from dying.

6 "Sandwich" layers

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

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

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

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

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

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

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

7 Loss layers: Softmax and SVM

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

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

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

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

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

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

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

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-09
```

8 Two-layer network

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

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

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

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

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

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.20e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

9 Solver

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

[ ]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

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

```
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-4,
                },
                lr_decay=0.95,
                num_epochs=10,
                batch_size=200,
                print_every=100,
                verbose=True)

solver.train()

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

```
(Iteration 1 / 2450) loss: 2.302457
(Epoch 0 / 10) train acc: 0.074000; val_acc: 0.072000
(Iteration 101 / 2450) loss: 2.253992
(Iteration 201 / 2450) loss: 2.144879
(Epoch 1 / 10) train acc: 0.241000; val_acc: 0.251000
(Iteration 301 / 2450) loss: 2.087296
(Iteration 401 / 2450) loss: 2.006982
(Epoch 2 / 10) train acc: 0.297000; val_acc: 0.295000
(Iteration 501 / 2450) loss: 1.981725
(Iteration 601 / 2450) loss: 1.932737
(Iteration 701 / 2450) loss: 1.804961
(Epoch 3 / 10) train acc: 0.353000; val_acc: 0.335000
(Iteration 801 / 2450) loss: 1.814617
(Iteration 901 / 2450) loss: 1.855922
(Epoch 4 / 10) train acc: 0.367000; val_acc: 0.350000
(Iteration 1001 / 2450) loss: 1.722371
(Iteration 1101 / 2450) loss: 1.818258
(Iteration 1201 / 2450) loss: 1.761373
(Epoch 5 / 10) train acc: 0.388000; val_acc: 0.373000
(Iteration 1301 / 2450) loss: 1.660947
(Iteration 1401 / 2450) loss: 1.819146
(Epoch 6 / 10) train acc: 0.365000; val_acc: 0.380000
(Iteration 1501 / 2450) loss: 1.674836
(Iteration 1601 / 2450) loss: 1.798832
(Iteration 1701 / 2450) loss: 1.842074
(Epoch 7 / 10) train acc: 0.405000; val_acc: 0.383000
(Iteration 1801 / 2450) loss: 1.708081
```

```
(Iteration 1901 / 2450) loss: 1.630032
(Epoch 8 / 10) train acc: 0.392000; val_acc: 0.401000
(Iteration 2001 / 2450) loss: 1.735227
(Iteration 2101 / 2450) loss: 1.592238
(Iteration 2201 / 2450) loss: 1.631562
(Epoch 9 / 10) train acc: 0.406000; val_acc: 0.405000
(Iteration 2301 / 2450) loss: 1.640245
(Iteration 2401 / 2450) loss: 1.699114
(Epoch 10 / 10) train acc: 0.401000; val_acc: 0.412000
```

