knn

August 18, 2021

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

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

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

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

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

kNN
kNN k k

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

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

# This is a bit of magic to make matplotlib figures appear inline in the␣
 ↪notebook
# rather than in a new window.
```

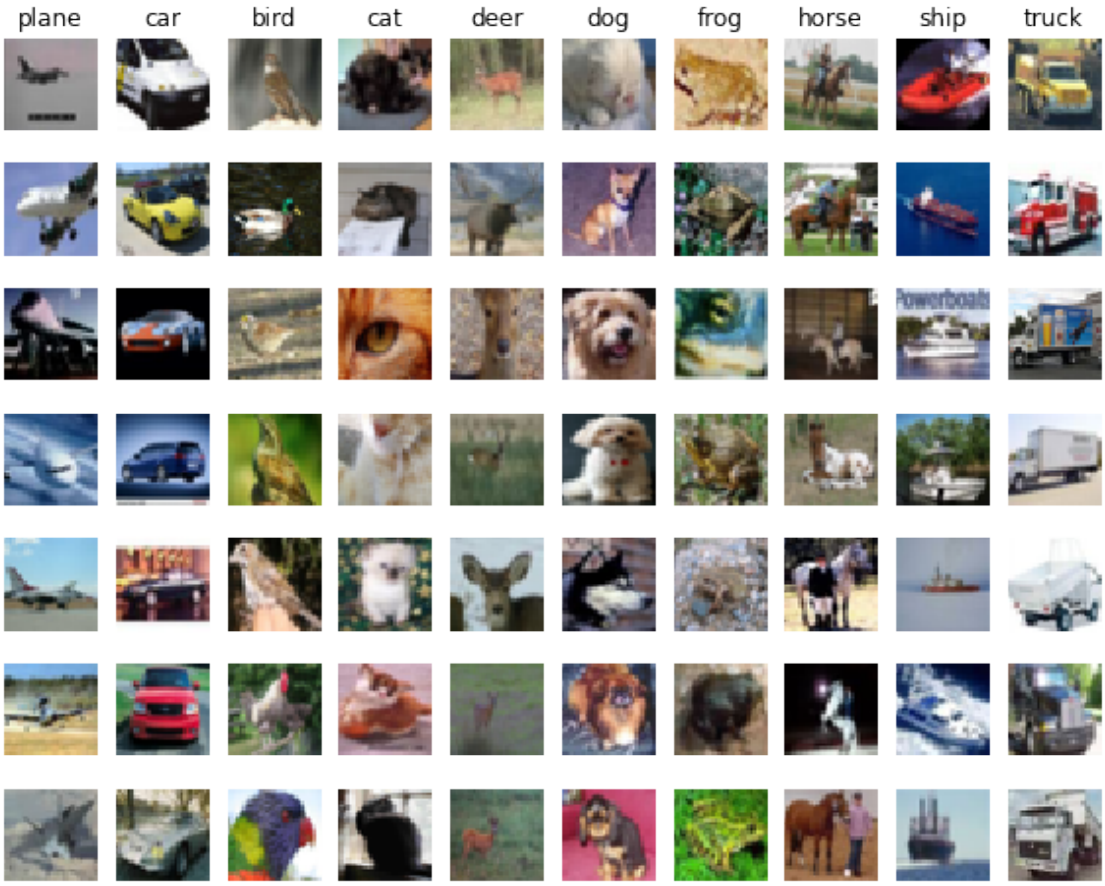
```
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

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

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

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

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

[4]:
```
# 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()
```



[5]:
```
# 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)
```

[6]:
```
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)
```

kNN

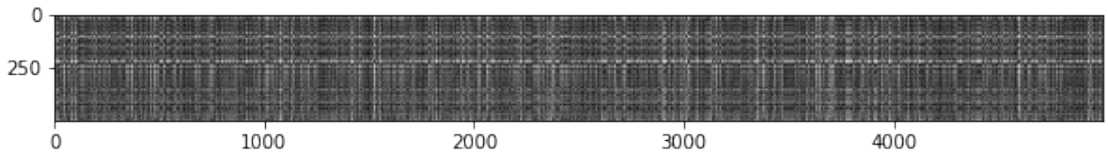
1.
2. k

cs231n/classifiers/k_nearest_neighbor.py compute_distances_two_loops

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

[9]:
```
# 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

Your Answer :

[10]:
```
# 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`:

[11]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

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

$Your Answer:$1,2,3

YourExplanation :

1:L1

2:L1

3:1L1

4:L1

5:L2L1

[12]:
```
# 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
```

[13]:
```
# 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
```

[14]:
```
# 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 40.294217 seconds
One loop version took 28.044386 seconds
No loop version took 0.427388 seconds
```

0.0.1 Cross-validation

k

[15]:
```
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)*****
# num_flods
X_train_folds = np.array_split(X_train,num_folds)
y_train_folds = np.array_split(y_train,num_folds)
pass
```

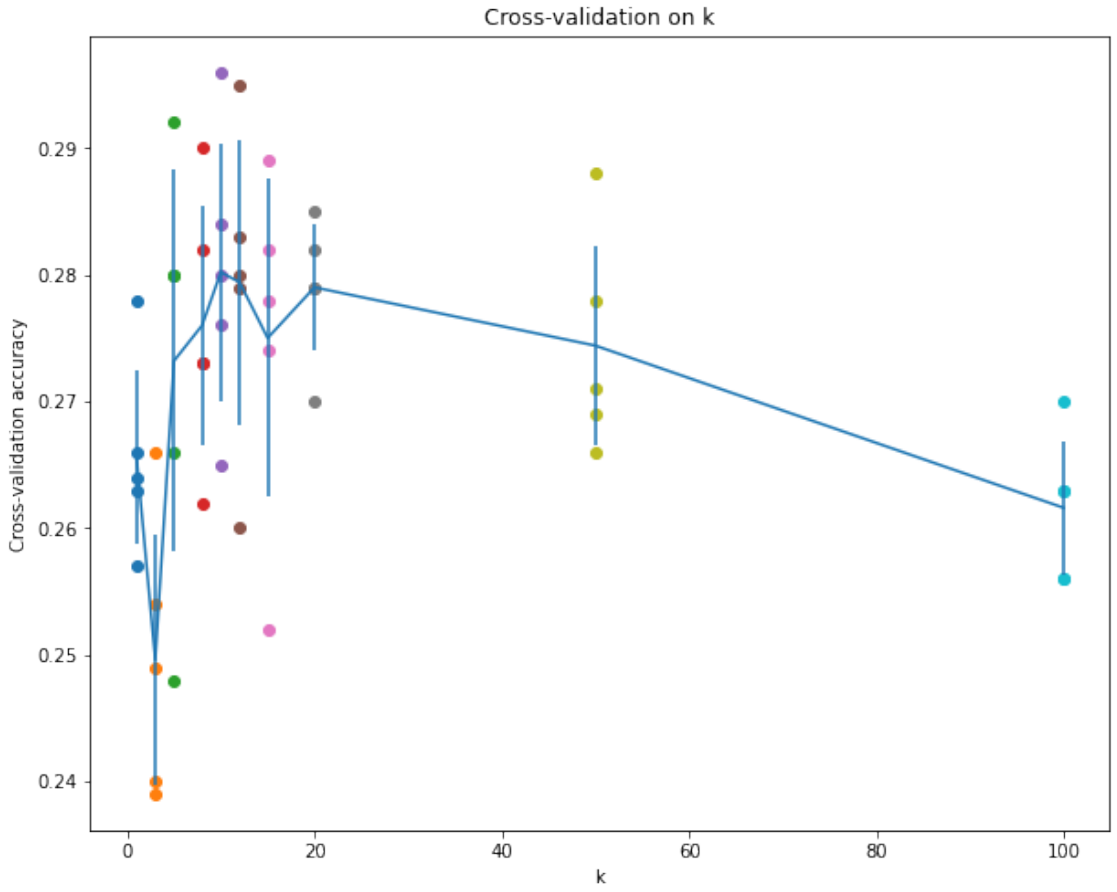
```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 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:
    # num_folds
    k_to_accuracies[k] = np.zeros(num_folds)
    acc = []
    for i in range(num_folds):
        # i,
        X_tr = X_train_folds[0:i] + X_train_folds[i+1:]
        y_tr = y_train_folds[0:i] + y_train_folds[i+1:]
        # concatenatenum_folds - 1
        X_tr = np.concatenate(X_tr,axis=0)
        y_tr = np.concatenate(y_tr,axis=0)
        X_cv = X_train_folds[i]
        y_cv = y_train_folds[i]


        classifier = KNearestNeighbor()
        #
        classifier.train(X_tr,y_tr)
        #
        dists = classifier.compute_distances_no_loops(X_cv)
        #
```

```
        y_cv_pred = classifier.predict_labels(dists,k=k)
        #
        num_correct = np.mean(y_cv_pred == y_cv)

        acc.append(num_correct)
    k_to_accuracies[k] = acc
pass

# *****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
```

