```
[info] ROOT=/home/kim1484/assignment1_colab/assignment1
[info] DATA_DIR=/home/kim1484/assignment1_colab/assignment1/cs231n/datasets
[ready] 환경 준비 완료.
```

k-Nearest Neighbor (kNN) exercise

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

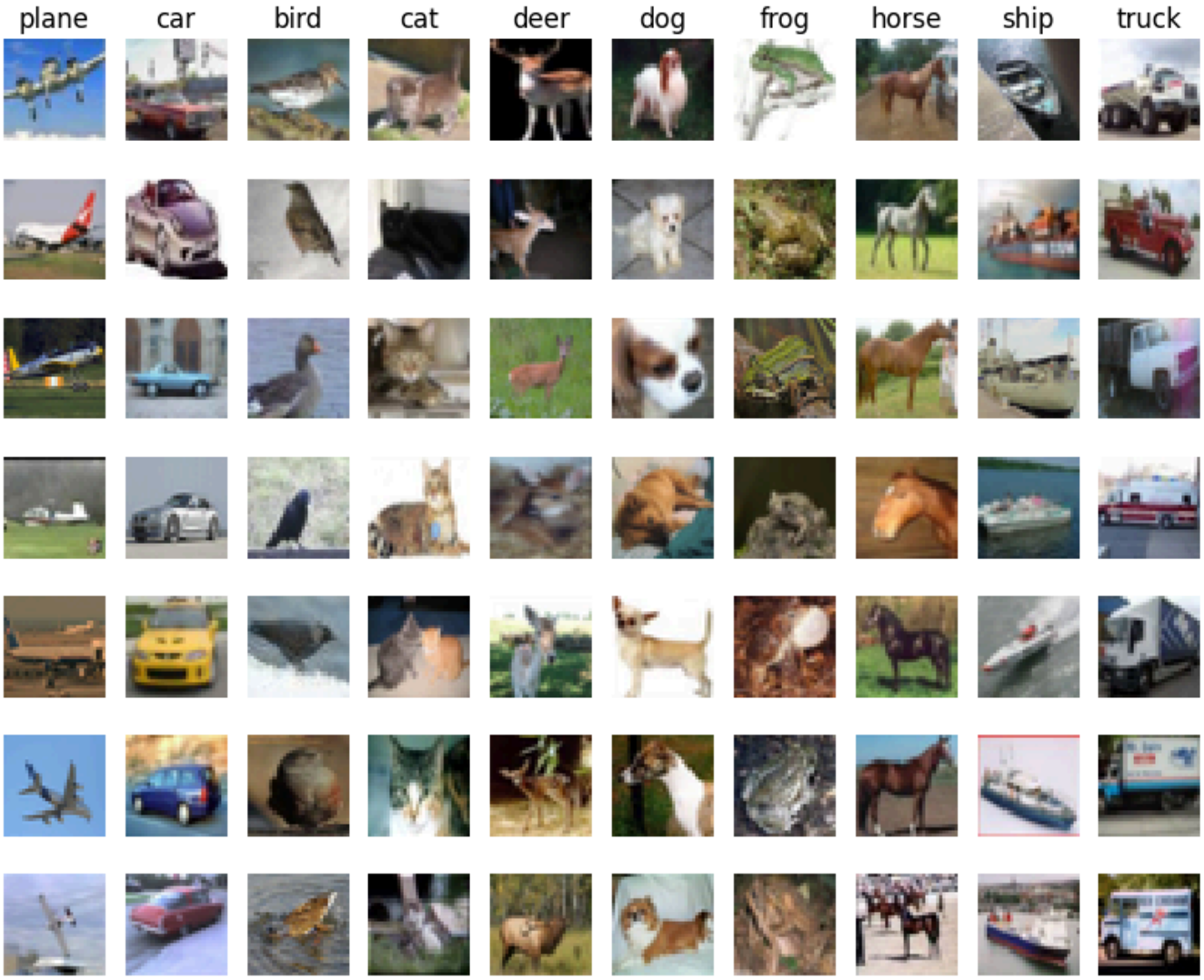
The kNN classifier consists of two stages:

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

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

```
[ready] Visualization & autoreload settings complete.
```

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




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

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

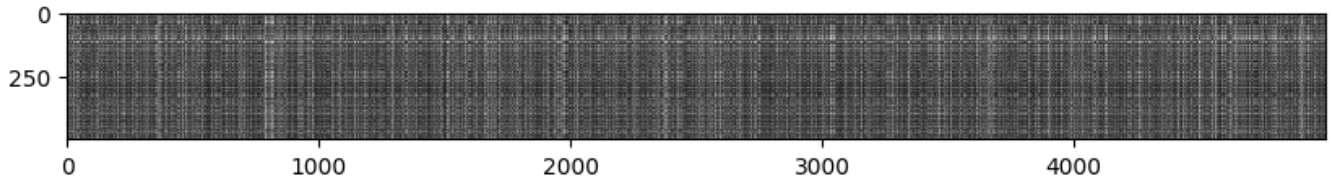
1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

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

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

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

```
(500, 5000)
```



Inline Question 1

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

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

$\mathit{Your Answer}$: *The bright rows indicate that the i-th test sample has large distance from most training samples. While bright columns mean that the j-th train sample is far from most testing samples.*

```
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`:

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

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

Inline Question 2

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

the mean μ across all pixels over all images is

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

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

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

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

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. To clarify, both training and test examples are preprocessed in the same way.

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

Your Answer : 1, 2

Your Explanation : Because L1 distance is based on absolute difference, subtracting the mean does not change the distances between data points. Which means there will be no changes in the performace of a Nearest Neighbor classifier

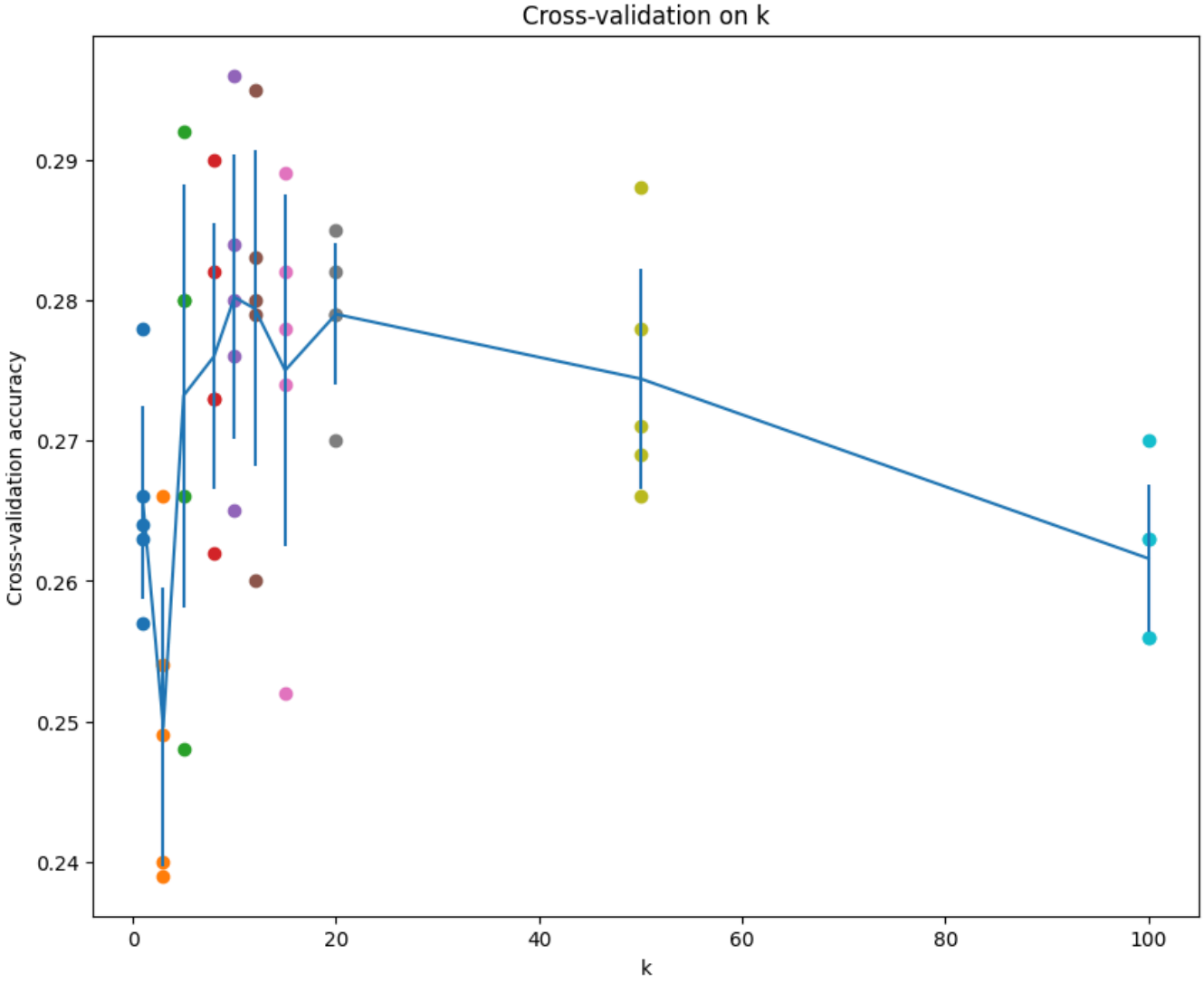
```
One loop difference was: 0.000000
Good! The distance matrices are the same

(500, 5000)
No loop difference was: 0.000000
Good! The distance matrices are the same

Two loop version took 11.938408 seconds
One loop version took 23.488079 seconds
No loop version took 0.082033 seconds
```

Cross-validation

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

{1: [0.263, 0.257, 0.264, 0.278, 0.266], 3: [0.239, 0.249, 0.24, 0.266, 0.254], 5: [0.248, 0.266, 0.28, 0.292, 0.28], 8: [0.262, 0.282, 0.273, 0.29, 0.273], 10: [0.265, 0.296, 0.276, 0.284, 0.28], 12: [0.26, 0.295, 0.279, 0.283, 0.28], 15: [0.252, 0.289, 0.278, 0.282, 0.274], 20: [0.27, 0.279, 0.279, 0.282, 0.285], 50: [0.271, 0.288, 0.278, 0.269, 0.266], 100: [0.256, 0.27, 0.263, 0.256, 0.263]}
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

```
[0.2656 0.2496 0.2732 0.276  0.2802 0.2794 0.275  0.279  0.2744 0.2616]
```




```
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 : 2, 4

Your Explanation :

1. **False**: The decision boundary of the k-NN classifier is determined by the data distribution, which is not linear.
2. **True**: If the k-NN classifier is trained with k=1, it will be overfitted to the training data. Therefore, the training error of 1-NN will always be lower than or equal to that of 5-NN.
3. **False**: A 1-NN classifier tends to overfit to the training data and does not generalize well to unseen test data. Therefore, the test error of 1-NN is higher than that of 5-NN.
4. **True**: Since the distance matrix must be computed between the training and test data, the computation time increases proportionally with the size of the training set.

```
[info] ROOT=/home/kim1484/assignment1_colab/assignment1
[info] DATA_DIR=/home/kim1484/assignment1_colab/assignment1/cs231n/datasets
[ready] 환경 준비 완료.
```

Softmax Classifier exercise

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

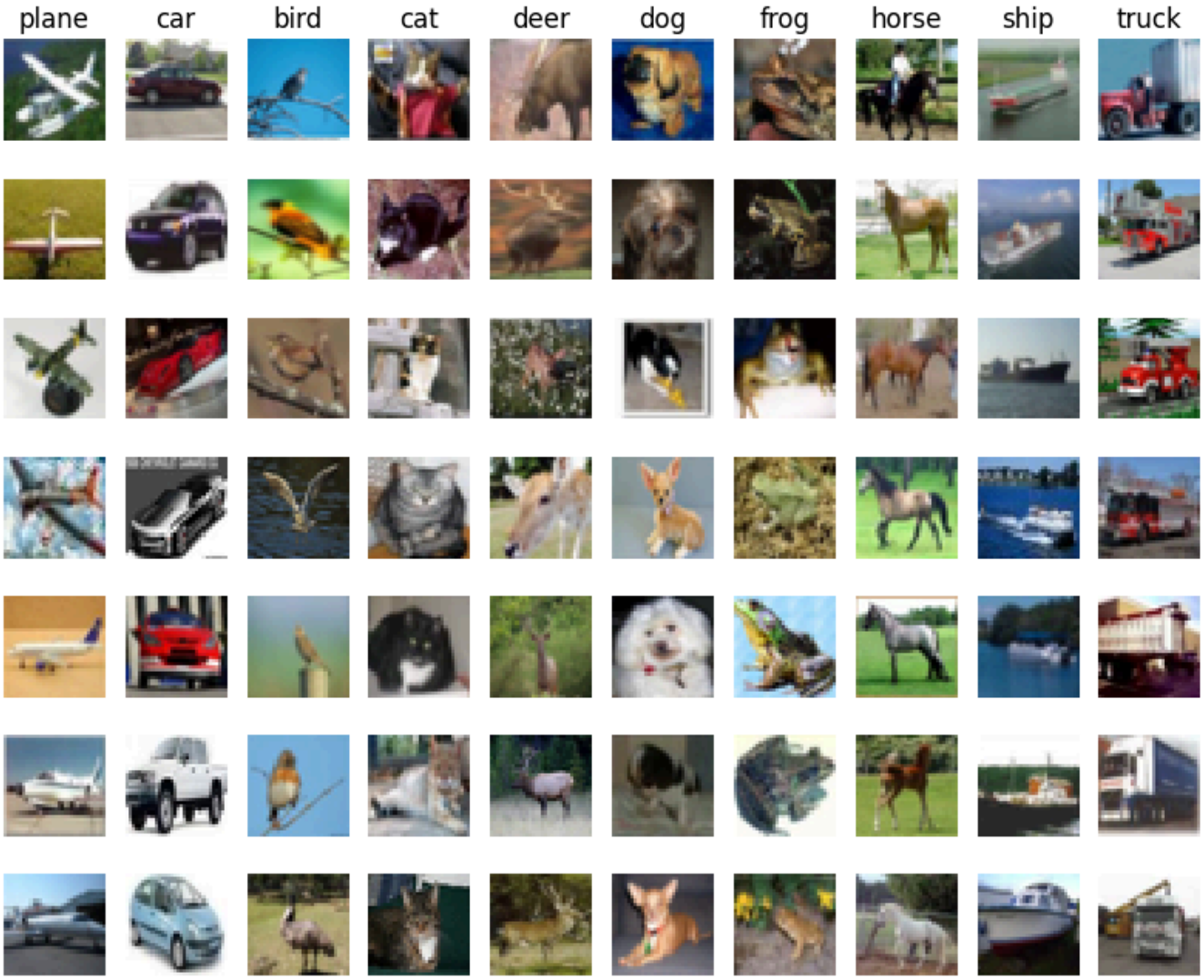
In this exercise you will:

- implement a fully-vectorized **loss function** for the Softmax classifier.
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** 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

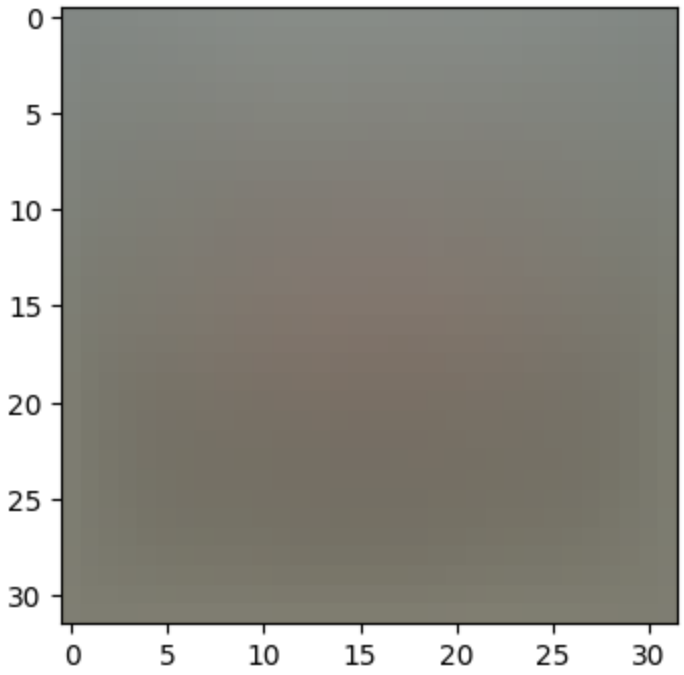
```
[ready] Visualization & autoreload settings complete.
```

CIFAR-10 Data Loading and Preprocessing

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




```
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,)

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

[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 Classifier

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

As you can see, we have prefilled the function `softmax_loss_naive` which uses for loops to evaluate the softmax loss function.