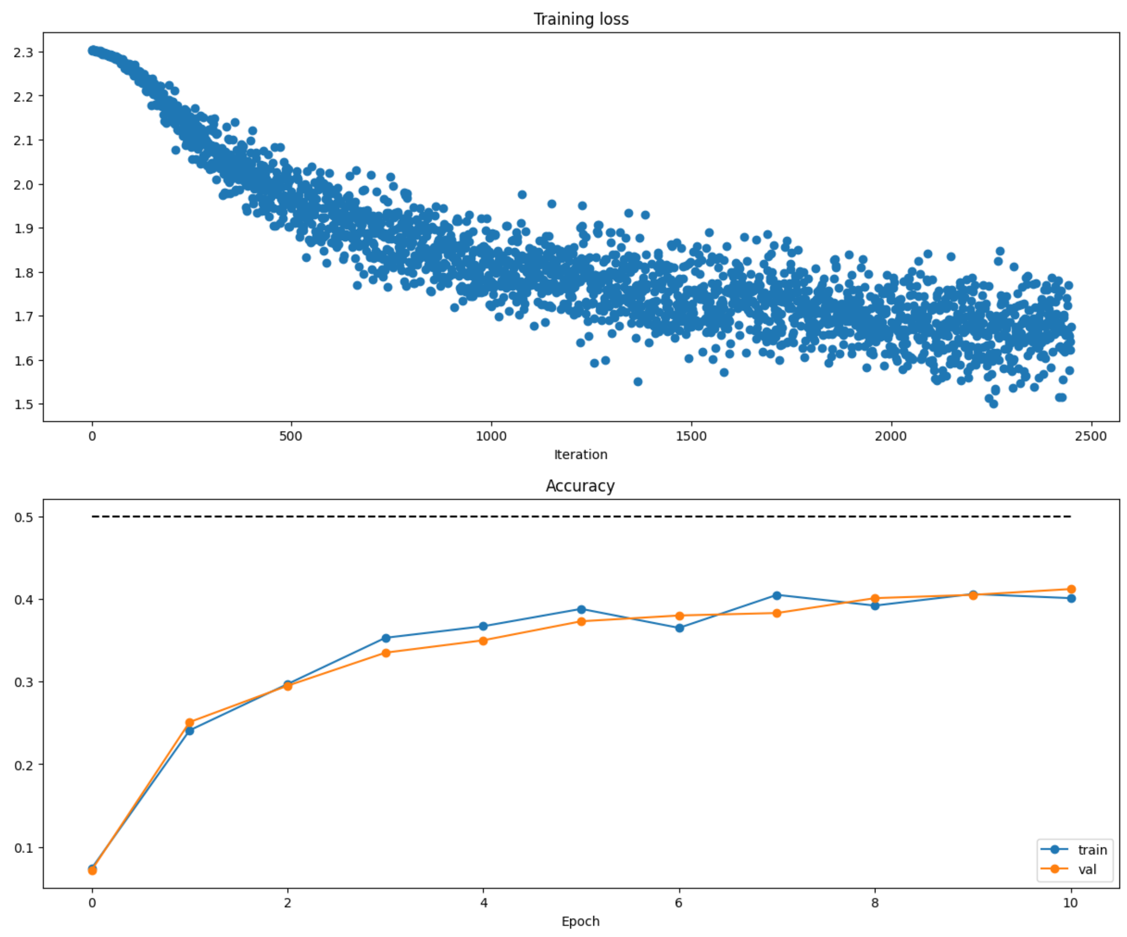
10 Debug the training

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

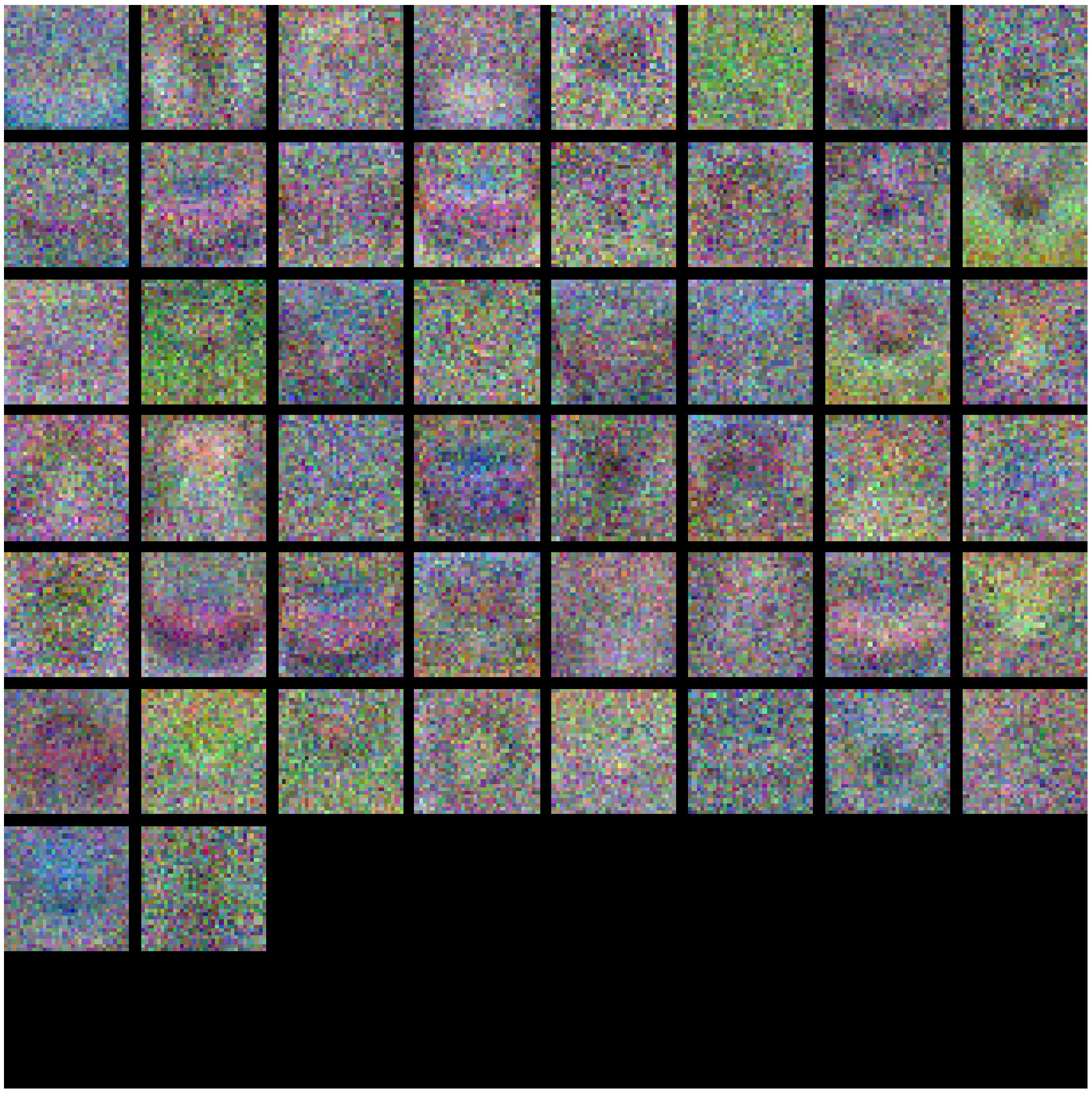
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