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



[18]:
```
# 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 = 10

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

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error

of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

$Your Answer:$4

YourExplanation :

1

21-NN 5-NN

32

4

[ ]:

svm

August 18, 2021

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

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

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

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

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

[10]:
```
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

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

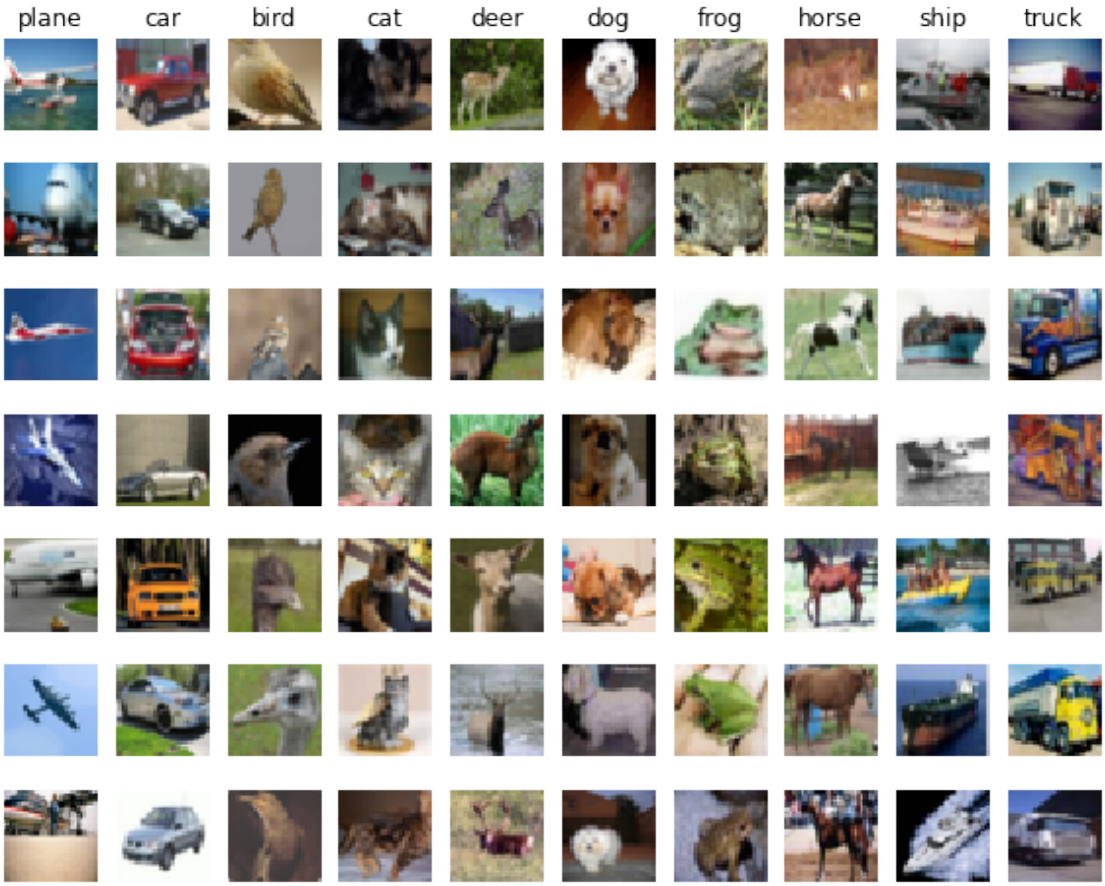
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.1 CIFAR-10 Data Loading and Preprocessing

[11]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

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

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

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

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

[12]:
```
# 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()
```



[13]:
```
# 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,)
```

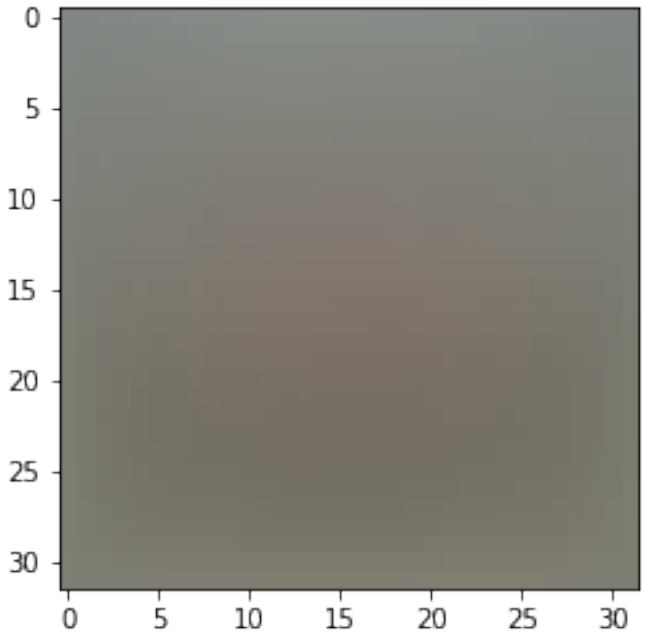
[14]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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)
```

[15]:
```
# 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)
```

softmax

August 18, 2021

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

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

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

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

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

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

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

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

[4]:
```
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`.