```
loss: 2.388963
loss: 2.388963
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$ *Since there are 10 classes, a random prediction would give each class a probability of 1/10, which is 0.1.*

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

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

```
numerical: -1.700435 analytic: -1.700435, relative error: 3.644047e-09
numerical: -0.352173 analytic: -0.352173, relative error: 1.929128e-07
numerical: 2.418968 analytic: 2.418967, relative error: 3.997566e-08
numerical: -1.057496 analytic: -1.057496, relative error: 1.467064e-08
numerical: 1.863354 analytic: 1.863354, relative error: 2.638738e-08
numerical: 0.537221 analytic: 0.537221, relative error: 1.318414e-08
numerical: 4.258377 analytic: 4.258377, relative error: 1.221471e-11
numerical: -0.677913 analytic: -0.677913, relative error: 4.155236e-08
numerical: 0.412116 analytic: 0.412116, relative error: 2.328918e-07
numerical: -1.805972 analytic: -1.805972, relative error: 9.318435e-09
numerical: -1.394039 analytic: -1.394039, relative error: 3.926214e-08
numerical: -1.899599 analytic: -1.899599, relative error: 1.148263e-08
numerical: -0.196209 analytic: -0.196209, relative error: 2.470200e-07
numerical: -0.199058 analytic: -0.199058, relative error: 4.392419e-08
numerical: -1.208800 analytic: -1.208800, relative error: 1.844511e-08
numerical: -3.001003 analytic: -3.001003, relative error: 3.431242e-08
numerical: -1.380397 analytic: -1.380397, relative error: 7.616553e-09
numerical: 4.914987 analytic: 4.914987, relative error: 1.968208e-08
numerical: -2.546576 analytic: -2.546576, relative error: 5.239118e-10
numerical: 0.445246 analytic: 0.445245, relative error: 1.272662e-07
```

Inline Question 2

Although gradcheck is reliable softmax loss, it is possible that for SVM loss, once in a while, a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a svm loss gradient check could fail? How would change the margin affect of the frequency of this happening?

Note that SVM loss for a sample (x_i, y_i) is defined as:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

where j iterates over all classes except the correct class y_i and s_j denotes the classifier score for j^{th} class. Δ is a scalar margin. For more information, refer to 'Multiclass Support Vector Machine loss' on this page.

Hint: the SVM loss function is not strictly speaking differentiable.

$Your Answer$: *SVM loss have a nondifferentiable points. When* $s_j - s_{y_i} + margin = 0$ *the gradient is not uniquely defined. Because of this, numerical gradient checking can occasionally fail. During numerical differentiation using* $\frac{f(x+h)-f(x-h)}{2h}$ *the two evaluations may fall on different sides of the hinge:*

- $(x + h)$ *: analytic gradient = 1*
- $(x - h)$ *: analytic gradient = 0*

Then the numerical gradient will be $\frac{1-0}{2} = 0.5$*, which differs from the analytic gradient.*

So, a simple 1D example is $L(z) = max(0, z)$ where gradient checking will fail near (z = 0) as written above.

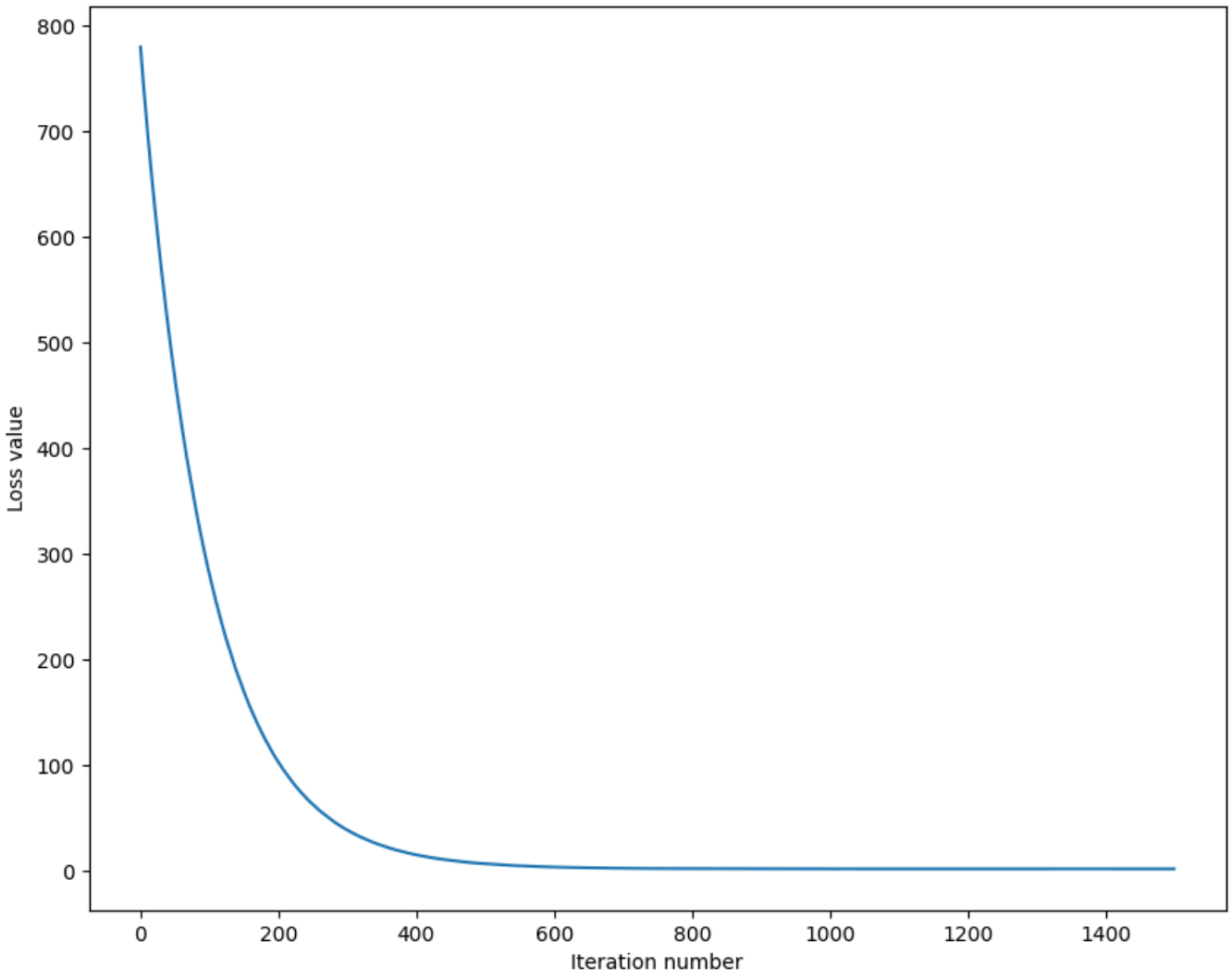
```
Naive loss: 2.388963e+00 computed in 0.042176s
Vectorized loss: 2.388963e+00 computed in 0.001615s
difference: -0.000000
```

```
Naive loss and gradient: computed in 0.027760s
Vectorized loss and gradient: computed in 0.001544s
difference: 1487.845189
```

Stochastic Gradient Descent

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

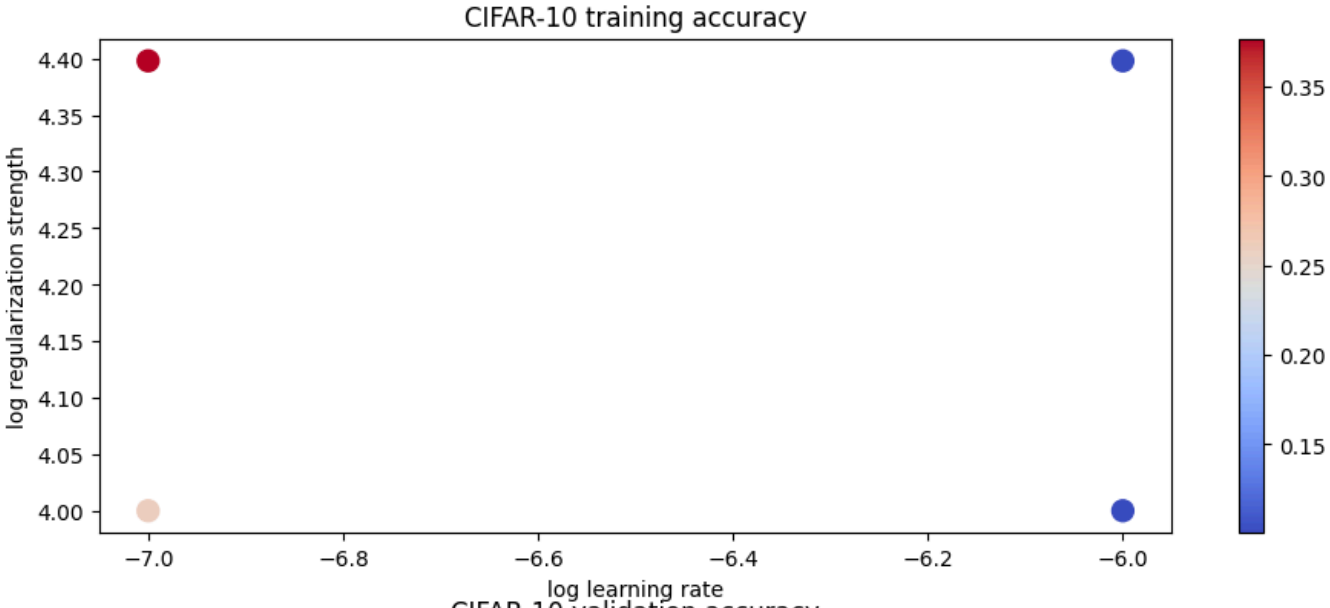
```
iteration 0 / 1500: loss 779.342178
iteration 100 / 1500: loss 280.282180
iteration 200 / 1500: loss 103.210536
iteration 300 / 1500: loss 38.990252
iteration 400 / 1500: loss 15.643946
iteration 500 / 1500: loss 7.179823
iteration 600 / 1500: loss 4.084288
iteration 700 / 1500: loss 2.968384
iteration 800 / 1500: loss 2.581588
iteration 900 / 1500: loss 2.458608
iteration 1000 / 1500: loss 2.398108
iteration 1100 / 1500: loss 2.385488
iteration 1200 / 1500: loss 2.366672
iteration 1300 / 1500: loss 2.382821
iteration 1400 / 1500: loss 2.389853
That took 2.411880s
```

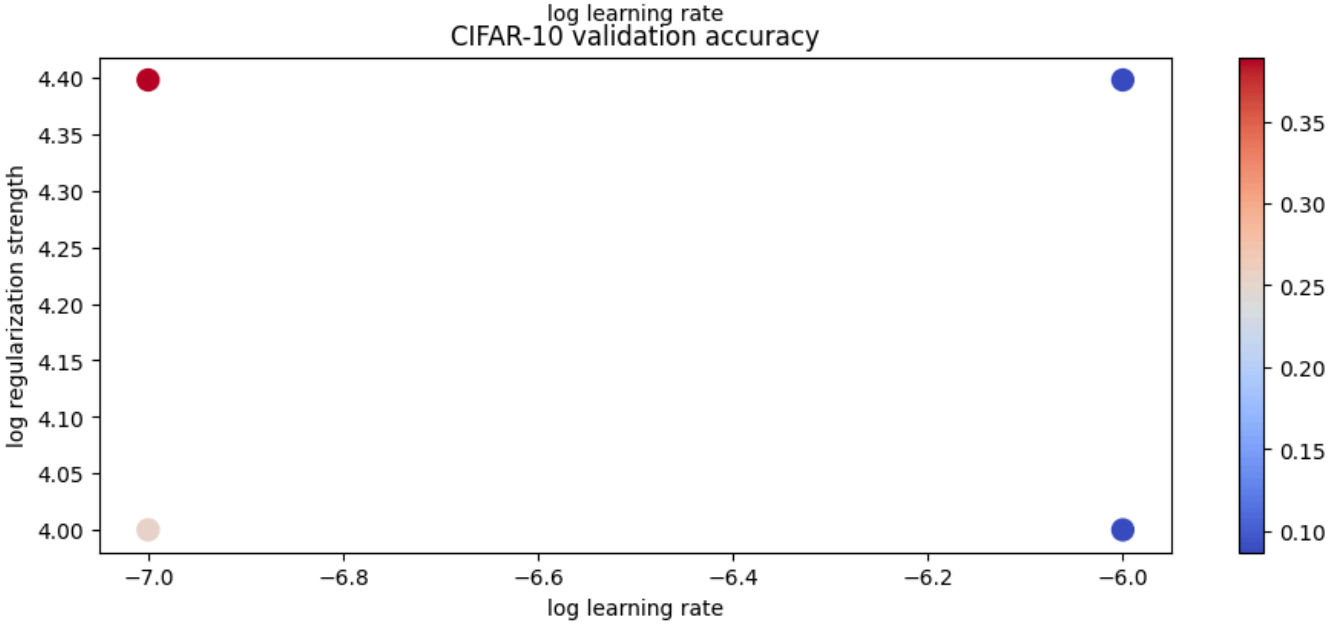



```
training accuracy: 0.380878
validation accuracy: 0.393000

softmax.npy saved.
```

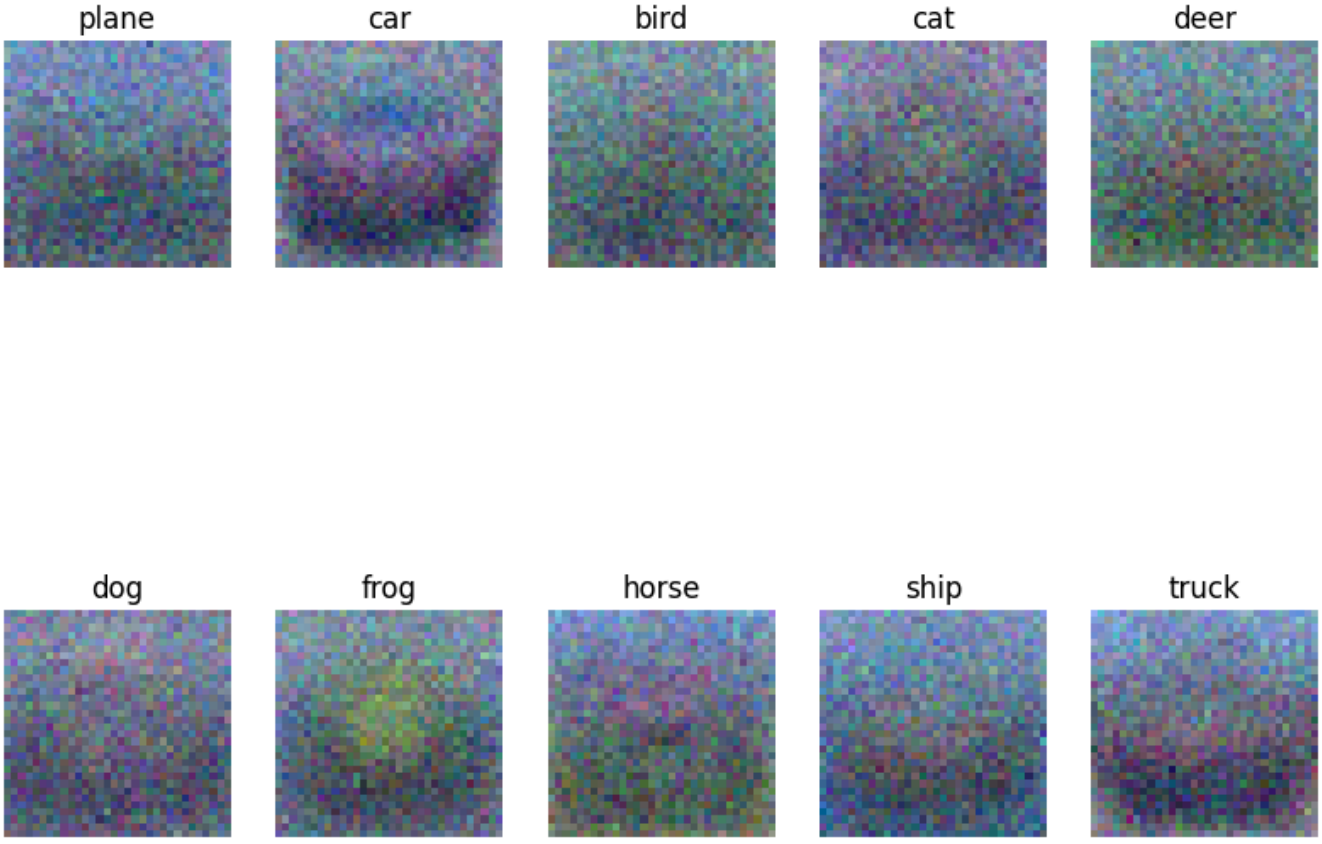
```
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:85: Runtime
Warning: divide by zero encountered in log
  loss -= np.sum(np.log(result))
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:86: Runtime
Warning: overflow encountered in scalar multiply
  loss = loss / num_train + reg * np.sum(W * W)
/home/kim1484/anaconda3/envs/cs231n/lib/python3.8/site-packages/numpy/core/fromnumeri
c.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:86: Runtime
Warning: overflow encountered in multiply
  loss = loss / num_train + reg * np.sum(W * W)
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:79: Runtime
Warning: invalid value encountered in subtract
  scores -= np.reshape(np.max(scores, axis=1), (-1, 1))  # (500, 10)
```

```
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.260592 val accuracy: 0.253000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.376918 val accuracy: 0.389000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.389000
```






```
Softmax classifier on raw pixels final test set accuracy: 0.374000

best_softmax.npy saved.
```



Inline question 3

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

$Your Answer$: *The visualized Softmax weights appear as noisy and blurry color patterns rather than distinct shapes. This is because the softmax classifier is a linear model, it cannot learn complex spatial patterns or object structures. As a result, it only reflects the average color distribution or coarse shape tendency of each class (e.g. the car class)*

Inline Question 4 - *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 change the softmax loss, but leave the SVM loss unchanged.