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

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

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



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

     # Visualize the weights of the network

     def show_net_weights(net):
         W1 = net.params['W1']
         W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
         plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
         plt.gca().axis('off')
         plt.show()

     show_net_weights(model)
```



11 Tune your hyperparameters

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

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

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

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

[26]:
```
best_model = None


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

best_model = None
best_val_acc = -1
results = {}

# we define the grid of hyperparameters to search over.
learning_rates = np.logspace(-3.3, -2.7, 3)
regularization_strengths = np.linspace(0.1, 0.6, 4)

# Iterate through each combination of hyperparameters
for lr in learning_rates:
    for reg in regularization_strengths:
        model = TwoLayerNet(
            hidden_dim=100,
            reg=reg,
            weight_scale=1e-3
        )

        solver = Solver(
            model,
            data,
            update_rule='sgd',
            optim_config={'learning_rate': lr},
            lr_decay=0.95,
            num_epochs=10,
            batch_size=200,
            verbose=False
        )

        solver.train()

        # get the best validation accuracy achieved during this training run
        val_acc = solver.best_val_acc
        results[(lr, reg)] = val_acc
        print(f'lr {lr:.1e} reg {reg:.2f} => val_acc: {val_acc:.4f}')

        # Check if this model is the best one we've seen so far
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_model = model

# Print the final best validation accuracy
print(f'\nBest validation accuracy achieved during hyperparameter search:␣
 ↪{best_val_acc:.4f}')



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

```
lr 5.0e-04 reg 0.10 => val_acc: 0.5140
lr 5.0e-04 reg 0.27 => val_acc: 0.5080
lr 5.0e-04 reg 0.43 => val_acc: 0.5160
lr 5.0e-04 reg 0.60 => val_acc: 0.5120
lr 1.0e-03 reg 0.10 => val_acc: 0.5270
lr 1.0e-03 reg 0.27 => val_acc: 0.5150
lr 1.0e-03 reg 0.43 => val_acc: 0.5200
lr 1.0e-03 reg 0.60 => val_acc: 0.5150
lr 2.0e-03 reg 0.10 => val_acc: 0.5190
lr 2.0e-03 reg 0.27 => val_acc: 0.5060
lr 2.0e-03 reg 0.43 => val_acc: 0.5060
lr 2.0e-03 reg 0.60 => val_acc: 0.5120

Best validation accuracy achieved during hyperparameter search: 0.5270
```

12 Test your model!

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

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

```
Validation set accuracy:  0.528
```

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

```
Test set accuracy:  0.513
```

12.1 Inline Question 2:

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

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

$Your Answer:$

Statements `1 and 3` decrease the gap.

$Your Explanation:$

We define this gap between a high training accuracy and a much lower testing accuracy as the result of `overfitting`. The model has learned the training data too well, so it has successfully memorized the training data, including its noise and quirks, such that it fails to generalize to new, unseen data.

- Statement 1 is `a possible way` to overcome this issue:

  A larger and more diverse dataset makes it much harder for the model to simply memorize the training examples. It forces the model to learn the true, generalizable patterns in the data rather than the noise. But with larger dataset we need more varying samples as the model will not learn from the similar samples.

- Statement 2 is `not a way` to overcome this issue, instead it helps overfitting:

  Adding more hidden units increases the model's capacity (its ability to learn complex functions), making the model more powerful but also more prone to overfitting the training data. After a certain value of training accuracy, we don't generally increase hidden units, because as it increases the training accuracy, it also decreases the test accuracy.