[12]:
```
# 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: 2.315162
sanity check: 2.302585
```

Inline Question 1
Why do we expect our loss to be close to -log(0.1)? Explain briefly.**
Your Answer : W 0.1 -log0.1

[13]:
```
# 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: 3.355374 analytic: 3.355374, relative error: 1.807112e-10
numerical: 0.909450 analytic: 0.909450, relative error: 1.324868e-08
numerical: 1.849160 analytic: 1.849160, relative error: 3.264310e-09
numerical: -1.233032 analytic: -1.233032, relative error: 3.696433e-08
numerical: -0.682402 analytic: -0.682402, relative error: 3.304693e-08
numerical: 1.367303 analytic: 1.367303, relative error: 1.075631e-08
```

```
numerical: -0.266831 analytic: -0.266831, relative error: 8.189308e-09
numerical: 1.691008 analytic: 1.691008, relative error: 5.143088e-08
numerical: -1.592797 analytic: -1.592798, relative error: 2.961665e-08
numerical: -2.332568 analytic: -2.332568, relative error: 4.355729e-09
numerical: 3.824821 analytic: 3.824821, relative error: 2.882612e-08
numerical: -0.237360 analytic: -0.237360, relative error: 1.280823e-07
numerical: -0.092034 analytic: -0.092034, relative error: 4.340122e-09
numerical: -0.152978 analytic: -0.152978, relative error: 2.710958e-08
numerical: 0.196063 analytic: 0.196063, relative error: 1.821423e-07
numerical: -0.341546 analytic: -0.341546, relative error: 6.468527e-08
numerical: 0.661372 analytic: 0.661372, relative error: 4.278568e-09
numerical: 1.934688 analytic: 1.934688, relative error: 1.165699e-08
numerical: 0.200524 analytic: 0.200524, relative error: 2.953361e-07
numerical: -0.557184 analytic: -0.557184, relative error: 2.410310e-08
```

[15]:
```
# 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: 2.315162e+00 computed in 0.167752s
vectorized loss: 2.315162e+00 computed in 0.013871s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[18]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
```

```
# get a classification accuracy of over 0.35 on the validation set.

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

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

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

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for lr in learning_rates:
  for reg in regularization_strengths:
    softmax = Softmax()
    softmax.train(X_train, y_train, lr, reg, num_iters = 3000)
    y_train_pred = softmax.predict(X_train)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = softmax.predict(X_val)
    val_accuracy = np.mean(y_val == y_val_pred)
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_softmax = softmax
    results[(lr, reg)] = train_accuracy, val_accuracy
pass

# *****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: 0.332918 val accuracy: 0.345000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.304388 val accuracy: 0.321000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.328510 val accuracy: 0.347000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.299020 val accuracy: 0.309000
best validation accuracy achieved during cross-validation: 0.347000
```

[19]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.343000
```

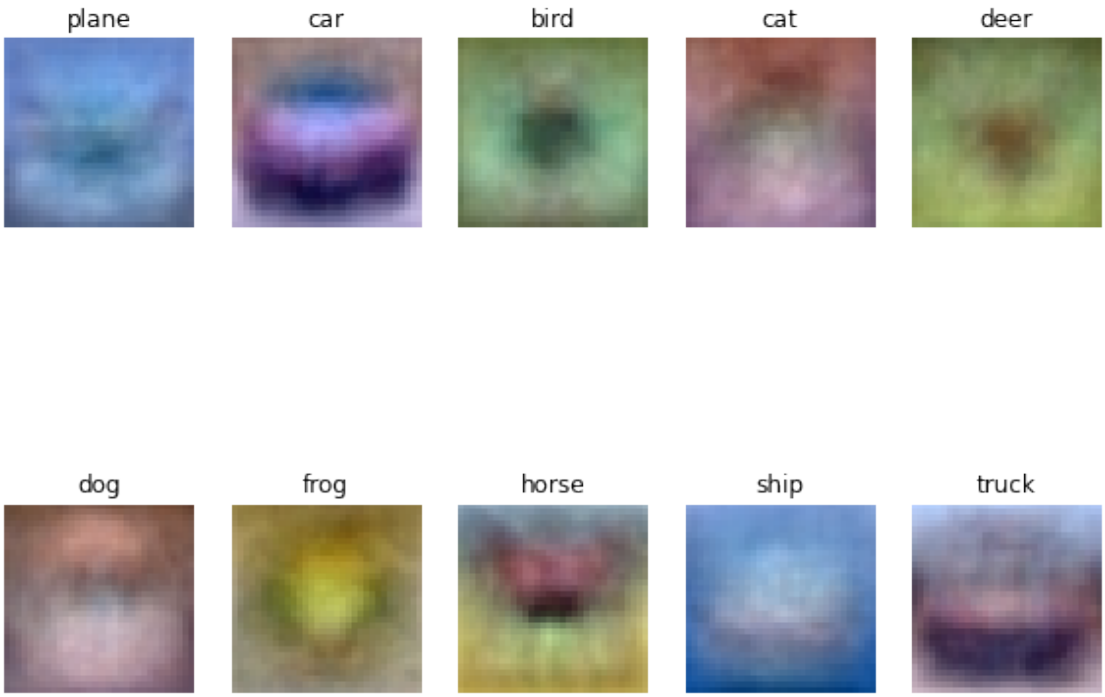
Inline Question 2 - *True or False*

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.

YourAnswer :

YourExplanation :SVMloss0loss; softmaxloss0loss

[20]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

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

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

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

[ ]:

two_layer_net

August 18, 2021

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

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

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

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

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

1 Fully-Connected Neural Nets

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

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

```
  # Do some more computations ...
  out = # the output

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

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

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

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

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

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

[ ]:
```
# 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))
print(out)
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
[[1.49834967 1.70660132 1.91485297]
 [3.25553199 3.5141327  3.77273342]]
```

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:

[1,2 sigmoid0relu00]

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 sum_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: 7.98e-08
b1 relative error: 1.35e-08
b2 relative error: 7.76e-10
```

9 Solver

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

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

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
solver = Solver(model,data,
                num_epochs = 30,
                update_rule = 'sgd',
                optim_config = {
                    'learning_rate': 1e-3
                },
                lr_decay = 0.95,
                batch_size = 128,
                print_every = 100
               )
solver.train()


pass

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

```
5000
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Epoch 28 / 30) train acc: 0.615000; val_acc: 0.508000
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 10701 / 11460) loss: 1.041259
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 10801 / 11460) loss: 1.085886
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 10901 / 11460) loss: 0.996730
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 11001 / 11460) loss: 0.939815
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
```

```
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Epoch 29 / 30) train acc: 0.644000; val_acc: 0.502000
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
```

```
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 11101 / 11460) loss: 1.166177
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
```

```
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 11201 / 11460) loss: 1.159367
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
```

```
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 11301 / 11460) loss: 1.102351
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
```

```
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Iteration 11401 / 11460) loss: 0.928256
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
```