$Your Answer$: Yes

$Your Explanation$: For the SVM loss, each term has the form $max(0, s_j - s_{y_i} + margin)$. If $(s_j - s_{y_i} + margin)$ is below 0 for all incorrect classes, the hinge loss for that datapoint becomes 0, so adding this datapoint does not change the total SVM loss.

```
[info] ROOT=/home/kim1484/assignment1_colab/assignment1
[info] DATA_DIR=/home/kim1484/assignment1_colab/assignment1/cs231n/datasets
[ready] 환경 준비 완료.
```

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.

```
[ready] Visualization & autoreload settings complete.
```

```
('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,))
```

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:

```
x shape: (2, 4, 5, 6)
w shape: (120, 3)
b shape: (3,)
Testing affine_forward function:
difference:  9.769849468192957e-10
```

Affine layer: backward

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

```
dout shape: (10, 5)
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

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

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

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:

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

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

$Your Answer:$ 1, 2
Sigmoid: About the sigmoide if the input is too large or small the gradient will get zero (or close to zero).
ReLU: About ReLU, if the input is negative the result will be zero. Which means the gradient will be also zero.
Leaky ReLU: Always have a small non-zero gradient(α) for negative inputs, which avoids zero gradient flow during backpropagation

"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:

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

Loss layers: Softmax

Now implement the loss and gradient for softmax in the `softmax_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py`. Other loss functions (e.g. `svm_loss`) can also be implemented in a modular way, however, it is not required for this assignment.

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

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.483503037636722e-09
```

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.

```
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: 3.31e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

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

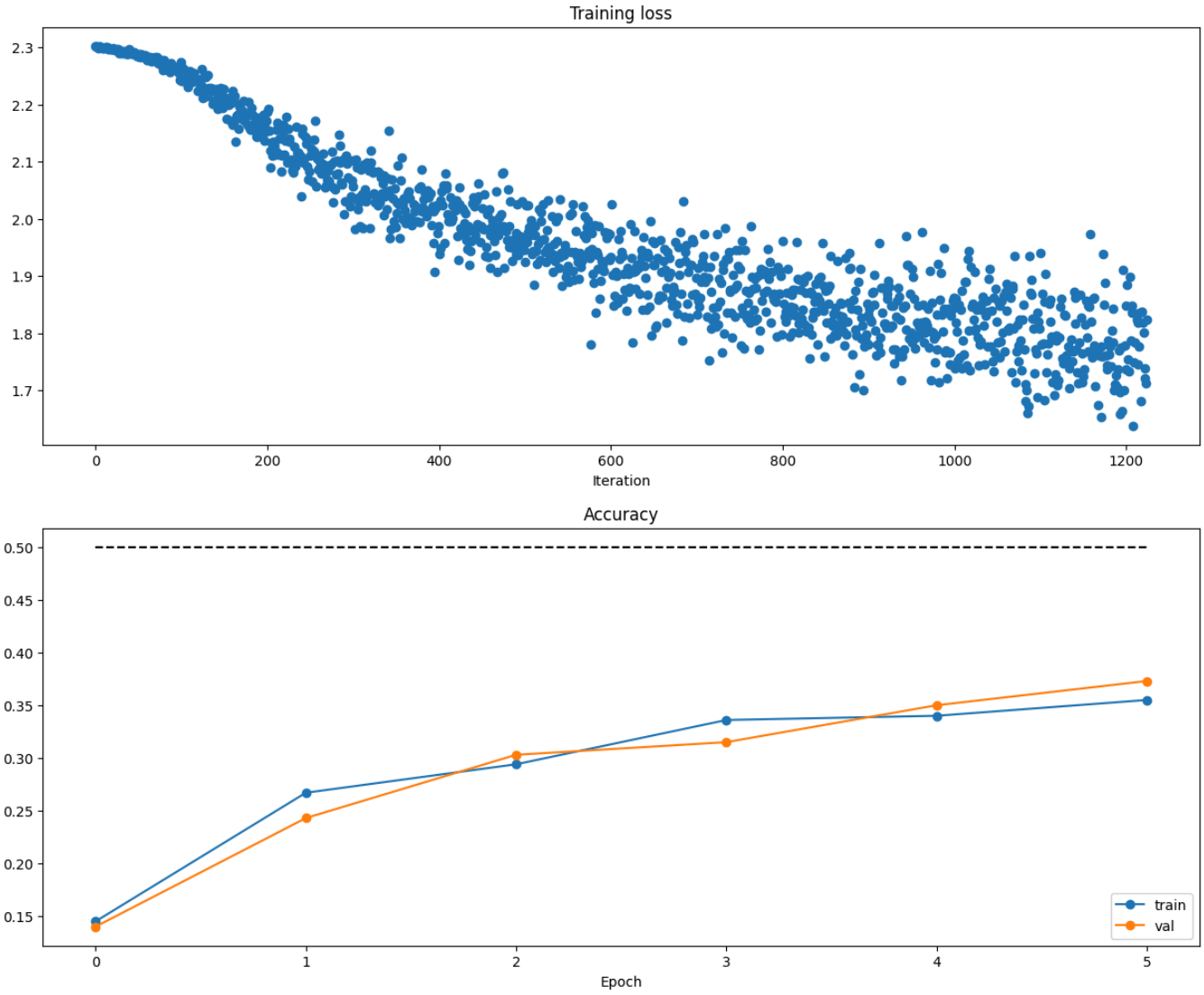
```
(Iteration 1 / 1225) loss: 2.301725
(Epoch 0 / 5) train acc: 0.145000; val_acc: 0.140000
(Iteration 101 / 1225) loss: 2.241923
(Iteration 201 / 1225) loss: 2.187425
(Epoch 1 / 5) train acc: 0.267000; val_acc: 0.243000
(Iteration 301 / 1225) loss: 2.056790
(Iteration 401 / 1225) loss: 1.937978
(Epoch 2 / 5) train acc: 0.294000; val_acc: 0.303000
(Iteration 501 / 1225) loss: 1.924555
(Iteration 601 / 1225) loss: 1.933743
(Iteration 701 / 1225) loss: 1.832777
(Epoch 3 / 5) train acc: 0.336000; val_acc: 0.315000
(Iteration 801 / 1225) loss: 1.960827
(Iteration 901 / 1225) loss: 1.832752
(Epoch 4 / 5) train acc: 0.340000; val_acc: 0.350000
(Iteration 1001 / 1225) loss: 1.739182
(Iteration 1101 / 1225) loss: 1.940517
(Iteration 1201 / 1225) loss: 1.848443
(Epoch 5 / 5) train acc: 0.355000; val_acc: 0.373000
```

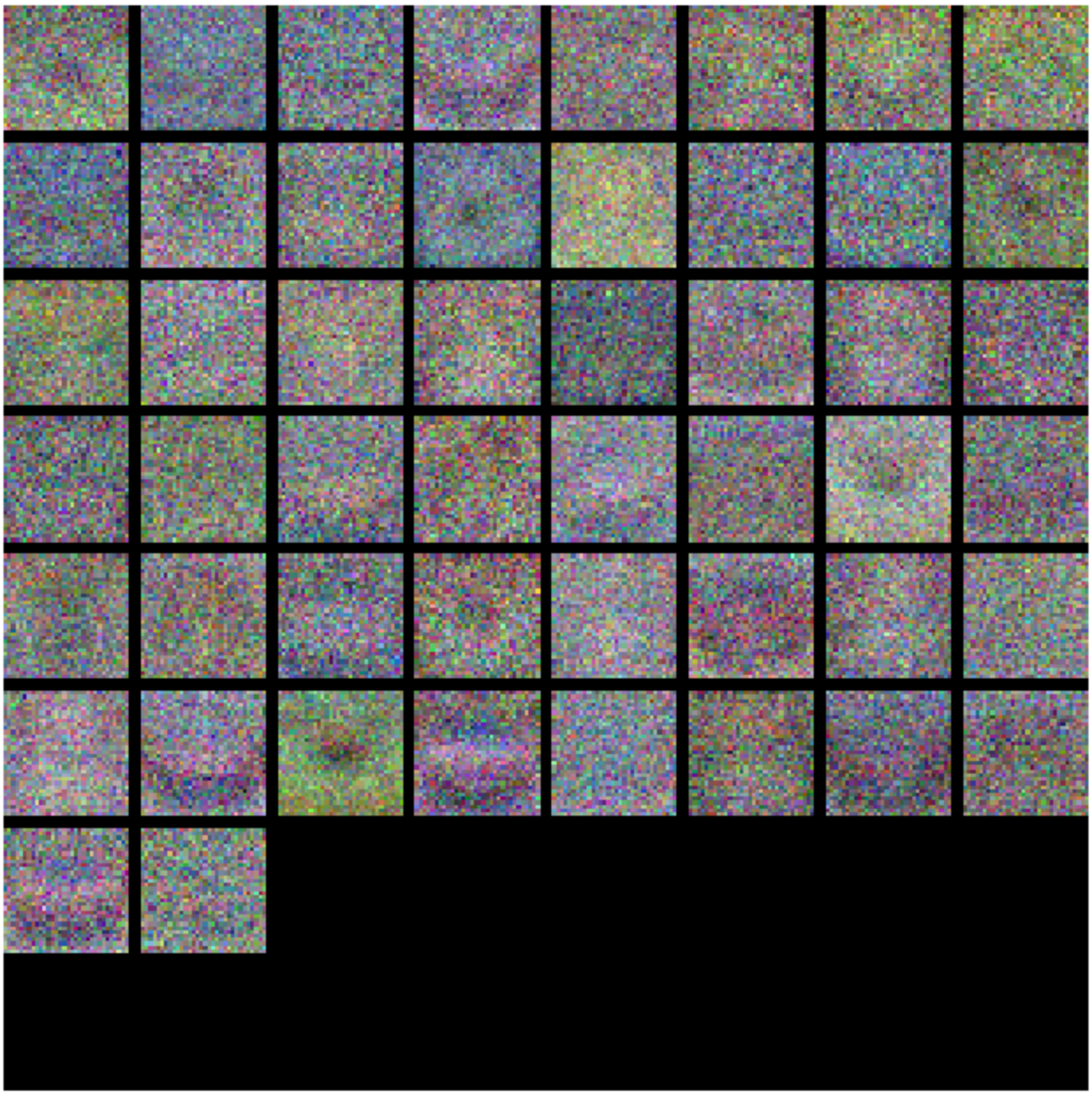
Debug the training

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.





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.).

```
============================================================
hidden size: 50, learning rate: 0.0001, lr decay: 0.75
============================================================
(Iteration 1 / 3675) loss: 2.302208
(Epoch 0 / 15) train acc: 0.089000; val_acc: 0.103000
(Iteration 101 / 3675) loss: 2.269872
(Iteration 201 / 3675) loss: 2.180729
(Epoch 1 / 15) train acc: 0.225000; val_acc: 0.243000
(Iteration 301 / 3675) loss: 2.139334
(Iteration 401 / 3675) loss: 2.055889
(Epoch 2 / 15) train acc: 0.292000; val_acc: 0.291000
(Iteration 501 / 3675) loss: 2.027681
(Iteration 601 / 3675) loss: 2.000881
(Iteration 701 / 3675) loss: 1.865790
(Epoch 3 / 15) train acc: 0.299000; val_acc: 0.311000
(Iteration 801 / 3675) loss: 2.005687
(Iteration 901 / 3675) loss: 1.927090
(Epoch 4 / 15) train acc: 0.327000; val_acc: 0.325000
(Iteration 1001 / 3675) loss: 1.814586
(Iteration 1101 / 3675) loss: 1.919126
(Iteration 1201 / 3675) loss: 1.914328
(Epoch 5 / 15) train acc: 0.315000; val_acc: 0.327000
(Iteration 1301 / 3675) loss: 1.823837
(Iteration 1401 / 3675) loss: 1.949548
(Epoch 6 / 15) train acc: 0.292000; val_acc: 0.335000
(Iteration 1501 / 3675) loss: 1.867285
(Iteration 1601 / 3675) loss: 1.902650
(Iteration 1701 / 3675) loss: 1.849380
(Epoch 7 / 15) train acc: 0.330000; val_acc: 0.345000
(Iteration 1801 / 3675) loss: 1.920131
(Iteration 1901 / 3675) loss: 1.899820
(Epoch 8 / 15) train acc: 0.351000; val_acc: 0.352000
(Iteration 2001 / 3675) loss: 1.814006
(Iteration 2101 / 3675) loss: 1.946156
(Iteration 2201 / 3675) loss: 1.786926
(Epoch 9 / 15) train acc: 0.380000; val_acc: 0.353000
(Iteration 2301 / 3675) loss: 1.788916
(Iteration 2401 / 3675) loss: 1.863340
(Epoch 10 / 15) train acc: 0.353000; val_acc: 0.357000
(Iteration 2501 / 3675) loss: 1.810877
(Iteration 2601 / 3675) loss: 1.833393
(Epoch 11 / 15) train acc: 0.349000; val_acc: 0.358000
(Iteration 2701 / 3675) loss: 1.727624
(Iteration 2801 / 3675) loss: 1.838444
(Iteration 2901 / 3675) loss: 1.770730
(Epoch 12 / 15) train acc: 0.348000; val_acc: 0.360000
(Iteration 3001 / 3675) loss: 1.900193
(Iteration 3101 / 3675) loss: 1.717027
(Epoch 13 / 15) train acc: 0.333000; val_acc: 0.357000
(Iteration 3201 / 3675) loss: 1.829480
(Iteration 3301 / 3675) loss: 1.702334
(Iteration 3401 / 3675) loss: 1.772162
(Epoch 14 / 15) train acc: 0.364000; val_acc: 0.360000
(Iteration 3501 / 3675) loss: 1.861890
(Iteration 3601 / 3675) loss: 1.749665
```

```
(Epoch 15 / 15) train acc: 0.363000; val_acc: 0.359000
================================================================
hidden size: 75, learning rate: 0.0003, lr decay: 0.8
================================================================
(Iteration 1 / 3675) loss: 2.302020
(Epoch 0 / 15) train acc: 0.127000; val_acc: 0.122000
(Iteration 101 / 3675) loss: 2.094417
(Iteration 201 / 3675) loss: 1.877332
(Epoch 1 / 15) train acc: 0.311000; val_acc: 0.354000
(Iteration 301 / 3675) loss: 1.798343
(Iteration 401 / 3675) loss: 1.729738
(Epoch 2 / 15) train acc: 0.387000; val_acc: 0.398000
(Iteration 501 / 3675) loss: 1.778323
(Iteration 601 / 3675) loss: 1.652451
(Iteration 701 / 3675) loss: 1.700581
(Epoch 3 / 15) train acc: 0.401000; val_acc: 0.432000
(Iteration 801 / 3675) loss: 1.727696
(Iteration 901 / 3675) loss: 1.687392
(Epoch 4 / 15) train acc: 0.468000; val_acc: 0.448000
(Iteration 1001 / 3675) loss: 1.582012
(Iteration 1101 / 3675) loss: 1.622367
(Iteration 1201 / 3675) loss: 1.649013
(Epoch 5 / 15) train acc: 0.424000; val_acc: 0.446000
(Iteration 1301 / 3675) loss: 1.665999
(Iteration 1401 / 3675) loss: 1.473240
(Epoch 6 / 15) train acc: 0.434000; val_acc: 0.463000
(Iteration 1501 / 3675) loss: 1.600735
(Iteration 1601 / 3675) loss: 1.621141
(Iteration 1701 / 3675) loss: 1.568323
(Epoch 7 / 15) train acc: 0.433000; val_acc: 0.465000
(Iteration 1801 / 3675) loss: 1.441128
(Iteration 1901 / 3675) loss: 1.675594
(Epoch 8 / 15) train acc: 0.496000; val_acc: 0.458000
(Iteration 2001 / 3675) loss: 1.550722
(Iteration 2101 / 3675) loss: 1.475482
(Iteration 2201 / 3675) loss: 1.533579
(Epoch 9 / 15) train acc: 0.442000; val_acc: 0.466000
(Iteration 2301 / 3675) loss: 1.547075
(Iteration 2401 / 3675) loss: 1.466843
(Epoch 10 / 15) train acc: 0.466000; val_acc: 0.465000
(Iteration 2501 / 3675) loss: 1.417340
(Iteration 2601 / 3675) loss: 1.573513
(Epoch 11 / 15) train acc: 0.449000; val_acc: 0.462000
(Iteration 2701 / 3675) loss: 1.457753
(Iteration 2801 / 3675) loss: 1.491303
(Iteration 2901 / 3675) loss: 1.535973
(Epoch 12 / 15) train acc: 0.448000; val_acc: 0.467000
(Iteration 3001 / 3675) loss: 1.542745
(Iteration 3101 / 3675) loss: 1.580296
(Epoch 13 / 15) train acc: 0.447000; val_acc: 0.467000
(Iteration 3201 / 3675) loss: 1.640223
(Iteration 3301 / 3675) loss: 1.594003
(Iteration 3401 / 3675) loss: 1.440307
(Epoch 14 / 15) train acc: 0.465000; val_acc: 0.469000
(Iteration 3501 / 3675) loss: 1.404369
(Iteration 3601 / 3675) loss: 1.470520
```