- Statement 3 is `also a way` to overcome overfitting problem:

  Specifically designed to reduce overfitting, regularization adds a penalty to the loss function for large weights. This discourages the model from becoming too complex and relying heavily on any single feature. It forces the model to find a simpler, "smoother" solution that generalizes better to the test set.

features

October 2, 2025

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

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

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

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

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 Image features exercise

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

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

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

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


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

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

import sys, types, importlib
imp = types.ModuleType("imp")
imp.reload = importlib.reload
sys.modules["imp"] = imp

%load_ext autoreload
%autoreload 2
```

1.1 Load data

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

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

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

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

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

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

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

1.2 Extract Features

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

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

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

[5]:
```
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 SVM on features

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

[16]:
```
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

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

for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()

        svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,
                  num_iters=1500, verbose = False)

        # predict on the training and validation sets
        y_train_pred, y_val_pred = svm.predict(X_train_feats), svm.
 ↪predict(X_val_feats)
```

```
        # then calculate the accuracy for this model and store them in␣
 ↪dictionary
        train_accuracy, val_accuracy = np.mean(y_train == y_train_pred), np.
 ↪mean(y_val == y_val_pred)
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # update best_val if this model's validation accuracy is the best we've␣
 ↪seen.
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm


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

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

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

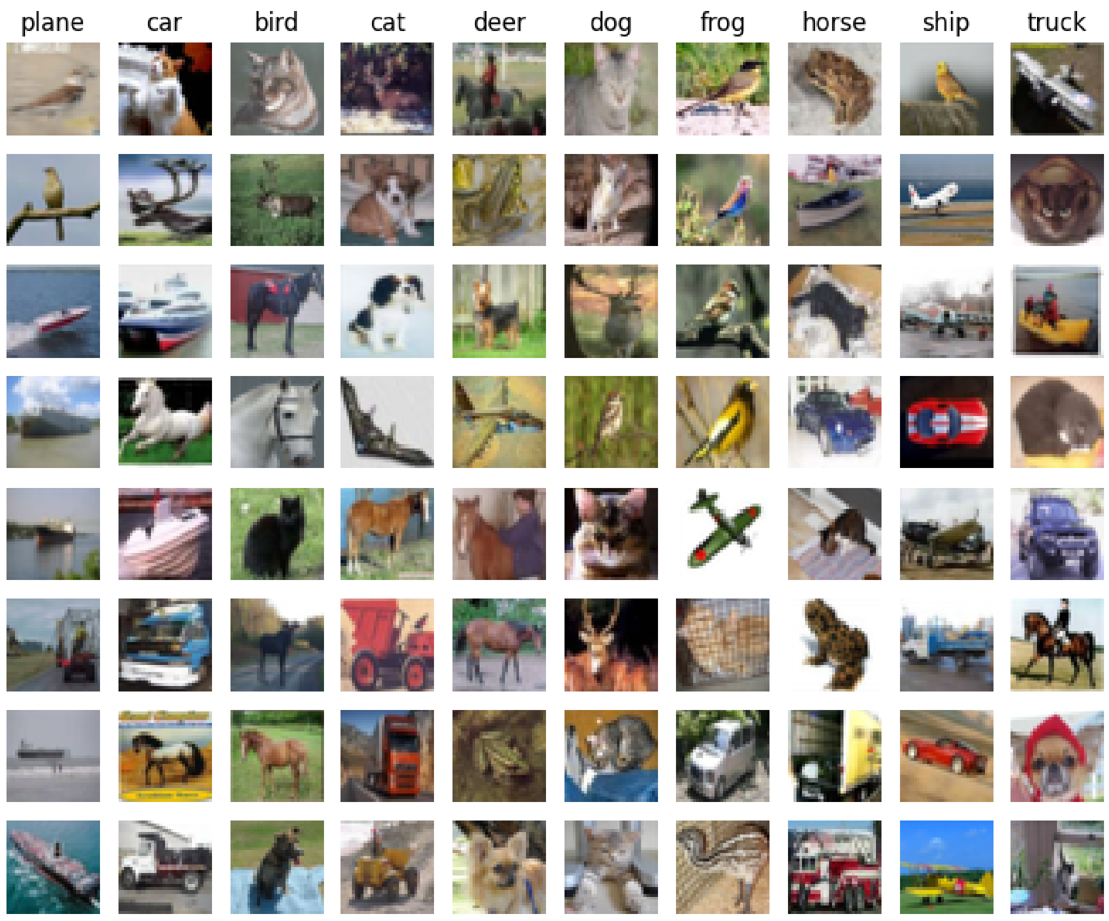
```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.085694 val accuracy: 0.102000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.106980 val accuracy: 0.097000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.413551 val accuracy: 0.421000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.090224 val accuracy: 0.088000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.414020 val accuracy: 0.420000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.416327 val accuracy: 0.423000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.416265 val accuracy: 0.412000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.398265 val accuracy: 0.404000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.327469 val accuracy: 0.369000
best validation accuracy achieved: 0.423000
```

[17]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
 ↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.426
```