```
(50, 10)
(10,)
(50,) 1
(10,)
(3072, 50)
(50,)
(50, 10)
(10,)
(Epoch 30 / 30) train acc: 0.652000; val_acc: 0.510000
```

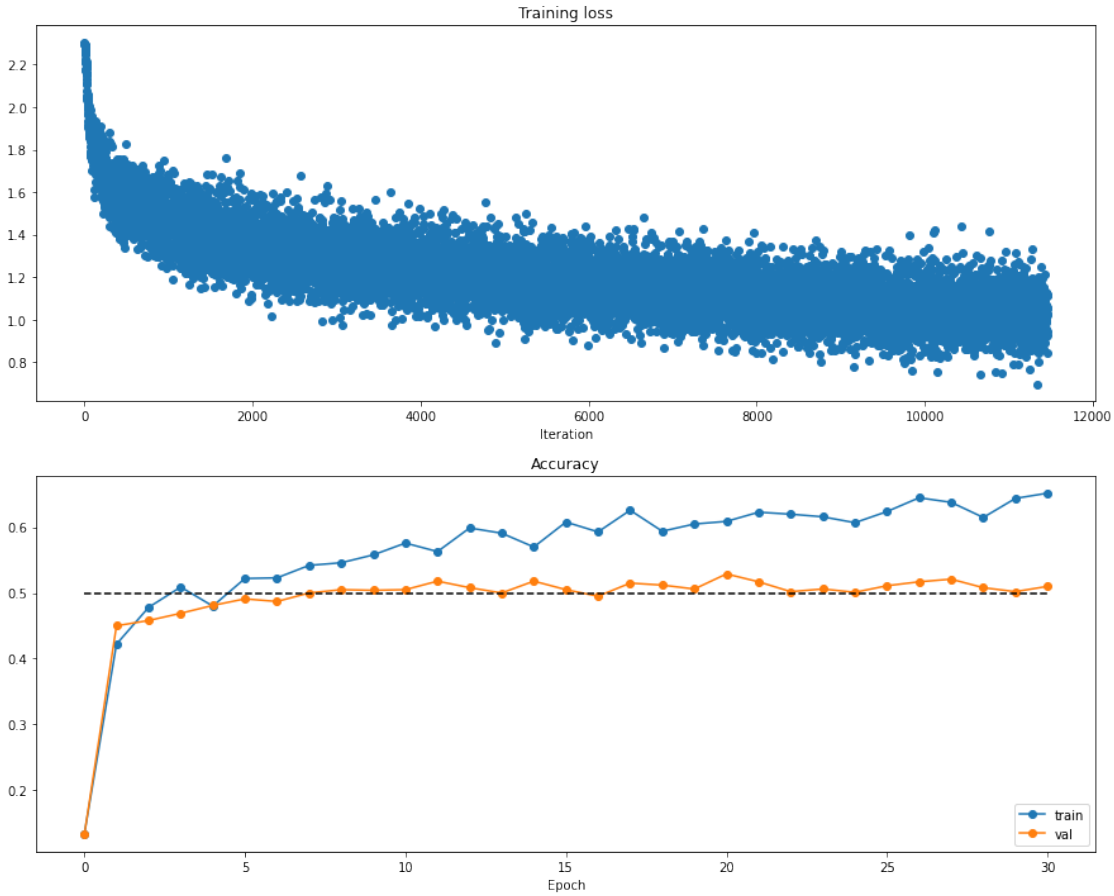
10 Debug the training

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

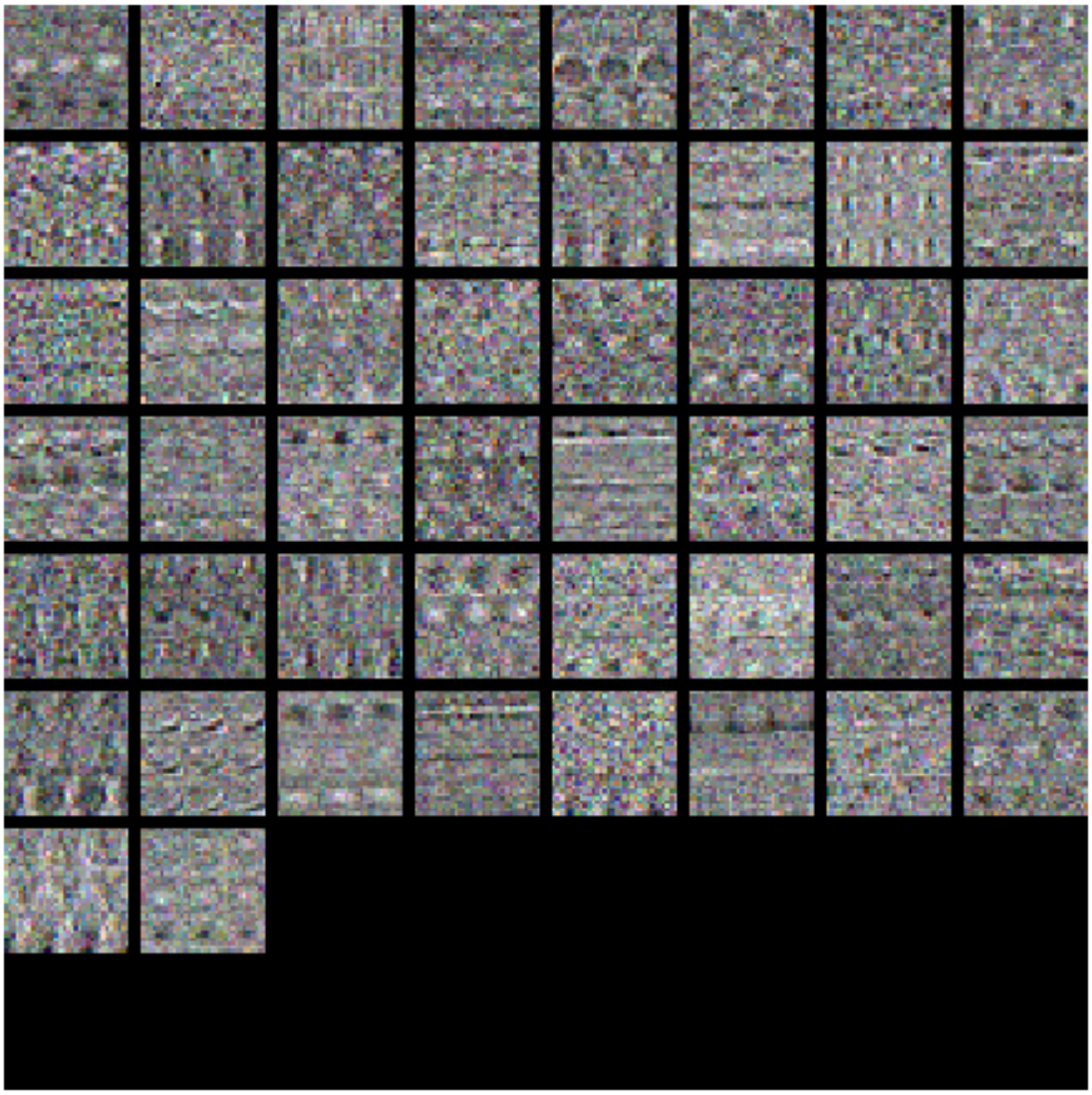
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

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

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

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



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

# Visualize the weights of the network

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

show_net_weights(model)
```