```
(Epoch 15 / 15) train acc: 0.481000; val_acc: 0.468000
============================================================
hidden size: 100, learning rate: 0.0005, lr decay: 0.85
============================================================
(Iteration 1 / 3675) loss: 2.305878
(Epoch 0 / 15) train acc: 0.103000; val_acc: 0.125000
(Iteration 101 / 3675) loss: 1.942257
(Iteration 201 / 3675) loss: 1.827047
(Epoch 1 / 15) train acc: 0.374000; val_acc: 0.405000
(Iteration 301 / 3675) loss: 1.703828
(Iteration 401 / 3675) loss: 1.634592
(Epoch 2 / 15) train acc: 0.447000; val_acc: 0.435000
(Iteration 501 / 3675) loss: 1.631321
(Iteration 601 / 3675) loss: 1.662849
(Iteration 701 / 3675) loss: 1.470808
(Epoch 3 / 15) train acc: 0.428000; val_acc: 0.467000
(Iteration 801 / 3675) loss: 1.641560
(Iteration 901 / 3675) loss: 1.480923
(Epoch 4 / 15) train acc: 0.470000; val_acc: 0.470000
(Iteration 1001 / 3675) loss: 1.531020
(Iteration 1101 / 3675) loss: 1.577232
(Iteration 1201 / 3675) loss: 1.436203
(Epoch 5 / 15) train acc: 0.494000; val_acc: 0.474000
(Iteration 1301 / 3675) loss: 1.378047
(Iteration 1401 / 3675) loss: 1.346023
(Epoch 6 / 15) train acc: 0.500000; val_acc: 0.488000
(Iteration 1501 / 3675) loss: 1.441503
(Iteration 1601 / 3675) loss: 1.304212
(Iteration 1701 / 3675) loss: 1.370535
(Epoch 7 / 15) train acc: 0.511000; val_acc: 0.500000
(Iteration 1801 / 3675) loss: 1.401865
(Iteration 1901 / 3675) loss: 1.447340
(Epoch 8 / 15) train acc: 0.514000; val_acc: 0.503000
(Iteration 2001 / 3675) loss: 1.316364
(Iteration 2101 / 3675) loss: 1.420318
(Iteration 2201 / 3675) loss: 1.514687
(Epoch 9 / 15) train acc: 0.535000; val_acc: 0.499000
(Iteration 2301 / 3675) loss: 1.331169
(Iteration 2401 / 3675) loss: 1.238247
(Epoch 10 / 15) train acc: 0.551000; val_acc: 0.500000
(Iteration 2501 / 3675) loss: 1.337534
(Iteration 2601 / 3675) loss: 1.319425
(Epoch 11 / 15) train acc: 0.525000; val_acc: 0.493000
(Iteration 2701 / 3675) loss: 1.365279
(Iteration 2801 / 3675) loss: 1.389755
(Iteration 2901 / 3675) loss: 1.358643
(Epoch 12 / 15) train acc: 0.536000; val_acc: 0.499000
(Iteration 3001 / 3675) loss: 1.464928
(Iteration 3101 / 3675) loss: 1.218806
(Epoch 13 / 15) train acc: 0.552000; val_acc: 0.505000
(Iteration 3201 / 3675) loss: 1.260621
(Iteration 3301 / 3675) loss: 1.240901
(Iteration 3401 / 3675) loss: 1.371293
(Epoch 14 / 15) train acc: 0.547000; val_acc: 0.504000
(Iteration 3501 / 3675) loss: 1.256126
(Iteration 3601 / 3675) loss: 1.435832
```

```
(Epoch 15 / 15) train acc: 0.553000; val_acc: 0.509000
================================================================
hidden size: 125, learning rate: 0.0007, lr decay: 0.9
================================================================
(Iteration 1 / 3675) loss: 2.298223
(Epoch 0 / 15) train acc: 0.130000; val_acc: 0.135000
(Iteration 101 / 3675) loss: 1.786951
(Iteration 201 / 3675) loss: 1.693566
(Epoch 1 / 15) train acc: 0.413000; val_acc: 0.407000
(Iteration 301 / 3675) loss: 1.629973
(Iteration 401 / 3675) loss: 1.670703
(Epoch 2 / 15) train acc: 0.461000; val_acc: 0.456000
(Iteration 501 / 3675) loss: 1.619464
(Iteration 601 / 3675) loss: 1.499812
(Iteration 701 / 3675) loss: 1.530245
(Epoch 3 / 15) train acc: 0.467000; val_acc: 0.460000
(Iteration 801 / 3675) loss: 1.490324
(Iteration 901 / 3675) loss: 1.451270
(Epoch 4 / 15) train acc: 0.530000; val_acc: 0.479000
(Iteration 1001 / 3675) loss: 1.372859
(Iteration 1101 / 3675) loss: 1.370228
(Iteration 1201 / 3675) loss: 1.320328
(Epoch 5 / 15) train acc: 0.530000; val_acc: 0.489000
(Iteration 1301 / 3675) loss: 1.174534
(Iteration 1401 / 3675) loss: 1.378404
(Epoch 6 / 15) train acc: 0.518000; val_acc: 0.492000
(Iteration 1501 / 3675) loss: 1.332694
(Iteration 1601 / 3675) loss: 1.219642
(Iteration 1701 / 3675) loss: 1.335411
(Epoch 7 / 15) train acc: 0.557000; val_acc: 0.504000
(Iteration 1801 / 3675) loss: 1.289719
(Iteration 1901 / 3675) loss: 1.406917
(Epoch 8 / 15) train acc: 0.559000; val_acc: 0.502000
(Iteration 2001 / 3675) loss: 1.261266
(Iteration 2101 / 3675) loss: 1.184197
(Iteration 2201 / 3675) loss: 1.267301
(Epoch 9 / 15) train acc: 0.555000; val_acc: 0.515000
(Iteration 2301 / 3675) loss: 1.149572
(Iteration 2401 / 3675) loss: 1.189977
(Epoch 10 / 15) train acc: 0.591000; val_acc: 0.524000
(Iteration 2501 / 3675) loss: 1.317291
(Iteration 2601 / 3675) loss: 1.154614
(Epoch 11 / 15) train acc: 0.576000; val_acc: 0.515000
(Iteration 2701 / 3675) loss: 1.095757
(Iteration 2801 / 3675) loss: 1.239589
(Iteration 2901 / 3675) loss: 1.293036
(Epoch 12 / 15) train acc: 0.574000; val_acc: 0.512000
(Iteration 3001 / 3675) loss: 1.127370
(Iteration 3101 / 3675) loss: 1.251010
(Epoch 13 / 15) train acc: 0.597000; val_acc: 0.517000
(Iteration 3201 / 3675) loss: 1.272345
(Iteration 3301 / 3675) loss: 1.163338
(Iteration 3401 / 3675) loss: 1.140771
(Epoch 14 / 15) train acc: 0.628000; val_acc: 0.518000
(Iteration 3501 / 3675) loss: 1.175282
(Iteration 3601 / 3675) loss: 1.092050
```

```
(Epoch 15 / 15) train acc: 0.601000; val_acc: 0.539000
================================================================
hidden size: 150, learning rate: 0.001, lr decay: 0.95
================================================================
(Iteration 1 / 3675) loss: 2.305420
(Epoch 0 / 15) train acc: 0.153000; val_acc: 0.151000
(Iteration 101 / 3675) loss: 1.717234
(Iteration 201 / 3675) loss: 1.659075
(Epoch 1 / 15) train acc: 0.423000; val_acc: 0.430000
(Iteration 301 / 3675) loss: 1.628145
(Iteration 401 / 3675) loss: 1.510267
(Epoch 2 / 15) train acc: 0.496000; val_acc: 0.459000
(Iteration 501 / 3675) loss: 1.517149
(Iteration 601 / 3675) loss: 1.469220
(Iteration 701 / 3675) loss: 1.466176
(Epoch 3 / 15) train acc: 0.521000; val_acc: 0.506000
(Iteration 801 / 3675) loss: 1.466712
(Iteration 901 / 3675) loss: 1.270487
(Epoch 4 / 15) train acc: 0.561000; val_acc: 0.518000
(Iteration 1001 / 3675) loss: 1.418880
(Iteration 1101 / 3675) loss: 1.349000
(Iteration 1201 / 3675) loss: 1.280850
(Epoch 5 / 15) train acc: 0.555000; val_acc: 0.503000
(Iteration 1301 / 3675) loss: 1.202312
(Iteration 1401 / 3675) loss: 1.247353
(Epoch 6 / 15) train acc: 0.551000; val_acc: 0.528000
(Iteration 1501 / 3675) loss: 1.350306
(Iteration 1601 / 3675) loss: 1.271361
(Iteration 1701 / 3675) loss: 1.185882
(Epoch 7 / 15) train acc: 0.587000; val_acc: 0.513000
(Iteration 1801 / 3675) loss: 1.100769
(Iteration 1901 / 3675) loss: 1.123216
(Epoch 8 / 15) train acc: 0.598000; val_acc: 0.529000
(Iteration 2001 / 3675) loss: 1.128339
(Iteration 2101 / 3675) loss: 1.265283
(Iteration 2201 / 3675) loss: 1.356814
(Epoch 9 / 15) train acc: 0.594000; val_acc: 0.514000
(Iteration 2301 / 3675) loss: 1.148923
(Iteration 2401 / 3675) loss: 1.146622
(Epoch 10 / 15) train acc: 0.596000; val_acc: 0.525000
(Iteration 2501 / 3675) loss: 1.160169
(Iteration 2601 / 3675) loss: 1.083604
(Epoch 11 / 15) train acc: 0.605000; val_acc: 0.539000
(Iteration 2701 / 3675) loss: 1.133964
(Iteration 2801 / 3675) loss: 1.115286
(Iteration 2901 / 3675) loss: 1.033042
(Epoch 12 / 15) train acc: 0.599000; val_acc: 0.541000
(Iteration 3001 / 3675) loss: 1.153512
(Iteration 3101 / 3675) loss: 0.956935
(Epoch 13 / 15) train acc: 0.626000; val_acc: 0.543000
(Iteration 3201 / 3675) loss: 1.093087
(Iteration 3301 / 3675) loss: 1.029433
(Iteration 3401 / 3675) loss: 1.144068
(Epoch 14 / 15) train acc: 0.661000; val_acc: 0.533000
(Iteration 3501 / 3675) loss: 0.991575
(Iteration 3601 / 3675) loss: 1.059259
```

```
(Epoch 15 / 15) train acc: 0.620000; val_acc: 0.541000
best validation accuracy achieved during cross-validation: 0.543000
```

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.

```
Validation set accuracy:  0.551
```

```
Test set accuracy:  0.546
```

```
best_two_layer_net.npy saved.
```

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. Training on a larger and more diverse dataset allows the model to learn a wider variety of patterns, improving generalization and thus reducing the gap between training and test accuracy.
2. Increasing the number of hidden units expands the model's capacity, allowing it to fit the training data more accurately. This can increase the gap.
3. A stronger regularization term prevents the model from fitting the training data too closely, thereby reducing overfitting and improving performance on the test set.

```
[info] ROOT=/home/kim1484/assignment1_colab/assignment1
[info] DATA_DIR=/home/kim1484/assignment1_colab/assignment1/cs231n/datasets
[ready] 환경 준비 완료.
```

Image features exercise

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

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

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

Load data

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

Extract Features

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

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

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

```
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
```

Train Softmax classifier on features

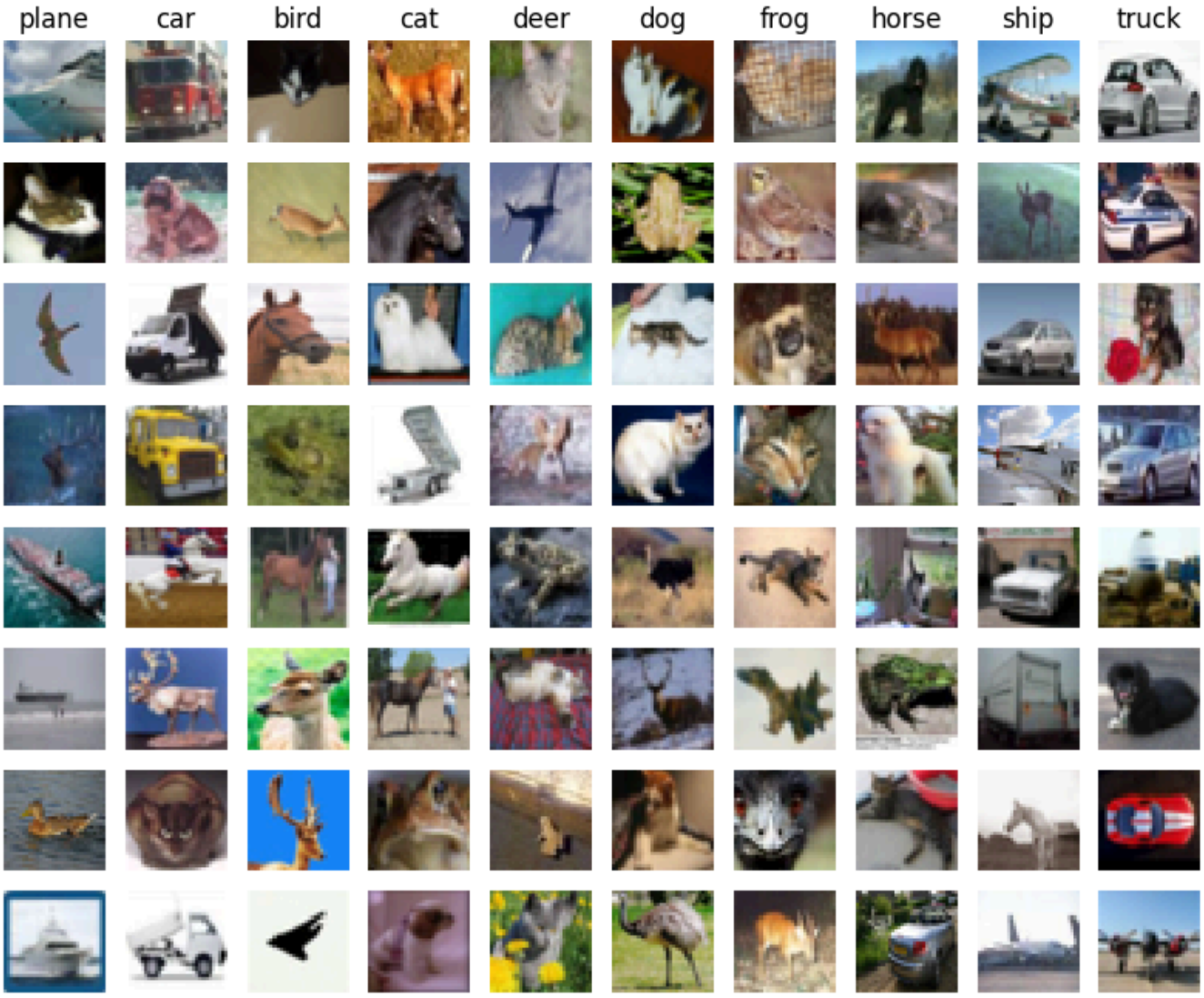
Using the Softmax code developed earlier in the assignment, train Softmax classifiers on top of the features extracted above; this should achieve better results than training them

directly on top of raw pixels.

```
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:85: Runtime
Warning: divide by zero encountered in log
  loss -= np.sum(np.log(result))
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:86: Runtime
Warning: overflow encountered in scalar multiply
  loss = loss / num_train + reg * np.sum(W * W)
/home/kim1484/anaconda3/envs/cs231n/lib/python3.8/site-packages/numpy/core/fromnumeri
c.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:86: Runtime
Warning: overflow encountered in multiply
  loss = loss / num_train + reg * np.sum(W * W)
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.420612 val accuracy: 0.432000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.348000 val accuracy: 0.364000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.323306 val accuracy: 0.335000
lr 1.000000e-06 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved: 0.432000
/home/kim1484/assignment1_colab/assignment1/cs231n/classifiers/softmax.py:102: Runtim
eWarning: overflow encountered in multiply
  dW = dW / num_train + reg * 2 * W
```

```
0.438
```

```
best_softmax_features.npy saved.
```



Inline question 1:

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

$\mathit{Your Answer}$: Most of the misclassifications do not appear to make much sense, as there is no consistent pattern across most classes. However, for the plane class, the model tends to predict images with sky or sea backgrounds as plane. In contrast, the other classes show no consistent relationship with background features, indicating that the model's predictions are largely inconsistent and not based on stable visual cues.

Neural Network on image features

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

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

```
(49000, 170)
(49000, 169)

learning rate: 0.001 regularization strength: 0.0 val accuracy: 0.612
learning rate: 0.001 regularization strength: 0.0001 val accuracy: 0.608
learning rate: 0.001 regularization strength: 0.0003 val accuracy: 0.613
learning rate: 0.001 regularization strength: 0.001 val accuracy: 0.62
learning rate: 0.0003 regularization strength: 0.0 val accuracy: 0.553
learning rate: 0.0003 regularization strength: 0.0001 val accuracy: 0.556
learning rate: 0.0003 regularization strength: 0.0003 val accuracy: 0.553
learning rate: 0.0003 regularization strength: 0.001 val accuracy: 0.559
learning rate: 0.1 regularization strength: 0.0 val accuracy: 0.124
learning rate: 0.1 regularization strength: 0.0001 val accuracy: 0.112
learning rate: 0.1 regularization strength: 0.0003 val accuracy: 0.112
learning rate: 0.1 regularization strength: 0.001 val accuracy: 0.124
learning rate: 0.0005 regularization strength: 0.0 val accuracy: 0.591
learning rate: 0.0005 regularization strength: 0.0001 val accuracy: 0.578
learning rate: 0.0005 regularization strength: 0.0003 val accuracy: 0.59
learning rate: 0.0005 regularization strength: 0.001 val accuracy: 0.579

0.589

best_two_layer_net_features.npy saved.
```

```
[info] ROOT=/home/kim1484/assignment1_colab/assignment1
[info] DATA_DIR=/home/kim1484/assignment1_colab/assignment1/cs231n/datasets
[ready] 환경 준비 완료.
```

Multi-Layer Fully Connected Network

In this exercise, you will implement a fully connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the network initialization, forward pass, and backward pass. Throughout this assignment, you will be implementing layers in `cs231n/layers.py`. You can re-use your implementations for `affine_forward`, `affine_backward`, `relu_forward`, `relu_backward`, and `softmax_loss` from before. For right now, don't worry about implementing dropout or batch/layer normalization yet, as you will add those features later.