[18]:
```
# 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 :

A Linear SVM learns one `template` for each category from the training data. To classify a new image, it calculates a `match score` (a dot product) between the image and each of the ten templates. The template with the highest score wins. The problem is that these templates are very simple. They are essentially an average of all the images in a class. This approach is easily fooled by color, texture, and background. So, any test image matches with any of these is likely to be misclassified to that template.

- e.g. cat classiied as dog makes sense because the SVM learns a single, blurry `cat` template and a single, blurry `dog` template. So it is highly possible that the two templates will end up looking very similar—like generic, some color, four-legged animal shapes. That ends up misclassification of a cat image in test data as a dog, because it also matchs with `dog` template.
- Another example is airplane classified as bird, it also makes sense because Example 2: Airplane classified as Bird the `airplane` template learns to look for a small shape in the center of a large blue background (the sky) which also the `bird` template looks ofr. The resulting templates can be very similar. A test image of a plane will get a high score from the airplane template, but it will also get a high score from the bird template because the background match is so strong. So, that misclassification is likely to occur in this matching.
- One more example can be given as truck misclassified as car, briefly, average `truck` template can be red color with four wheels, viewed from front, thus, a red car in test data, having the same color with truck template will be misclassified as truck, which makes sense.

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.

[19]:
```
# 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)
```

[23]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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

data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

#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)*****

best_val_acc = -1

num_trials = 10

print(f"Running {num_trials} random trials for hyperparameter tuning...")

# run the random search
for i in range(num_trials):
    # sample learning rate from a log-uniform distribution between 1e-3 and 1e-1
    lr = 10**np.random.uniform(-3, -1)

    # Sample regularization strength from a log-uniform distribution between␣
 ↪1e-5 and 1e-3
    reg = 10**np.random.uniform(-5, -3)
    hidden_dim = np.random.randint(400, 600)

    # for each experiment, we create a new, fresh model.
    # so, that we comment out the one that is defined outside the loop
    model = TwoLayerNet(input_dim, hidden_dim, num_classes)

    solver = Solver(
        model,
        data,
        optim_config={'learning_rate': lr},
        num_epochs=20,
        verbose=False
        )
    solver.train()

    val_acc = solver.best_val_acc

    print(f'Trial {i+1}/{num_trials}: lr {lr:.2e}, reg {reg:.2e}, hid␣
 ↪{hidden_dim} => val_acc: {val_acc:.4f}')

    # if this fresh model is the best so far, we save it
    if val_acc > best_val_acc:
      best_val_acc = val_acc
      best_net = model

print(f'\nBest validation accuracy: {best_val_acc:.4f}')

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

```
Running 10 random trials for hyperparameter tuning…
Trial 1/10: lr 3.01e-02, reg 1.20e-05, hid 553 => val_acc: 0.5850
Trial 2/10: lr 5.79e-02, reg 1.94e-04, hid 537 => val_acc: 0.6030
Trial 3/10: lr 7.30e-03, reg 6.29e-04, hid 548 => val_acc: 0.5140
Trial 4/10: lr 5.58e-02, reg 3.03e-05, hid 559 => val_acc: 0.6060
Trial 5/10: lr 1.91e-03, reg 8.34e-04, hid 585 => val_acc: 0.3490
Trial 6/10: lr 3.18e-03, reg 9.00e-04, hid 426 => val_acc: 0.4350
Trial 7/10: lr 9.49e-02, reg 3.16e-05, hid 509 => val_acc: 0.6100
Trial 8/10: lr 2.73e-02, reg 1.99e-04, hid 530 => val_acc: 0.5840
Trial 9/10: lr 2.11e-03, reg 5.07e-04, hid 491 => val_acc: 0.3710
Trial 10/10: lr 1.56e-02, reg 2.34e-05, hid 559 => val_acc: 0.5410

Best validation accuracy: 0.6100
```

[24]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

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

```
0.594
```