11 Tune your hyperparameters

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

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

also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

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

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

[ ]:
```
best_net = None # store the best model into this
################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
 ↪#
# model in best_net.                                                           ␣
 ↪#
#                                                                              ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative   ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
 ↪#
#                                                                              ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters         ␣
 ↪#
# automatically like we did on the previous exercises.                         ␣
 ↪#
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

input_size = 32 * 32 * 3
hidden_dim = 100
num_classes = 10

net1 = TwoLayerNet(input_size, hidden_dim, num_classes)
# Train the network
solver = Solver(net1,data,
                num_epochs = 30,
                update_rule = 'sgd',
                optim_config = {
                    'learning_rate': 1e-3
                },
                lr_decay = 0.95,
```

```
                batch_size = 128,
                print_every = 100
               )
solver.train()


pass

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

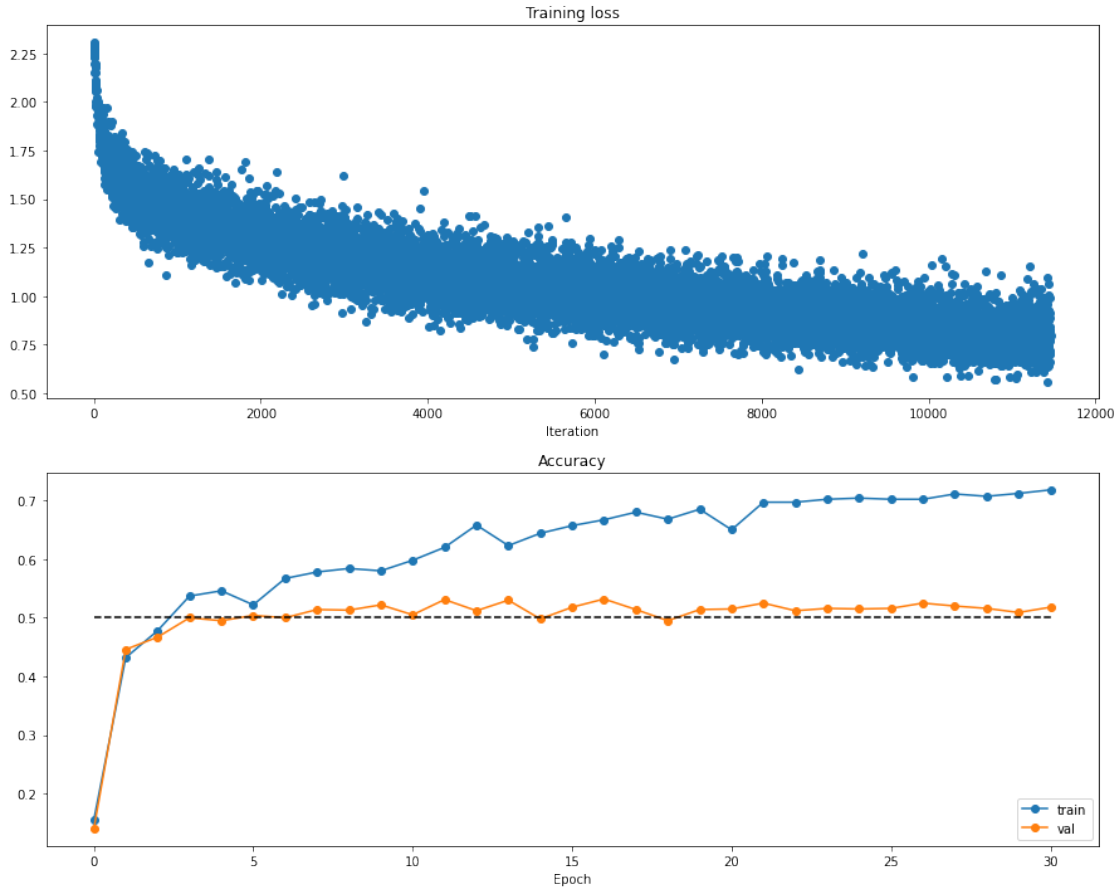
```
(Iteration 1 / 11460) loss: 2.308033
(Epoch 0 / 30) train acc: 0.155000; val_acc: 0.140000
(Iteration 101 / 11460) loss: 1.880864
(Iteration 201 / 11460) loss: 1.631505
(Iteration 301 / 11460) loss: 1.626828
(Epoch 1 / 30) train acc: 0.432000; val_acc: 0.446000
(Iteration 401 / 11460) loss: 1.645574
(Iteration 501 / 11460) loss: 1.538218
(Iteration 601 / 11460) loss: 1.575305
(Iteration 701 / 11460) loss: 1.483644
(Epoch 2 / 30) train acc: 0.478000; val_acc: 0.467000
(Iteration 801 / 11460) loss: 1.471914
(Iteration 901 / 11460) loss: 1.612547
(Iteration 1001 / 11460) loss: 1.307263
(Iteration 1101 / 11460) loss: 1.488253
(Epoch 3 / 30) train acc: 0.537000; val_acc: 0.500000
(Iteration 1201 / 11460) loss: 1.256703
(Iteration 1301 / 11460) loss: 1.405606
(Iteration 1401 / 11460) loss: 1.278710
(Iteration 1501 / 11460) loss: 1.357907
(Epoch 4 / 30) train acc: 0.546000; val_acc: 0.495000
(Iteration 1601 / 11460) loss: 1.188716
(Iteration 1701 / 11460) loss: 1.374608
(Iteration 1801 / 11460) loss: 1.305599
(Iteration 1901 / 11460) loss: 1.299954
(Epoch 5 / 30) train acc: 0.522000; val_acc: 0.504000
(Iteration 2001 / 11460) loss: 1.162949
(Iteration 2101 / 11460) loss: 1.261244
(Iteration 2201 / 11460) loss: 1.416487
(Epoch 6 / 30) train acc: 0.567000; val_acc: 0.500000
(Iteration 2301 / 11460) loss: 1.221083
(Iteration 2401 / 11460) loss: 1.317491
(Iteration 2501 / 11460) loss: 1.266476
```