```
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,)
```

Initial Loss and Gradient Check

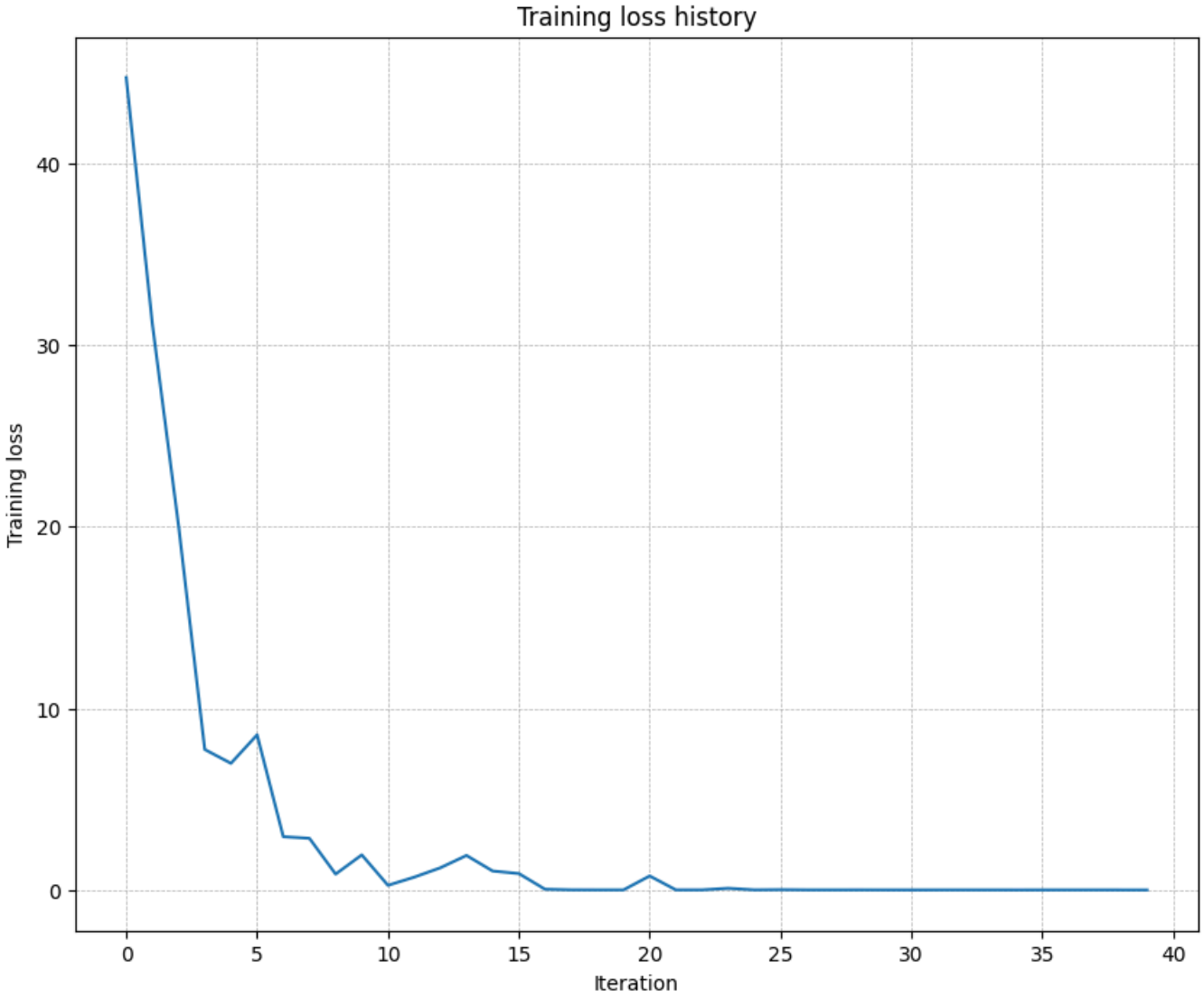
As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. This is a good way to see if the initial losses seem reasonable.

For gradient checking, you should expect to see errors around 1e-7 or less.

```
Running check with reg =  0
Initial loss:  2.300479089768492
W1 relative error: 1.0252674471656573e-07
W2 relative error: 2.2120479295080622e-05
W3 relative error: 4.5623278736665505e-07
b1 relative error: 4.6600944653202505e-09
b2 relative error: 2.085654276112763e-09
b3 relative error: 1.689724888469736e-10
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 3.904541941902138e-09
W2 relative error: 6.86942277940646e-08
W3 relative error: 3.483989247437803e-08
b1 relative error: 1.4752427965311745e-08
b2 relative error: 1.4615869332918208e-09
b3 relative error: 1.3200479211447775e-10
```

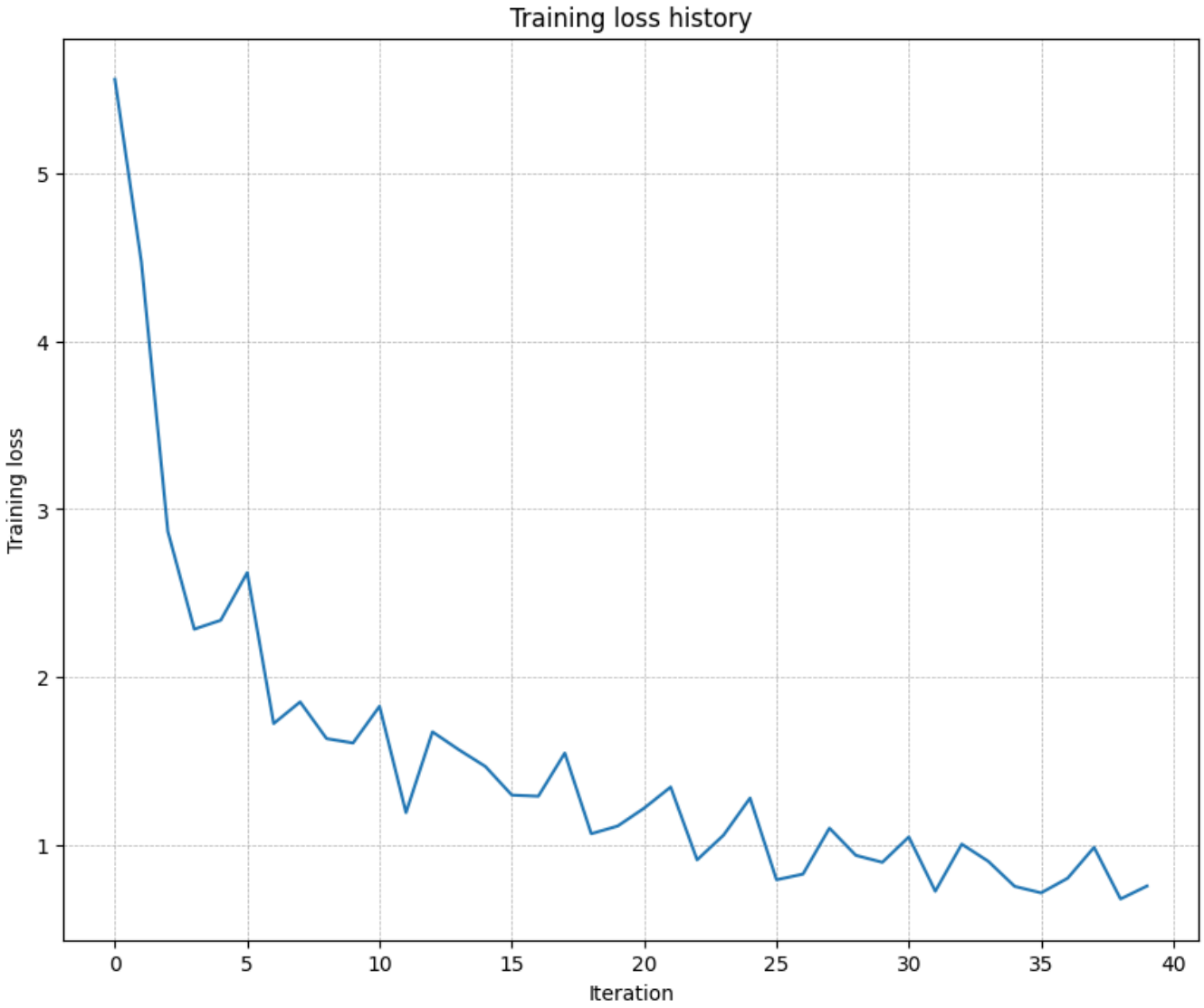
As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy within 20 epochs.

```
(Iteration 1 / 40) loss: 44.720838
(Epoch 0 / 20) train acc: 0.120000; val_acc: 0.122000
(Epoch 1 / 20) train acc: 0.220000; val_acc: 0.137000
(Epoch 2 / 20) train acc: 0.460000; val_acc: 0.141000
(Epoch 3 / 20) train acc: 0.660000; val_acc: 0.155000
(Epoch 4 / 20) train acc: 0.720000; val_acc: 0.167000
(Epoch 5 / 20) train acc: 0.820000; val_acc: 0.176000
(Iteration 11 / 40) loss: 0.254061
(Epoch 6 / 20) train acc: 0.860000; val_acc: 0.166000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.157000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.164000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.162000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.165000
(Iteration 21 / 40) loss: 0.769242
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.160000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.160000
(Iteration 31 / 40) loss: 0.001292
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.161000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.161000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.161000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.161000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.161000
```



Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

```
(Iteration 1 / 40) loss: 5.557032
(Epoch 0 / 20) train acc: 0.120000; val_acc: 0.128000
(Epoch 1 / 20) train acc: 0.220000; val_acc: 0.115000
(Epoch 2 / 20) train acc: 0.320000; val_acc: 0.114000
(Epoch 3 / 20) train acc: 0.360000; val_acc: 0.118000
(Epoch 4 / 20) train acc: 0.340000; val_acc: 0.123000
(Epoch 5 / 20) train acc: 0.460000; val_acc: 0.112000
(Iteration 11 / 40) loss: 1.827969
(Epoch 6 / 20) train acc: 0.500000; val_acc: 0.126000
(Epoch 7 / 20) train acc: 0.600000; val_acc: 0.127000
(Epoch 8 / 20) train acc: 0.660000; val_acc: 0.123000
(Epoch 9 / 20) train acc: 0.660000; val_acc: 0.127000
(Epoch 10 / 20) train acc: 0.660000; val_acc: 0.132000
(Iteration 21 / 40) loss: 1.222287
(Epoch 11 / 20) train acc: 0.700000; val_acc: 0.128000
(Epoch 12 / 20) train acc: 0.760000; val_acc: 0.141000
(Epoch 13 / 20) train acc: 0.800000; val_acc: 0.135000
(Epoch 14 / 20) train acc: 0.780000; val_acc: 0.134000
(Epoch 15 / 20) train acc: 0.780000; val_acc: 0.140000
(Iteration 31 / 40) loss: 1.050772
(Epoch 16 / 20) train acc: 0.800000; val_acc: 0.138000
(Epoch 17 / 20) train acc: 0.840000; val_acc: 0.136000
(Epoch 18 / 20) train acc: 0.840000; val_acc: 0.146000
(Epoch 19 / 20) train acc: 0.840000; val_acc: 0.141000
(Epoch 20 / 20) train acc: 0.880000; val_acc: 0.143000
```



Inline Question 1:

Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Answer:

[Training the five-layer network seemed to be more sensitive to the initalization scale. Using the same initialization scale and learning rate, the five-layer network demonstrated significantly larger fluctuations and slower stabilization compared to the three-layer network. This indicates that the deeper model is more sensitive to weight initialization.

This is because, the variance of activations and gradients is multipled across layers, making both exploding and vanishing effect.]

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

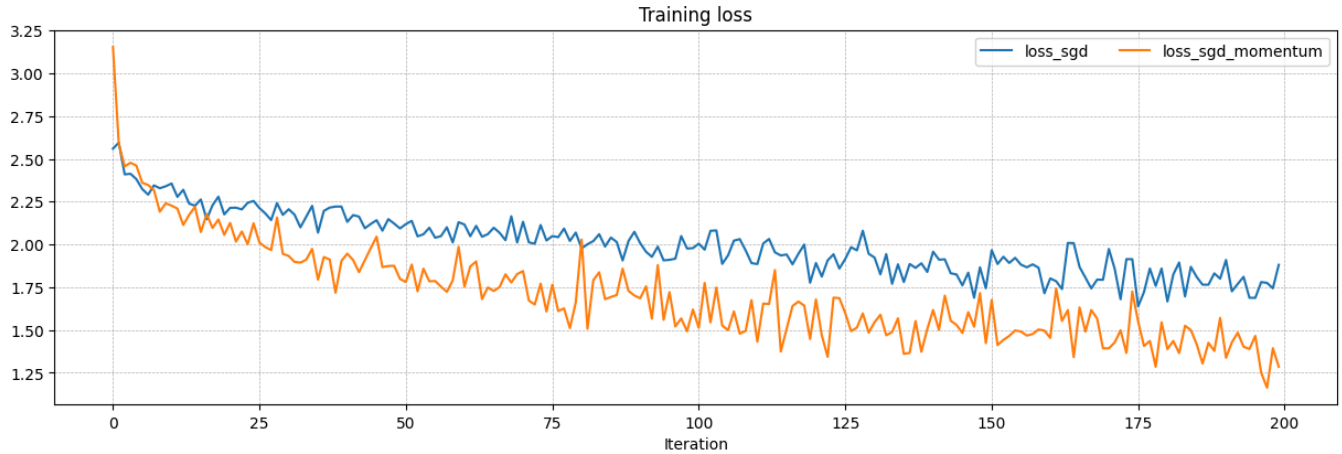
Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

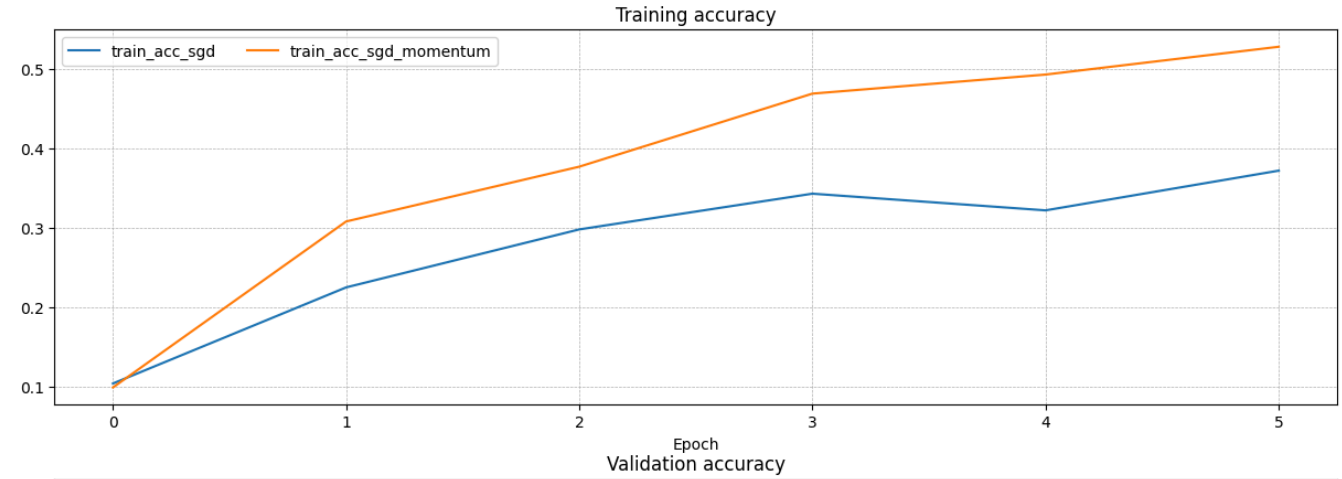
```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