```
(Iteration 2601 / 11460) loss: 1.183092
(Epoch 7 / 30) train acc: 0.578000; val_acc: 0.514000
(Iteration 2701 / 11460) loss: 1.166111
(Iteration 2801 / 11460) loss: 1.214418
(Iteration 2901 / 11460) loss: 1.085149
(Iteration 3001 / 11460) loss: 1.177323
(Epoch 8 / 30) train acc: 0.584000; val_acc: 0.513000
(Iteration 3101 / 11460) loss: 1.194932
(Iteration 3201 / 11460) loss: 1.072525
(Iteration 3301 / 11460) loss: 1.060404
(Iteration 3401 / 11460) loss: 1.104279
(Epoch 9 / 30) train acc: 0.580000; val_acc: 0.522000
(Iteration 3501 / 11460) loss: 1.106955
(Iteration 3601 / 11460) loss: 1.311488
(Iteration 3701 / 11460) loss: 1.388364
(Iteration 3801 / 11460) loss: 1.200945
(Epoch 10 / 30) train acc: 0.598000; val_acc: 0.505000
(Iteration 3901 / 11460) loss: 1.017804
(Iteration 4001 / 11460) loss: 1.046032
(Iteration 4101 / 11460) loss: 0.950209
(Iteration 4201 / 11460) loss: 1.176096
(Epoch 11 / 30) train acc: 0.620000; val_acc: 0.531000
(Iteration 4301 / 11460) loss: 1.072208
(Iteration 4401 / 11460) loss: 0.946625
(Iteration 4501 / 11460) loss: 1.008079
(Epoch 12 / 30) train acc: 0.658000; val_acc: 0.512000
(Iteration 4601 / 11460) loss: 1.176289
(Iteration 4701 / 11460) loss: 1.159015
(Iteration 4801 / 11460) loss: 1.204808
(Iteration 4901 / 11460) loss: 1.116052
(Epoch 13 / 30) train acc: 0.623000; val_acc: 0.530000
(Iteration 5001 / 11460) loss: 1.171049
(Iteration 5101 / 11460) loss: 0.968639
(Iteration 5201 / 11460) loss: 1.090662
(Iteration 5301 / 11460) loss: 0.864923
(Epoch 14 / 30) train acc: 0.644000; val_acc: 0.498000
(Iteration 5401 / 11460) loss: 0.875622
(Iteration 5501 / 11460) loss: 0.913750
(Iteration 5601 / 11460) loss: 1.036842
(Iteration 5701 / 11460) loss: 1.015126
(Epoch 15 / 30) train acc: 0.657000; val_acc: 0.518000
(Iteration 5801 / 11460) loss: 0.998780
(Iteration 5901 / 11460) loss: 0.968589
(Iteration 6001 / 11460) loss: 0.980538
(Iteration 6101 / 11460) loss: 0.954642
(Epoch 16 / 30) train acc: 0.667000; val_acc: 0.532000
(Iteration 6201 / 11460) loss: 0.920786
(Iteration 6301 / 11460) loss: 1.071190
```

```
(Iteration 6401 / 11460) loss: 1.007807
(Epoch 17 / 30) train acc: 0.680000; val_acc: 0.514000
(Iteration 6501 / 11460) loss: 0.986627
(Iteration 6601 / 11460) loss: 0.867931
(Iteration 6701 / 11460) loss: 0.981690
(Iteration 6801 / 11460) loss: 1.082519
(Epoch 18 / 30) train acc: 0.668000; val_acc: 0.495000
(Iteration 6901 / 11460) loss: 0.936332
(Iteration 7001 / 11460) loss: 0.865492
(Iteration 7101 / 11460) loss: 0.844417
(Iteration 7201 / 11460) loss: 0.866526
(Epoch 19 / 30) train acc: 0.685000; val_acc: 0.514000
(Iteration 7301 / 11460) loss: 1.087729
(Iteration 7401 / 11460) loss: 1.047538
(Iteration 7501 / 11460) loss: 1.004426
(Iteration 7601 / 11460) loss: 0.956205
(Epoch 20 / 30) train acc: 0.650000; val_acc: 0.515000
(Iteration 7701 / 11460) loss: 1.061666
(Iteration 7801 / 11460) loss: 0.976049
(Iteration 7901 / 11460) loss: 0.741305
(Iteration 8001 / 11460) loss: 0.884274
(Epoch 21 / 30) train acc: 0.697000; val_acc: 0.525000
(Iteration 8101 / 11460) loss: 0.847551
(Iteration 8201 / 11460) loss: 0.846074
(Iteration 8301 / 11460) loss: 0.826019
(Iteration 8401 / 11460) loss: 0.839390
(Epoch 22 / 30) train acc: 0.697000; val_acc: 0.512000
(Iteration 8501 / 11460) loss: 0.764874
(Iteration 8601 / 11460) loss: 0.885150
(Iteration 8701 / 11460) loss: 0.940178
(Epoch 23 / 30) train acc: 0.702000; val_acc: 0.516000
(Iteration 8801 / 11460) loss: 0.853836
(Iteration 8901 / 11460) loss: 1.000228
(Iteration 9001 / 11460) loss: 0.785990
(Iteration 9101 / 11460) loss: 0.923716
(Epoch 24 / 30) train acc: 0.704000; val_acc: 0.515000
(Iteration 9201 / 11460) loss: 0.914342
(Iteration 9301 / 11460) loss: 0.932678
(Iteration 9401 / 11460) loss: 0.952509
(Iteration 9501 / 11460) loss: 0.917636
(Epoch 25 / 30) train acc: 0.702000; val_acc: 0.516000
(Iteration 9601 / 11460) loss: 0.889835
(Iteration 9701 / 11460) loss: 0.891476
(Iteration 9801 / 11460) loss: 0.842381
(Iteration 9901 / 11460) loss: 0.729633
(Epoch 26 / 30) train acc: 0.702000; val_acc: 0.525000
(Iteration 10001 / 11460) loss: 0.856182
(Iteration 10101 / 11460) loss: 0.848120
```

```
(Iteration 10201 / 11460) loss: 0.842721
(Iteration 10301 / 11460) loss: 0.943881
(Epoch 27 / 30) train acc: 0.711000; val_acc: 0.520000
(Iteration 10401 / 11460) loss: 0.846666
(Iteration 10501 / 11460) loss: 0.722797
(Iteration 10601 / 11460) loss: 0.849899
(Epoch 28 / 30) train acc: 0.707000; val_acc: 0.516000
(Iteration 10701 / 11460) loss: 0.831273
(Iteration 10801 / 11460) loss: 0.805012
(Iteration 10901 / 11460) loss: 0.889505
(Iteration 11001 / 11460) loss: 0.699593
(Epoch 29 / 30) train acc: 0.712000; val_acc: 0.509000
(Iteration 11101 / 11460) loss: 0.824407
(Iteration 11201 / 11460) loss: 0.876137
(Iteration 11301 / 11460) loss: 0.849739
(Iteration 11401 / 11460) loss: 0.793733
(Epoch 30 / 30) train acc: 0.718000; val_acc: 0.518000
```

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



[ ]:

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(net1.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.532
```

[ ]:
```
y_test_pred = np.argmax(net1.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.521
```

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:$1, 3
$Your Explanation:$1
2
3

[ ]:

features

August 18, 2021

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

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

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

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

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

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

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

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

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

1.1 Load data

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

[3]:
```
from cs231n.features import color_histogram_hsv, hog_feature

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

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

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

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

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

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

1.2 Extract Features

(HOG) HSV HOG
HOG
hog_feature color_histogram_hsv

[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.

[ ]:
```
# 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 rs in regularization_strengths:
  for lr in learning_rates:
    svm = LinearSVM()
    svm.train(X_train_feats, y_train, lr, rs, num_iters = 3000)
    y_train_pred = svm.predict(X_train_feats)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = svm.predict(X_val_feats)
    val_accuracy = np.mean(y_val == y_val_pred)
    if val_accuracy > best_val:
      best_val = val_accuracy
      best_svm = svm
    results[(lr, rs)] = train_accuracy, val_accuracy
pass

# *****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)
```

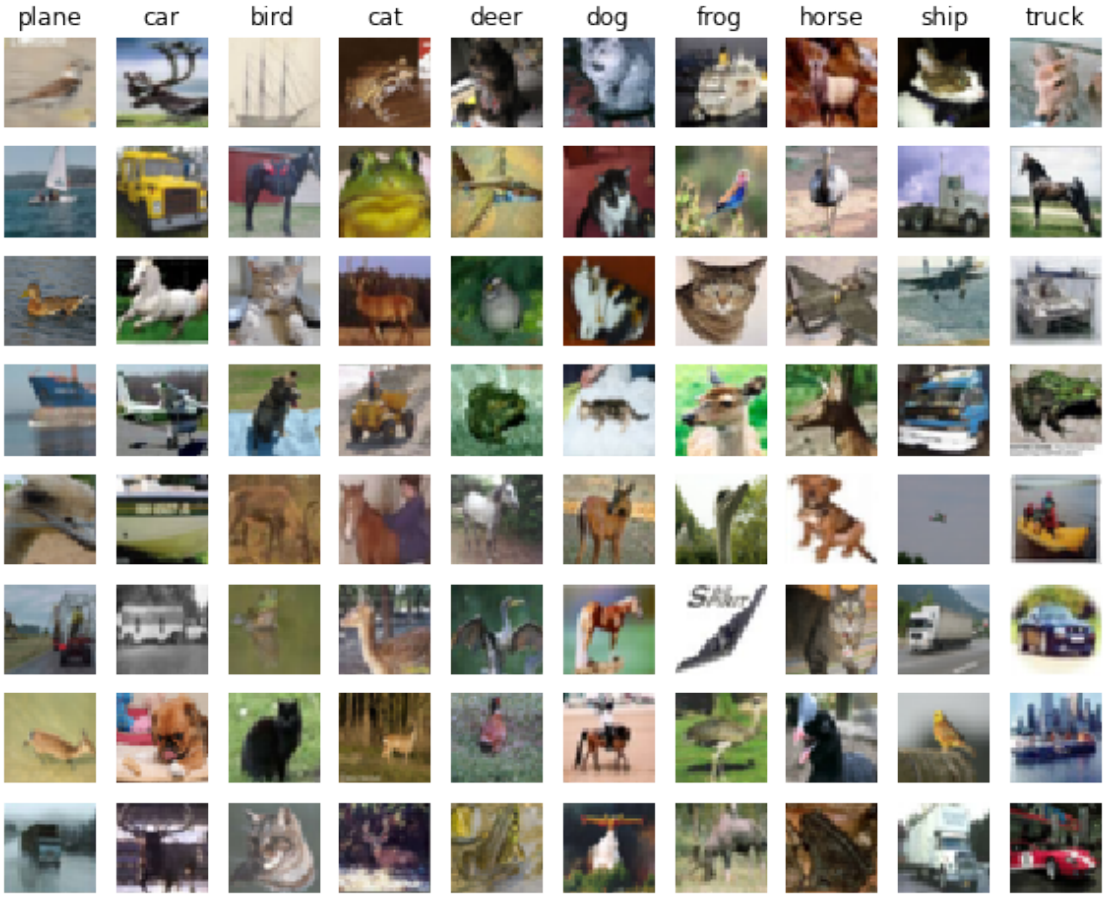
```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.104122 val accuracy: 0.093000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.095143 val accuracy: 0.090000
```

```
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.414898 val accuracy: 0.420000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.140347 val accuracy: 0.134000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.413837 val accuracy: 0.419000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.401612 val accuracy: 0.392000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.415306 val accuracy: 0.420000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.406755 val accuracy: 0.407000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.325163 val accuracy: 0.310000
best validation accuracy achieved: 0.420000
```

[ ]:
```
# 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)
```

```
0.42
```

[ ]:
```
# 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 :

1.4 Neural Network on image features

55% 60%

[6]:
```
# 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)
```

[17]:
```
from cs231n.classifiers.fc_net import TwoLayerNet
from cs231n.solver import Solver

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

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None
#data = [X_train_feats, y_train, X_val_feats, y_val, X_test_feats, y_test]
################################################################################
# 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)*****

# Train the network
solver = Solver(net,data,
                num_epochs = 30,
                update_rule = 'sgd',
                optim_config = {
                    'learning_rate': 1e-1
                },
                lr_decay = 0.95,
                batch_size = 128,
                print_every = 100
               )
solver.train()

pass

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

```
(Iteration 1 / 11460) loss: 2.302577
(Epoch 0 / 30) train acc: 0.112000; val_acc: 0.118000
(Iteration 101 / 11460) loss: 2.216280
(Iteration 201 / 11460) loss: 1.811840
(Iteration 301 / 11460) loss: 1.591681
(Epoch 1 / 30) train acc: 0.454000; val_acc: 0.464000
(Iteration 401 / 11460) loss: 1.353552
```

```
(Iteration 501 / 11460) loss: 1.308354
(Iteration 601 / 11460) loss: 1.325722
(Iteration 701 / 11460) loss: 1.390743
(Epoch 2 / 30) train acc: 0.502000; val_acc: 0.505000
(Iteration 801 / 11460) loss: 1.318039
(Iteration 901 / 11460) loss: 1.266553
(Iteration 1001 / 11460) loss: 1.336841
(Iteration 1101 / 11460) loss: 1.298824
(Epoch 3 / 30) train acc: 0.549000; val_acc: 0.523000
(Iteration 1201 / 11460) loss: 1.324211
(Iteration 1301 / 11460) loss: 1.261865
(Iteration 1401 / 11460) loss: 1.405834
(Iteration 1501 / 11460) loss: 1.365341
(Epoch 4 / 30) train acc: 0.571000; val_acc: 0.533000
(Iteration 1601 / 11460) loss: 1.303444
(Iteration 1701 / 11460) loss: 1.130615
(Iteration 1801 / 11460) loss: 1.318906
(Iteration 1901 / 11460) loss: 1.344625
(Epoch 5 / 30) train acc: 0.586000; val_acc: 0.548000
(Iteration 2001 / 11460) loss: 1.140704
(Iteration 2101 / 11460) loss: 1.277661
(Iteration 2201 / 11460) loss: 1.096596
(Epoch 6 / 30) train acc: 0.581000; val_acc: 0.556000
(Iteration 2301 / 11460) loss: 1.261517
(Iteration 2401 / 11460) loss: 1.143507
(Iteration 2501 / 11460) loss: 1.166270
(Iteration 2601 / 11460) loss: 1.195310
(Epoch 7 / 30) train acc: 0.587000; val_acc: 0.568000
(Iteration 2701 / 11460) loss: 1.214881
(Iteration 2801 / 11460) loss: 1.135423
(Iteration 2901 / 11460) loss: 1.252509
(Iteration 3001 / 11460) loss: 1.043810
(Epoch 8 / 30) train acc: 0.612000; val_acc: 0.572000
(Iteration 3101 / 11460) loss: 0.995083
(Iteration 3201 / 11460) loss: 0.901438
(Iteration 3301 / 11460) loss: 1.001180
(Iteration 3401 / 11460) loss: 1.089491
(Epoch 9 / 30) train acc: 0.632000; val_acc: 0.591000
(Iteration 3501 / 11460) loss: 1.101599
(Iteration 3601 / 11460) loss: 1.007513
(Iteration 3701 / 11460) loss: 1.014693
(Iteration 3801 / 11460) loss: 1.055264
(Epoch 10 / 30) train acc: 0.644000; val_acc: 0.587000
(Iteration 3901 / 11460) loss: 1.055186
(Iteration 4001 / 11460) loss: 1.076214
(Iteration 4101 / 11460) loss: 1.079289
(Iteration 4201 / 11460) loss: 1.056375
(Epoch 11 / 30) train acc: 0.651000; val_acc: 0.585000
```