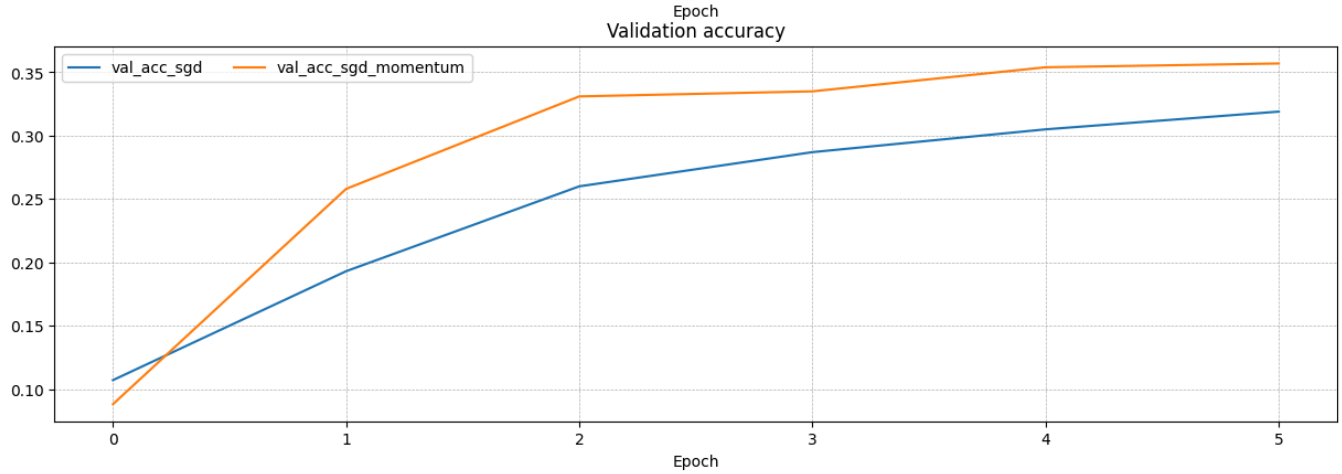
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

```
Running with  sgd
(Iteration 1 / 200) loss: 2.559978
```

```
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.107000
(Iteration 11 / 200) loss: 2.356070
(Iteration 21 / 200) loss: 2.214091
(Iteration 31 / 200) loss: 2.205928
(Epoch 1 / 5) train acc: 0.225000; val_acc: 0.193000
(Iteration 41 / 200) loss: 2.132095
(Iteration 51 / 200) loss: 2.118950
(Iteration 61 / 200) loss: 2.116443
(Iteration 71 / 200) loss: 2.132549
(Epoch 2 / 5) train acc: 0.298000; val_acc: 0.260000
(Iteration 81 / 200) loss: 1.977227
(Iteration 91 / 200) loss: 2.007528
(Iteration 101 / 200) loss: 2.004762
(Iteration 111 / 200) loss: 1.885342
(Epoch 3 / 5) train acc: 0.343000; val_acc: 0.287000
(Iteration 121 / 200) loss: 1.891517
(Iteration 131 / 200) loss: 1.923677
(Iteration 141 / 200) loss: 1.957744
(Iteration 151 / 200) loss: 1.966736
(Epoch 4 / 5) train acc: 0.322000; val_acc: 0.305000
(Iteration 161 / 200) loss: 1.801483
(Iteration 171 / 200) loss: 1.973779
(Iteration 181 / 200) loss: 1.666573
(Iteration 191 / 200) loss: 1.909494
(Epoch 5 / 5) train acc: 0.372000; val_acc: 0.319000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153777
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.088000
(Iteration 11 / 200) loss: 2.227203
(Iteration 21 / 200) loss: 2.125706
(Iteration 31 / 200) loss: 1.932679
(Epoch 1 / 5) train acc: 0.308000; val_acc: 0.258000
(Iteration 41 / 200) loss: 1.946329
(Iteration 51 / 200) loss: 1.780464
(Iteration 61 / 200) loss: 1.753502
(Iteration 71 / 200) loss: 1.844626
(Epoch 2 / 5) train acc: 0.377000; val_acc: 0.331000
(Iteration 81 / 200) loss: 2.028389
(Iteration 91 / 200) loss: 1.685415
(Iteration 101 / 200) loss: 1.513205
(Iteration 111 / 200) loss: 1.431671
(Epoch 3 / 5) train acc: 0.469000; val_acc: 0.335000
(Iteration 121 / 200) loss: 1.678510
(Iteration 131 / 200) loss: 1.545244
(Iteration 141 / 200) loss: 1.616973
(Iteration 151 / 200) loss: 1.675532
(Epoch 4 / 5) train acc: 0.493000; val_acc: 0.354000
(Iteration 161 / 200) loss: 1.454181
(Iteration 171 / 200) loss: 1.392746
(Iteration 181 / 200) loss: 1.388412
(Iteration 191 / 200) loss: 1.337808
(Epoch 5 / 5) train acc: 0.528000; val_acc: 0.357000
```







RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

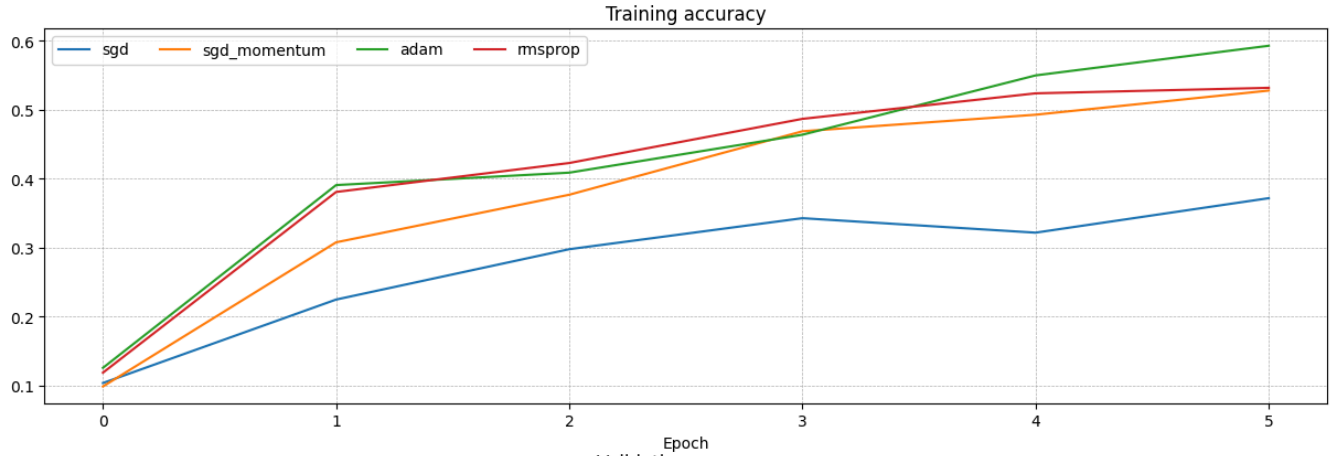
```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

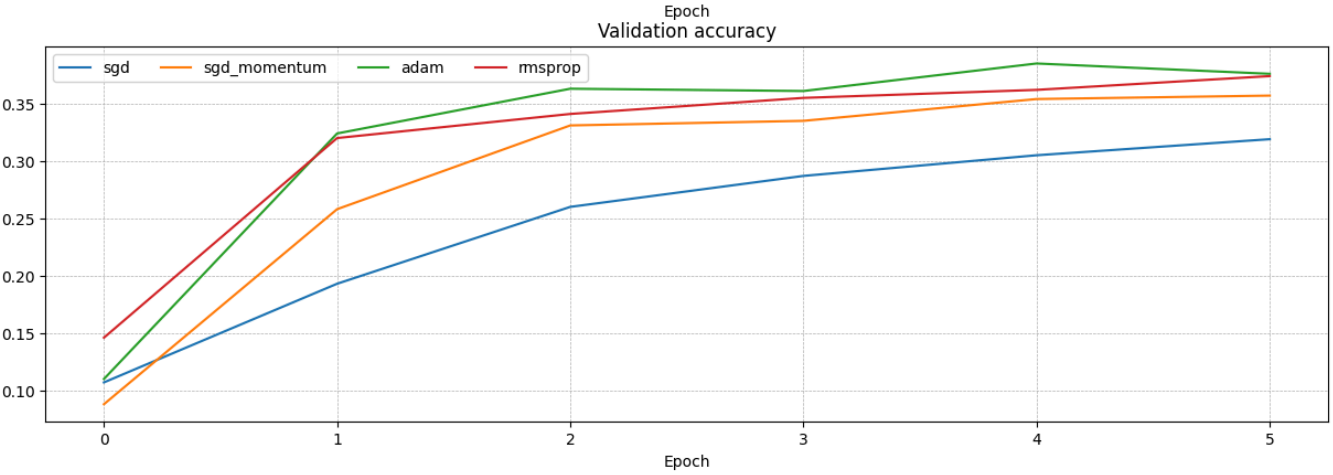
```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

```
Running with  adam
(Iteration 1 / 200) loss: 3.476928
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.014790
(Iteration 21 / 200) loss: 2.160730
(Iteration 31 / 200) loss: 1.877403
(Epoch 1 / 5) train acc: 0.391000; val_acc: 0.324000
(Iteration 41 / 200) loss: 1.770985
(Iteration 51 / 200) loss: 1.704804
(Iteration 61 / 200) loss: 1.927258
(Iteration 71 / 200) loss: 1.553151
(Epoch 2 / 5) train acc: 0.409000; val_acc: 0.363000
(Iteration 81 / 200) loss: 1.624887
(Iteration 91 / 200) loss: 1.370983
(Iteration 101 / 200) loss: 1.450614
(Iteration 111 / 200) loss: 1.478062
(Epoch 3 / 5) train acc: 0.464000; val_acc: 0.361000
(Iteration 121 / 200) loss: 1.253761
(Iteration 131 / 200) loss: 1.435260
(Iteration 141 / 200) loss: 1.429019
(Iteration 151 / 200) loss: 1.347495
(Epoch 4 / 5) train acc: 0.550000; val_acc: 0.385000
(Iteration 161 / 200) loss: 1.280793
(Iteration 171 / 200) loss: 1.248500
(Iteration 181 / 200) loss: 1.023637
(Iteration 191 / 200) loss: 1.176865
(Epoch 5 / 5) train acc: 0.593000; val_acc: 0.376000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895731
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.486923
(Iteration 71 / 200) loss: 1.628511
(Epoch 2 / 5) train acc: 0.423000; val_acc: 0.341000
(Iteration 81 / 200) loss: 1.506182
(Iteration 91 / 200) loss: 1.600674
(Iteration 101 / 200) loss: 1.478501
(Iteration 111 / 200) loss: 1.577709
(Epoch 3 / 5) train acc: 0.487000; val_acc: 0.355000
(Iteration 121 / 200) loss: 1.495931
(Iteration 131 / 200) loss: 1.525799
(Iteration 141 / 200) loss: 1.552580
(Iteration 151 / 200) loss: 1.654283
(Epoch 4 / 5) train acc: 0.524000; val_acc: 0.362000
(Iteration 161 / 200) loss: 1.589371
(Iteration 171 / 200) loss: 1.413529
(Iteration 181 / 200) loss: 1.500273
(Iteration 191 / 200) loss: 1.365942
```

```
(Epoch 5 / 5) train acc: 0.532000; val_acc: 0.374000
```





Inline Question 2:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad

update rule, why do you think the updates would become very small? Would Adam have the same issue?

Answer:

[In AdaGrad, the cache grows continuously as training progresses. Because the denominator in the update rule increases over time, the effective learning rate for each parameter steadily decreases and can eventually become extremely small. As a result, the parameter updates nearly vanish, causing the network to learn very slowly.

In contrast, Adam uses an exponential moving average rather than a cumulative sum. This prevents the denominator from growing without bound, so Adam does not suffer from the same issue as AdaGrad.]

Train a Good Model!

Train the best fully connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully connected network.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the next assignment, we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional networks rather than fully connected networks.