```
(Iteration 4301 / 11460) loss: 0.928836
(Iteration 4401 / 11460) loss: 0.885425
(Iteration 4501 / 11460) loss: 0.882382
(Epoch 12 / 30) train acc: 0.650000; val_acc: 0.585000
(Iteration 4601 / 11460) loss: 1.014534
(Iteration 4701 / 11460) loss: 0.980306
(Iteration 4801 / 11460) loss: 0.944443
(Iteration 4901 / 11460) loss: 0.926429
(Epoch 13 / 30) train acc: 0.681000; val_acc: 0.596000
(Iteration 5001 / 11460) loss: 1.000724
(Iteration 5101 / 11460) loss: 0.917415
(Iteration 5201 / 11460) loss: 0.926038
(Iteration 5301 / 11460) loss: 0.911793
(Epoch 14 / 30) train acc: 0.688000; val_acc: 0.590000
(Iteration 5401 / 11460) loss: 0.876594
(Iteration 5501 / 11460) loss: 0.966458
(Iteration 5601 / 11460) loss: 0.897514
(Iteration 5701 / 11460) loss: 0.915246
(Epoch 15 / 30) train acc: 0.692000; val_acc: 0.590000
(Iteration 5801 / 11460) loss: 0.885070
(Iteration 5901 / 11460) loss: 0.784102
(Iteration 6001 / 11460) loss: 0.897553
(Iteration 6101 / 11460) loss: 1.004714
(Epoch 16 / 30) train acc: 0.693000; val_acc: 0.582000
(Iteration 6201 / 11460) loss: 0.851781
(Iteration 6301 / 11460) loss: 0.883480
(Iteration 6401 / 11460) loss: 0.824778
(Epoch 17 / 30) train acc: 0.687000; val_acc: 0.590000
(Iteration 6501 / 11460) loss: 0.864027
(Iteration 6601 / 11460) loss: 0.844864
(Iteration 6701 / 11460) loss: 0.651566
(Iteration 6801 / 11460) loss: 0.870804
(Epoch 18 / 30) train acc: 0.700000; val_acc: 0.591000
(Iteration 6901 / 11460) loss: 0.982569
(Iteration 7001 / 11460) loss: 0.851629
(Iteration 7101 / 11460) loss: 0.878566
(Iteration 7201 / 11460) loss: 0.996685
(Epoch 19 / 30) train acc: 0.720000; val_acc: 0.601000
(Iteration 7301 / 11460) loss: 0.904910
(Iteration 7401 / 11460) loss: 0.889583
(Iteration 7501 / 11460) loss: 0.846406
(Iteration 7601 / 11460) loss: 0.914378
(Epoch 20 / 30) train acc: 0.705000; val_acc: 0.596000
(Iteration 7701 / 11460) loss: 0.866266
(Iteration 7801 / 11460) loss: 0.950564
(Iteration 7901 / 11460) loss: 0.777119
(Iteration 8001 / 11460) loss: 0.811205
(Epoch 21 / 30) train acc: 0.727000; val_acc: 0.600000
```

```
(Iteration 8101 / 11460) loss: 0.718027
(Iteration 8201 / 11460) loss: 0.878387
(Iteration 8301 / 11460) loss: 0.707713
(Iteration 8401 / 11460) loss: 0.956903
(Epoch 22 / 30) train acc: 0.715000; val_acc: 0.600000
(Iteration 8501 / 11460) loss: 1.044188
(Iteration 8601 / 11460) loss: 0.934253
(Iteration 8701 / 11460) loss: 0.830569
(Epoch 23 / 30) train acc: 0.743000; val_acc: 0.597000
(Iteration 8801 / 11460) loss: 0.891563
(Iteration 8901 / 11460) loss: 0.874410
(Iteration 9001 / 11460) loss: 0.740864
(Iteration 9101 / 11460) loss: 0.731308
(Epoch 24 / 30) train acc: 0.740000; val_acc: 0.592000
(Iteration 9201 / 11460) loss: 0.762761
(Iteration 9301 / 11460) loss: 0.778249
(Iteration 9401 / 11460) loss: 0.740520
(Iteration 9501 / 11460) loss: 0.904061
(Epoch 25 / 30) train acc: 0.743000; val_acc: 0.596000
(Iteration 9601 / 11460) loss: 0.801062
(Iteration 9701 / 11460) loss: 0.700198
(Iteration 9801 / 11460) loss: 0.716105
(Iteration 9901 / 11460) loss: 0.639859
(Epoch 26 / 30) train acc: 0.738000; val_acc: 0.615000
(Iteration 10001 / 11460) loss: 0.672990
(Iteration 10101 / 11460) loss: 0.788654
(Iteration 10201 / 11460) loss: 0.789167
(Iteration 10301 / 11460) loss: 0.740336
(Epoch 27 / 30) train acc: 0.753000; val_acc: 0.597000
(Iteration 10401 / 11460) loss: 0.797727
(Iteration 10501 / 11460) loss: 0.801571
(Iteration 10601 / 11460) loss: 0.812141
(Epoch 28 / 30) train acc: 0.739000; val_acc: 0.599000
(Iteration 10701 / 11460) loss: 0.760439
(Iteration 10801 / 11460) loss: 0.730743
(Iteration 10901 / 11460) loss: 0.824021
(Iteration 11001 / 11460) loss: 0.708976
(Epoch 29 / 30) train acc: 0.770000; val_acc: 0.599000
(Iteration 11101 / 11460) loss: 0.729489
(Iteration 11201 / 11460) loss: 0.711370
(Iteration 11301 / 11460) loss: 0.771306
(Iteration 11401 / 11460) loss: 0.600927
(Epoch 30 / 30) train acc: 0.786000; val_acc: 0.591000
```

[14]:
```
data = {}
data["X_train"] = X_train_feats
data["y_train"] = y_train
```

```
data["X_val"] = X_val_feats
data["y_val"] = y_val
data["X_test"] = X_test_feats
data["y_test"] = y_test
```

[19]:
```
# 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(net.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.6
```