Note: In the next assignment, you will learn techniques like BatchNormalization and Dropout which can help you train powerful models.

```
(Iteration 1 / 9800) loss: 2.702803
(Epoch 0 / 20) train acc: 0.124000; val_acc: 0.111000
(Iteration 11 / 9800) loss: 2.178173
(Iteration 21 / 9800) loss: 1.951725
(Iteration 31 / 9800) loss: 1.905712
(Iteration 41 / 9800) loss: 1.913033
(Iteration 51 / 9800) loss: 1.946167
(Iteration 61 / 9800) loss: 1.726409
(Iteration 71 / 9800) loss: 1.807045
(Iteration 81 / 9800) loss: 2.013886
(Iteration 91 / 9800) loss: 1.941308
(Iteration 101 / 9800) loss: 1.781140
(Iteration 111 / 9800) loss: 1.883894
(Iteration 121 / 9800) loss: 1.591975
(Iteration 131 / 9800) loss: 1.878382
(Iteration 141 / 9800) loss: 1.665713
(Iteration 151 / 9800) loss: 1.654119
(Iteration 161 / 9800) loss: 1.645535
(Iteration 171 / 9800) loss: 1.623918
(Iteration 181 / 9800) loss: 1.743576
(Iteration 191 / 9800) loss: 1.617158
(Iteration 201 / 9800) loss: 1.642356
(Iteration 211 / 9800) loss: 1.832537
(Iteration 221 / 9800) loss: 1.590473
(Iteration 231 / 9800) loss: 1.741856
(Iteration 241 / 9800) loss: 1.791671
(Iteration 251 / 9800) loss: 1.609693
(Iteration 261 / 9800) loss: 1.779830
(Iteration 271 / 9800) loss: 1.663785
(Iteration 281 / 9800) loss: 1.752007
(Iteration 291 / 9800) loss: 1.468363
(Iteration 301 / 9800) loss: 1.698357
(Iteration 311 / 9800) loss: 1.571781
(Iteration 321 / 9800) loss: 1.670224
(Iteration 331 / 9800) loss: 1.423602
(Iteration 341 / 9800) loss: 1.666045
(Iteration 351 / 9800) loss: 1.621980
(Iteration 361 / 9800) loss: 1.712833
(Iteration 371 / 9800) loss: 1.493231
(Iteration 381 / 9800) loss: 1.681604
(Iteration 391 / 9800) loss: 1.469238
(Iteration 401 / 9800) loss: 1.641612
(Iteration 411 / 9800) loss: 1.781581
(Iteration 421 / 9800) loss: 1.571803
(Iteration 431 / 9800) loss: 1.601936
(Iteration 441 / 9800) loss: 1.589639
(Iteration 451 / 9800) loss: 1.729328
(Iteration 461 / 9800) loss: 1.389024
(Iteration 471 / 9800) loss: 1.408194
(Iteration 481 / 9800) loss: 1.577713
(Epoch 1 / 20) train acc: 0.455000; val_acc: 0.454000
(Iteration 491 / 9800) loss: 1.755789
(Iteration 501 / 9800) loss: 1.485227
(Iteration 511 / 9800) loss: 1.728271
(Iteration 521 / 9800) loss: 1.513419
```

```
(Iteration 531 / 9800) loss: 1.464570
(Iteration 541 / 9800) loss: 1.638251
(Iteration 551 / 9800) loss: 1.376767
(Iteration 561 / 9800) loss: 1.473990
(Iteration 571 / 9800) loss: 1.551640
(Iteration 581 / 9800) loss: 1.375947
(Iteration 591 / 9800) loss: 1.547475
(Iteration 601 / 9800) loss: 1.474233
(Iteration 611 / 9800) loss: 1.508042
(Iteration 621 / 9800) loss: 1.411110
(Iteration 631 / 9800) loss: 1.428669
(Iteration 641 / 9800) loss: 1.512971
(Iteration 651 / 9800) loss: 1.507610
(Iteration 661 / 9800) loss: 1.548395
(Iteration 671 / 9800) loss: 1.668384
(Iteration 681 / 9800) loss: 1.639356
(Iteration 691 / 9800) loss: 1.482689
(Iteration 701 / 9800) loss: 1.525736
(Iteration 711 / 9800) loss: 1.431824
(Iteration 721 / 9800) loss: 1.571454
(Iteration 731 / 9800) loss: 1.508847
(Iteration 741 / 9800) loss: 1.345769
(Iteration 751 / 9800) loss: 1.456724
(Iteration 761 / 9800) loss: 1.527775
(Iteration 771 / 9800) loss: 1.607720
(Iteration 781 / 9800) loss: 1.531189
(Iteration 791 / 9800) loss: 1.581243
(Iteration 801 / 9800) loss: 1.396695
(Iteration 811 / 9800) loss: 1.245169
(Iteration 821 / 9800) loss: 1.274486
(Iteration 831 / 9800) loss: 1.525931
(Iteration 841 / 9800) loss: 1.386012
(Iteration 851 / 9800) loss: 1.372727
(Iteration 861 / 9800) loss: 1.454317
(Iteration 871 / 9800) loss: 1.508159
(Iteration 881 / 9800) loss: 1.476650
(Iteration 891 / 9800) loss: 1.450071
(Iteration 901 / 9800) loss: 1.430833
(Iteration 911 / 9800) loss: 1.509891
(Iteration 921 / 9800) loss: 1.424904
(Iteration 931 / 9800) loss: 1.519244
(Iteration 941 / 9800) loss: 1.556086
(Iteration 951 / 9800) loss: 1.400419
(Iteration 961 / 9800) loss: 1.266520
(Iteration 971 / 9800) loss: 1.588027
(Epoch 2 / 20) train acc: 0.508000; val_acc: 0.483000
(Iteration 981 / 9800) loss: 1.476671
(Iteration 991 / 9800) loss: 1.288002
(Iteration 1001 / 9800) loss: 1.432288
(Iteration 1011 / 9800) loss: 1.463827
(Iteration 1021 / 9800) loss: 1.498597
(Iteration 1031 / 9800) loss: 1.523860
(Iteration 1041 / 9800) loss: 1.346940
(Iteration 1051 / 9800) loss: 1.473660
(Iteration 1061 / 9800) loss: 1.554438
(Iteration 1071 / 9800) loss: 1.333298
```

```
(Iteration 1081 / 9800) loss: 1.531535
(Iteration 1091 / 9800) loss: 1.351099
(Iteration 1101 / 9800) loss: 1.403005
(Iteration 1111 / 9800) loss: 1.486148
(Iteration 1121 / 9800) loss: 1.199843
(Iteration 1131 / 9800) loss: 1.336271
(Iteration 1141 / 9800) loss: 1.282978
(Iteration 1151 / 9800) loss: 1.535887
(Iteration 1161 / 9800) loss: 1.313601
(Iteration 1171 / 9800) loss: 1.393495
(Iteration 1181 / 9800) loss: 1.466843
(Iteration 1191 / 9800) loss: 1.385225
(Iteration 1201 / 9800) loss: 1.473338
(Iteration 1211 / 9800) loss: 1.347852
(Iteration 1221 / 9800) loss: 1.387998
(Iteration 1231 / 9800) loss: 1.424726
(Iteration 1241 / 9800) loss: 1.400841
(Iteration 1251 / 9800) loss: 1.280523
(Iteration 1261 / 9800) loss: 1.400368
(Iteration 1271 / 9800) loss: 1.291686
(Iteration 1281 / 9800) loss: 1.085247
(Iteration 1291 / 9800) loss: 1.256361
(Iteration 1301 / 9800) loss: 1.404233
(Iteration 1311 / 9800) loss: 1.522087
(Iteration 1321 / 9800) loss: 1.241769
(Iteration 1331 / 9800) loss: 1.314986
(Iteration 1341 / 9800) loss: 1.438212
(Iteration 1351 / 9800) loss: 1.379263
(Iteration 1361 / 9800) loss: 1.397201
(Iteration 1371 / 9800) loss: 1.425493
(Iteration 1381 / 9800) loss: 1.279438
(Iteration 1391 / 9800) loss: 1.269084
(Iteration 1401 / 9800) loss: 1.586483
(Iteration 1411 / 9800) loss: 1.224854
(Iteration 1421 / 9800) loss: 1.227602
(Iteration 1431 / 9800) loss: 1.216819
(Iteration 1441 / 9800) loss: 1.327004
(Iteration 1451 / 9800) loss: 1.345085
(Iteration 1461 / 9800) loss: 1.355550
(Epoch 3 / 20) train acc: 0.546000; val_acc: 0.501000
(Iteration 1471 / 9800) loss: 1.245834
(Iteration 1481 / 9800) loss: 1.345647
(Iteration 1491 / 9800) loss: 1.421563
(Iteration 1501 / 9800) loss: 1.491060
(Iteration 1511 / 9800) loss: 1.362529
(Iteration 1521 / 9800) loss: 1.328036
(Iteration 1531 / 9800) loss: 1.379741
(Iteration 1541 / 9800) loss: 1.385374
(Iteration 1551 / 9800) loss: 1.142256
(Iteration 1561 / 9800) loss: 1.315898
(Iteration 1571 / 9800) loss: 1.423627
(Iteration 1581 / 9800) loss: 1.423694
(Iteration 1591 / 9800) loss: 1.599280
(Iteration 1601 / 9800) loss: 1.249422
(Iteration 1611 / 9800) loss: 1.254416
(Iteration 1621 / 9800) loss: 1.290352
```

```
(Iteration 1631 / 9800) loss: 1.267833
(Iteration 1641 / 9800) loss: 1.321321
(Iteration 1651 / 9800) loss: 1.380142
(Iteration 1661 / 9800) loss: 1.391724
(Iteration 1671 / 9800) loss: 1.238736
(Iteration 1681 / 9800) loss: 1.122107
(Iteration 1691 / 9800) loss: 1.277312
(Iteration 1701 / 9800) loss: 1.417275
(Iteration 1711 / 9800) loss: 1.334315
(Iteration 1721 / 9800) loss: 1.288414
(Iteration 1731 / 9800) loss: 1.152508
(Iteration 1741 / 9800) loss: 1.209650
(Iteration 1751 / 9800) loss: 1.464872
(Iteration 1761 / 9800) loss: 1.289903
(Iteration 1771 / 9800) loss: 1.152915
(Iteration 1781 / 9800) loss: 1.359977
(Iteration 1791 / 9800) loss: 1.452991
(Iteration 1801 / 9800) loss: 1.309753
(Iteration 1811 / 9800) loss: 1.164795
(Iteration 1821 / 9800) loss: 1.056199
(Iteration 1831 / 9800) loss: 1.402317
(Iteration 1841 / 9800) loss: 1.272056
(Iteration 1851 / 9800) loss: 1.372723
(Iteration 1861 / 9800) loss: 1.110256
(Iteration 1871 / 9800) loss: 1.237975
(Iteration 1881 / 9800) loss: 1.427127
(Iteration 1891 / 9800) loss: 1.220772
(Iteration 1901 / 9800) loss: 1.117505
(Iteration 1911 / 9800) loss: 1.231343
(Iteration 1921 / 9800) loss: 1.367520
(Iteration 1931 / 9800) loss: 1.305547
(Iteration 1941 / 9800) loss: 1.083777
(Iteration 1951 / 9800) loss: 1.339535
(Epoch 4 / 20) train acc: 0.541000; val_acc: 0.513000
(Iteration 1961 / 9800) loss: 1.167282
(Iteration 1971 / 9800) loss: 1.438934
(Iteration 1981 / 9800) loss: 1.318816
(Iteration 1991 / 9800) loss: 1.232330
(Iteration 2001 / 9800) loss: 1.335854
(Iteration 2011 / 9800) loss: 1.195528
(Iteration 2021 / 9800) loss: 1.251077
(Iteration 2031 / 9800) loss: 1.494910
(Iteration 2041 / 9800) loss: 1.381534
(Iteration 2051 / 9800) loss: 1.241804
(Iteration 2061 / 9800) loss: 1.339859
(Iteration 2071 / 9800) loss: 1.275170
(Iteration 2081 / 9800) loss: 1.344104
(Iteration 2091 / 9800) loss: 1.233493
(Iteration 2101 / 9800) loss: 1.230569
(Iteration 2111 / 9800) loss: 1.323297
(Iteration 2121 / 9800) loss: 1.177891
(Iteration 2131 / 9800) loss: 1.418129
(Iteration 2141 / 9800) loss: 1.385728
(Iteration 2151 / 9800) loss: 1.254921
(Iteration 2161 / 9800) loss: 1.115170
(Iteration 2171 / 9800) loss: 1.288903
```

```
(Iteration 2181 / 9800) loss: 1.350088
(Iteration 2191 / 9800) loss: 1.419715
(Iteration 2201 / 9800) loss: 1.255964
(Iteration 2211 / 9800) loss: 1.156236
(Iteration 2221 / 9800) loss: 1.381212
(Iteration 2231 / 9800) loss: 1.238229
(Iteration 2241 / 9800) loss: 1.475892
(Iteration 2251 / 9800) loss: 1.280353
(Iteration 2261 / 9800) loss: 1.143529
(Iteration 2271 / 9800) loss: 1.091913
(Iteration 2281 / 9800) loss: 1.154643
(Iteration 2291 / 9800) loss: 1.380597
(Iteration 2301 / 9800) loss: 1.274749
(Iteration 2311 / 9800) loss: 1.327720
(Iteration 2321 / 9800) loss: 1.186956
(Iteration 2331 / 9800) loss: 1.157865
(Iteration 2341 / 9800) loss: 1.081448
(Iteration 2351 / 9800) loss: 1.019765
(Iteration 2361 / 9800) loss: 1.341794
(Iteration 2371 / 9800) loss: 1.337426
(Iteration 2381 / 9800) loss: 1.356621
(Iteration 2391 / 9800) loss: 1.286818
(Iteration 2401 / 9800) loss: 1.139214
(Iteration 2411 / 9800) loss: 1.187765
(Iteration 2421 / 9800) loss: 1.148632
(Iteration 2431 / 9800) loss: 1.194631
(Iteration 2441 / 9800) loss: 1.147632
(Epoch 5 / 20) train acc: 0.570000; val_acc: 0.499000
(Iteration 2451 / 9800) loss: 1.351823
(Iteration 2461 / 9800) loss: 1.323100
(Iteration 2471 / 9800) loss: 1.184994
(Iteration 2481 / 9800) loss: 1.239742
(Iteration 2491 / 9800) loss: 1.184222
(Iteration 2501 / 9800) loss: 1.109152
(Iteration 2511 / 9800) loss: 1.099257
(Iteration 2521 / 9800) loss: 1.171722
(Iteration 2531 / 9800) loss: 1.203512
(Iteration 2541 / 9800) loss: 1.206762
(Iteration 2551 / 9800) loss: 1.026705
(Iteration 2561 / 9800) loss: 1.370455
(Iteration 2571 / 9800) loss: 1.138838
(Iteration 2581 / 9800) loss: 1.201225
(Iteration 2591 / 9800) loss: 1.382983
(Iteration 2601 / 9800) loss: 1.068614
(Iteration 2611 / 9800) loss: 1.300780
(Iteration 2621 / 9800) loss: 1.292400
(Iteration 2631 / 9800) loss: 1.219516
(Iteration 2641 / 9800) loss: 1.104560
(Iteration 2651 / 9800) loss: 1.323212
(Iteration 2661 / 9800) loss: 1.041291
(Iteration 2671 / 9800) loss: 1.071948
(Iteration 2681 / 9800) loss: 1.138603
(Iteration 2691 / 9800) loss: 1.073794
(Iteration 2701 / 9800) loss: 1.139561
(Iteration 2711 / 9800) loss: 0.986153
(Iteration 2721 / 9800) loss: 1.099700
```

```
(Iteration 2731 / 9800) loss: 1.051041
(Iteration 2741 / 9800) loss: 1.192400
(Iteration 2751 / 9800) loss: 1.165448
(Iteration 2761 / 9800) loss: 1.500418
(Iteration 2771 / 9800) loss: 1.142388
(Iteration 2781 / 9800) loss: 1.085932
(Iteration 2791 / 9800) loss: 1.190146
(Iteration 2801 / 9800) loss: 0.944767
(Iteration 2811 / 9800) loss: 1.171080
(Iteration 2821 / 9800) loss: 1.233232
(Iteration 2831 / 9800) loss: 1.238354
(Iteration 2841 / 9800) loss: 1.237606
(Iteration 2851 / 9800) loss: 1.338535
(Iteration 2861 / 9800) loss: 1.344883
(Iteration 2871 / 9800) loss: 1.154288
(Iteration 2881 / 9800) loss: 1.100680
(Iteration 2891 / 9800) loss: 1.166691
(Iteration 2901 / 9800) loss: 1.074254
(Iteration 2911 / 9800) loss: 1.061827
(Iteration 2921 / 9800) loss: 0.958209
(Iteration 2931 / 9800) loss: 1.204298
(Epoch 6 / 20) train acc: 0.594000; val_acc: 0.519000
(Iteration 2941 / 9800) loss: 1.235023
(Iteration 2951 / 9800) loss: 1.151577
(Iteration 2961 / 9800) loss: 1.202734
(Iteration 2971 / 9800) loss: 1.056746
(Iteration 2981 / 9800) loss: 1.096857
(Iteration 2991 / 9800) loss: 1.125515
(Iteration 3001 / 9800) loss: 1.116349
(Iteration 3011 / 9800) loss: 1.125724
(Iteration 3021 / 9800) loss: 1.105375
(Iteration 3031 / 9800) loss: 1.095838
(Iteration 3041 / 9800) loss: 1.136922
(Iteration 3051 / 9800) loss: 1.095312
(Iteration 3061 / 9800) loss: 1.208858
(Iteration 3071 / 9800) loss: 1.214527
(Iteration 3081 / 9800) loss: 1.044673
(Iteration 3091 / 9800) loss: 1.261836
(Iteration 3101 / 9800) loss: 1.024950
(Iteration 3111 / 9800) loss: 1.378130
(Iteration 3121 / 9800) loss: 1.070434
(Iteration 3131 / 9800) loss: 1.139299
(Iteration 3141 / 9800) loss: 1.191006
(Iteration 3151 / 9800) loss: 1.075201
(Iteration 3161 / 9800) loss: 1.172091
(Iteration 3171 / 9800) loss: 1.164444
(Iteration 3181 / 9800) loss: 1.213513
(Iteration 3191 / 9800) loss: 1.215212
(Iteration 3201 / 9800) loss: 1.172483
(Iteration 3211 / 9800) loss: 1.150347
(Iteration 3221 / 9800) loss: 1.046196
(Iteration 3231 / 9800) loss: 1.147516
(Iteration 3241 / 9800) loss: 1.149188
(Iteration 3251 / 9800) loss: 1.238455
(Iteration 3261 / 9800) loss: 1.294803
(Iteration 3271 / 9800) loss: 1.089021
```

```
(Iteration 3281 / 9800) loss: 1.210028
(Iteration 3291 / 9800) loss: 1.143076
(Iteration 3301 / 9800) loss: 1.059879
(Iteration 3311 / 9800) loss: 1.017949
(Iteration 3321 / 9800) loss: 1.233673
(Iteration 3331 / 9800) loss: 1.117520
(Iteration 3341 / 9800) loss: 1.241708
(Iteration 3351 / 9800) loss: 1.394216
(Iteration 3361 / 9800) loss: 1.376558
(Iteration 3371 / 9800) loss: 1.145555
(Iteration 3381 / 9800) loss: 1.212833
(Iteration 3391 / 9800) loss: 1.183473
(Iteration 3401 / 9800) loss: 1.081811
(Iteration 3411 / 9800) loss: 1.164537
(Iteration 3421 / 9800) loss: 1.012572
(Epoch 7 / 20) train acc: 0.588000; val_acc: 0.516000
(Iteration 3431 / 9800) loss: 1.126703
(Iteration 3441 / 9800) loss: 1.101405
(Iteration 3451 / 9800) loss: 0.967516
(Iteration 3461 / 9800) loss: 1.064929
(Iteration 3471 / 9800) loss: 1.102843
(Iteration 3481 / 9800) loss: 1.318438
(Iteration 3491 / 9800) loss: 1.147158
(Iteration 3501 / 9800) loss: 1.285311
(Iteration 3511 / 9800) loss: 1.248535
(Iteration 3521 / 9800) loss: 1.059287
(Iteration 3531 / 9800) loss: 1.114023
(Iteration 3541 / 9800) loss: 1.187169
(Iteration 3551 / 9800) loss: 1.151922
(Iteration 3561 / 9800) loss: 0.938457
(Iteration 3571 / 9800) loss: 1.132080
(Iteration 3581 / 9800) loss: 1.260066
(Iteration 3591 / 9800) loss: 1.073154
(Iteration 3601 / 9800) loss: 1.005667
(Iteration 3611 / 9800) loss: 1.093877
(Iteration 3621 / 9800) loss: 0.864351
(Iteration 3631 / 9800) loss: 0.995371
(Iteration 3641 / 9800) loss: 1.292234
(Iteration 3651 / 9800) loss: 0.847873
(Iteration 3661 / 9800) loss: 1.144024
(Iteration 3671 / 9800) loss: 1.102591
(Iteration 3681 / 9800) loss: 1.207792
(Iteration 3691 / 9800) loss: 0.866027
(Iteration 3701 / 9800) loss: 1.011805
(Iteration 3711 / 9800) loss: 1.243870
(Iteration 3721 / 9800) loss: 1.024391
(Iteration 3731 / 9800) loss: 1.228998
(Iteration 3741 / 9800) loss: 1.154805
(Iteration 3751 / 9800) loss: 1.113036
(Iteration 3761 / 9800) loss: 1.155537
(Iteration 3771 / 9800) loss: 1.115245
(Iteration 3781 / 9800) loss: 1.199820
(Iteration 3791 / 9800) loss: 0.986382
(Iteration 3801 / 9800) loss: 1.241569
(Iteration 3811 / 9800) loss: 1.116719
(Iteration 3821 / 9800) loss: 1.089074
```

```
(Iteration 3831 / 9800) loss: 1.208819
(Iteration 3841 / 9800) loss: 1.167415
(Iteration 3851 / 9800) loss: 1.289346
(Iteration 3861 / 9800) loss: 0.856649
(Iteration 3871 / 9800) loss: 1.077684
(Iteration 3881 / 9800) loss: 0.955542
(Iteration 3891 / 9800) loss: 0.973487
(Iteration 3901 / 9800) loss: 1.067339
(Iteration 3911 / 9800) loss: 1.152389
(Epoch 8 / 20) train acc: 0.633000; val_acc: 0.511000
(Iteration 3921 / 9800) loss: 1.110580
(Iteration 3931 / 9800) loss: 1.217930
(Iteration 3941 / 9800) loss: 1.019999
(Iteration 3951 / 9800) loss: 1.158196
(Iteration 3961 / 9800) loss: 0.980334
(Iteration 3971 / 9800) loss: 0.968163
(Iteration 3981 / 9800) loss: 1.024047
(Iteration 3991 / 9800) loss: 0.987809
(Iteration 4001 / 9800) loss: 1.053094
(Iteration 4011 / 9800) loss: 1.011108
(Iteration 4021 / 9800) loss: 1.075417
(Iteration 4031 / 9800) loss: 1.082791
(Iteration 4041 / 9800) loss: 1.173363
(Iteration 4051 / 9800) loss: 1.088959
(Iteration 4061 / 9800) loss: 1.065405
(Iteration 4071 / 9800) loss: 1.006720
(Iteration 4081 / 9800) loss: 0.853418
(Iteration 4091 / 9800) loss: 0.979935
(Iteration 4101 / 9800) loss: 1.018717
(Iteration 4111 / 9800) loss: 1.128008
(Iteration 4121 / 9800) loss: 1.034482
(Iteration 4131 / 9800) loss: 1.081799
(Iteration 4141 / 9800) loss: 1.068148
(Iteration 4151 / 9800) loss: 1.066267
(Iteration 4161 / 9800) loss: 0.999982
(Iteration 4171 / 9800) loss: 0.837339
(Iteration 4181 / 9800) loss: 1.050571
(Iteration 4191 / 9800) loss: 1.095693
(Iteration 4201 / 9800) loss: 0.816809
(Iteration 4211 / 9800) loss: 0.962551
(Iteration 4221 / 9800) loss: 0.885475
(Iteration 4231 / 9800) loss: 0.841674
(Iteration 4241 / 9800) loss: 1.090313
(Iteration 4251 / 9800) loss: 1.054244
(Iteration 4261 / 9800) loss: 1.223532
(Iteration 4271 / 9800) loss: 1.102773
(Iteration 4281 / 9800) loss: 0.997905
(Iteration 4291 / 9800) loss: 1.158605
(Iteration 4301 / 9800) loss: 0.827481
(Iteration 4311 / 9800) loss: 0.952205
(Iteration 4321 / 9800) loss: 1.129143
(Iteration 4331 / 9800) loss: 0.927160
(Iteration 4341 / 9800) loss: 0.878172
(Iteration 4351 / 9800) loss: 0.959471
(Iteration 4361 / 9800) loss: 1.046034
(Iteration 4371 / 9800) loss: 1.071348
```

```
(Iteration 4381 / 9800) loss: 1.155551
(Iteration 4391 / 9800) loss: 1.010665
(Iteration 4401 / 9800) loss: 1.034425
(Epoch 9 / 20) train acc: 0.633000; val_acc: 0.507000
(Iteration 4411 / 9800) loss: 1.064040
(Iteration 4421 / 9800) loss: 0.990878
(Iteration 4431 / 9800) loss: 0.973593
(Iteration 4441 / 9800) loss: 1.116501
(Iteration 4451 / 9800) loss: 1.032293
(Iteration 4461 / 9800) loss: 1.124811
(Iteration 4471 / 9800) loss: 1.034922
(Iteration 4481 / 9800) loss: 1.065864
(Iteration 4491 / 9800) loss: 1.007038
(Iteration 4501 / 9800) loss: 0.998353
(Iteration 4511 / 9800) loss: 1.264745
(Iteration 4521 / 9800) loss: 0.941287
(Iteration 4531 / 9800) loss: 1.074142
(Iteration 4541 / 9800) loss: 0.867547
(Iteration 4551 / 9800) loss: 0.817419
(Iteration 4561 / 9800) loss: 1.200279
(Iteration 4571 / 9800) loss: 1.052691
(Iteration 4581 / 9800) loss: 1.110576
(Iteration 4591 / 9800) loss: 0.896724
(Iteration 4601 / 9800) loss: 0.982789
(Iteration 4611 / 9800) loss: 1.020063
(Iteration 4621 / 9800) loss: 1.295473
(Iteration 4631 / 9800) loss: 1.050335
(Iteration 4641 / 9800) loss: 0.987751
(Iteration 4651 / 9800) loss: 1.069991
(Iteration 4661 / 9800) loss: 0.829466
(Iteration 4671 / 9800) loss: 0.968191
(Iteration 4681 / 9800) loss: 1.185383
(Iteration 4691 / 9800) loss: 1.221613
(Iteration 4701 / 9800) loss: 0.913588
(Iteration 4711 / 9800) loss: 0.797655
(Iteration 4721 / 9800) loss: 0.971148
(Iteration 4731 / 9800) loss: 0.888165
(Iteration 4741 / 9800) loss: 0.846889
(Iteration 4751 / 9800) loss: 1.107543
(Iteration 4761 / 9800) loss: 1.044931
(Iteration 4771 / 9800) loss: 1.002691
(Iteration 4781 / 9800) loss: 1.068140
(Iteration 4791 / 9800) loss: 1.084906
(Iteration 4801 / 9800) loss: 0.871990
(Iteration 4811 / 9800) loss: 0.999907
(Iteration 4821 / 9800) loss: 1.042069
(Iteration 4831 / 9800) loss: 0.886474
(Iteration 4841 / 9800) loss: 1.124677
(Iteration 4851 / 9800) loss: 1.067717
(Iteration 4861 / 9800) loss: 1.062625
(Iteration 4871 / 9800) loss: 0.866862
(Iteration 4881 / 9800) loss: 0.926206
(Iteration 4891 / 9800) loss: 1.033800
(Epoch 10 / 20) train acc: 0.655000; val_acc: 0.498000
(Iteration 4901 / 9800) loss: 1.196615
(Iteration 4911 / 9800) loss: 0.940998
```

```
(Iteration 4921 / 9800) loss: 0.966525
(Iteration 4931 / 9800) loss: 1.067281
(Iteration 4941 / 9800) loss: 1.134524
(Iteration 4951 / 9800) loss: 1.019508
(Iteration 4961 / 9800) loss: 0.910535
(Iteration 4971 / 9800) loss: 0.913771
(Iteration 4981 / 9800) loss: 0.848318
(Iteration 4991 / 9800) loss: 0.849713
(Iteration 5001 / 9800) loss: 0.976090
(Iteration 5011 / 9800) loss: 1.082327
(Iteration 5021 / 9800) loss: 0.912977
(Iteration 5031 / 9800) loss: 1.126119
(Iteration 5041 / 9800) loss: 0.987612
(Iteration 5051 / 9800) loss: 1.251975
(Iteration 5061 / 9800) loss: 1.003866
(Iteration 5071 / 9800) loss: 0.613926
(Iteration 5081 / 9800) loss: 1.113141
(Iteration 5091 / 9800) loss: 1.043489
(Iteration 5101 / 9800) loss: 1.086584
(Iteration 5111 / 9800) loss: 0.995421
(Iteration 5121 / 9800) loss: 1.009711
(Iteration 5131 / 9800) loss: 1.118372
(Iteration 5141 / 9800) loss: 1.080592
(Iteration 5151 / 9800) loss: 1.109901
(Iteration 5161 / 9800) loss: 0.986837
(Iteration 5171 / 9800) loss: 1.044823
(Iteration 5181 / 9800) loss: 1.097604
(Iteration 5191 / 9800) loss: 0.994191
(Iteration 5201 / 9800) loss: 1.035136
(Iteration 5211 / 9800) loss: 1.173730
(Iteration 5221 / 9800) loss: 1.061021
(Iteration 5231 / 9800) loss: 0.789861
(Iteration 5241 / 9800) loss: 1.051217
(Iteration 5251 / 9800) loss: 1.023275
(Iteration 5261 / 9800) loss: 1.013110
(Iteration 5271 / 9800) loss: 0.923426
(Iteration 5281 / 9800) loss: 0.955747
(Iteration 5291 / 9800) loss: 0.997954
(Iteration 5301 / 9800) loss: 0.968057
(Iteration 5311 / 9800) loss: 0.842489
(Iteration 5321 / 9800) loss: 0.933340
(Iteration 5331 / 9800) loss: 0.870908
(Iteration 5341 / 9800) loss: 0.960548
(Iteration 5351 / 9800) loss: 1.034818
(Iteration 5361 / 9800) loss: 1.010375
(Iteration 5371 / 9800) loss: 0.965839
(Iteration 5381 / 9800) loss: 0.983799
(Epoch 11 / 20) train acc: 0.678000; val_acc: 0.510000
(Iteration 5391 / 9800) loss: 0.974062
(Iteration 5401 / 9800) loss: 0.758411
(Iteration 5411 / 9800) loss: 1.054531
(Iteration 5421 / 9800) loss: 0.998426
(Iteration 5431 / 9800) loss: 0.885930
(Iteration 5441 / 9800) loss: 0.861065
(Iteration 5451 / 9800) loss: 0.913299
(Iteration 5461 / 9800) loss: 1.180276
```

```
(Iteration 5471 / 9800) loss: 0.839506
(Iteration 5481 / 9800) loss: 0.910314
(Iteration 5491 / 9800) loss: 0.995421
(Iteration 5501 / 9800) loss: 0.931136
(Iteration 5511 / 9800) loss: 0.981049
(Iteration 5521 / 9800) loss: 1.081133
(Iteration 5531 / 9800) loss: 1.049714
(Iteration 5541 / 9800) loss: 0.943937
(Iteration 5551 / 9800) loss: 1.081251
(Iteration 5561 / 9800) loss: 1.123401
(Iteration 5571 / 9800) loss: 1.024085
(Iteration 5581 / 9800) loss: 0.927033
(Iteration 5591 / 9800) loss: 1.010984
(Iteration 5601 / 9800) loss: 1.078381
(Iteration 5611 / 9800) loss: 0.797791
(Iteration 5621 / 9800) loss: 0.886528
(Iteration 5631 / 9800) loss: 0.922667
(Iteration 5641 / 9800) loss: 0.941367
(Iteration 5651 / 9800) loss: 0.843206
(Iteration 5661 / 9800) loss: 0.839444
(Iteration 5671 / 9800) loss: 0.881875
(Iteration 5681 / 9800) loss: 1.091468
(Iteration 5691 / 9800) loss: 0.874410
(Iteration 5701 / 9800) loss: 0.796802
(Iteration 5711 / 9800) loss: 0.775673
(Iteration 5721 / 9800) loss: 0.952441
(Iteration 5731 / 9800) loss: 0.916567
(Iteration 5741 / 9800) loss: 0.885789
(Iteration 5751 / 9800) loss: 1.097942
(Iteration 5761 / 9800) loss: 0.946782
(Iteration 5771 / 9800) loss: 1.034226
(Iteration 5781 / 9800) loss: 0.965112
(Iteration 5791 / 9800) loss: 0.701352
(Iteration 5801 / 9800) loss: 0.766240
(Iteration 5811 / 9800) loss: 1.095407
(Iteration 5821 / 9800) loss: 0.972350
(Iteration 5831 / 9800) loss: 0.892998
(Iteration 5841 / 9800) loss: 0.864552
(Iteration 5851 / 9800) loss: 0.817898
(Iteration 5861 / 9800) loss: 1.015913
(Iteration 5871 / 9800) loss: 0.884251
(Epoch 12 / 20) train acc: 0.686000; val_acc: 0.513000
(Iteration 5881 / 9800) loss: 0.939642
(Iteration 5891 / 9800) loss: 0.734507
(Iteration 5901 / 9800) loss: 0.842008
(Iteration 5911 / 9800) loss: 0.809396
(Iteration 5921 / 9800) loss: 0.753707
(Iteration 5931 / 9800) loss: 0.969566
(Iteration 5941 / 9800) loss: 0.844427
(Iteration 5951 / 9800) loss: 0.886055
(Iteration 5961 / 9800) loss: 0.999003
(Iteration 5971 / 9800) loss: 0.830278
(Iteration 5981 / 9800) loss: 0.783678
(Iteration 5991 / 9800) loss: 0.952035
(Iteration 6001 / 9800) loss: 0.930499
(Iteration 6011 / 9800) loss: 0.971019
```

```
(Iteration 6021 / 9800) loss: 0.816636
(Iteration 6031 / 9800) loss: 0.951900
(Iteration 6041 / 9800) loss: 0.864198
(Iteration 6051 / 9800) loss: 0.768943
(Iteration 6061 / 9800) loss: 0.956829
(Iteration 6071 / 9800) loss: 0.783845
(Iteration 6081 / 9800) loss: 0.910374
(Iteration 6091 / 9800) loss: 0.780432
(Iteration 6101 / 9800) loss: 0.898892
(Iteration 6111 / 9800) loss: 0.838101
(Iteration 6121 / 9800) loss: 0.950702
(Iteration 6131 / 9800) loss: 1.057025
(Iteration 6141 / 9800) loss: 1.067681
(Iteration 6151 / 9800) loss: 0.867315
(Iteration 6161 / 9800) loss: 1.109836
(Iteration 6171 / 9800) loss: 0.931642
(Iteration 6181 / 9800) loss: 0.926901
(Iteration 6191 / 9800) loss: 0.970821
(Iteration 6201 / 9800) loss: 1.010480
(Iteration 6211 / 9800) loss: 1.017680
(Iteration 6221 / 9800) loss: 0.882496
(Iteration 6231 / 9800) loss: 0.971359
(Iteration 6241 / 9800) loss: 0.969547
(Iteration 6251 / 9800) loss: 0.994330
(Iteration 6261 / 9800) loss: 0.880249
(Iteration 6271 / 9800) loss: 0.902311
(Iteration 6281 / 9800) loss: 0.852894
(Iteration 6291 / 9800) loss: 0.876608
(Iteration 6301 / 9800) loss: 0.853413
(Iteration 6311 / 9800) loss: 0.793769
(Iteration 6321 / 9800) loss: 0.888272
(Iteration 6331 / 9800) loss: 1.008677
(Iteration 6341 / 9800) loss: 0.724882
(Iteration 6351 / 9800) loss: 0.794097
(Iteration 6361 / 9800) loss: 0.804445
(Epoch 13 / 20) train acc: 0.682000; val_acc: 0.515000
(Iteration 6371 / 9800) loss: 0.882058
(Iteration 6381 / 9800) loss: 0.768740
(Iteration 6391 / 9800) loss: 0.817273
(Iteration 6401 / 9800) loss: 0.938625
(Iteration 6411 / 9800) loss: 0.920957
(Iteration 6421 / 9800) loss: 0.865756
(Iteration 6431 / 9800) loss: 0.900197
(Iteration 6441 / 9800) loss: 1.151324
(Iteration 6451 / 9800) loss: 0.715478
(Iteration 6461 / 9800) loss: 0.811591
(Iteration 6471 / 9800) loss: 0.949442
(Iteration 6481 / 9800) loss: 0.997060
(Iteration 6491 / 9800) loss: 0.996502
(Iteration 6501 / 9800) loss: 0.925518
(Iteration 6511 / 9800) loss: 0.757595
(Iteration 6521 / 9800) loss: 0.840178
(Iteration 6531 / 9800) loss: 0.847318
(Iteration 6541 / 9800) loss: 0.871878
(Iteration 6551 / 9800) loss: 0.880157
(Iteration 6561 / 9800) loss: 1.011983
```

```
(Iteration 6571 / 9800) loss: 1.033732
(Iteration 6581 / 9800) loss: 0.932693
(Iteration 6591 / 9800) loss: 0.836586
(Iteration 6601 / 9800) loss: 0.680791
(Iteration 6611 / 9800) loss: 0.957559
(Iteration 6621 / 9800) loss: 1.048484
(Iteration 6631 / 9800) loss: 0.998396
(Iteration 6641 / 9800) loss: 0.967136
(Iteration 6651 / 9800) loss: 0.917413
(Iteration 6661 / 9800) loss: 0.811649
(Iteration 6671 / 9800) loss: 0.820953
(Iteration 6681 / 9800) loss: 0.899232
(Iteration 6691 / 9800) loss: 1.054467
(Iteration 6701 / 9800) loss: 0.834709
(Iteration 6711 / 9800) loss: 0.679216
(Iteration 6721 / 9800) loss: 0.957630
(Iteration 6731 / 9800) loss: 0.997459
(Iteration 6741 / 9800) loss: 1.013980
(Iteration 6751 / 9800) loss: 0.986151
(Iteration 6761 / 9800) loss: 0.929330
(Iteration 6771 / 9800) loss: 0.755469
(Iteration 6781 / 9800) loss: 0.867278
(Iteration 6791 / 9800) loss: 0.828597
(Iteration 6801 / 9800) loss: 0.762709
(Iteration 6811 / 9800) loss: 0.926288
(Iteration 6821 / 9800) loss: 1.028349
(Iteration 6831 / 9800) loss: 0.982469
(Iteration 6841 / 9800) loss: 0.808823
(Iteration 6851 / 9800) loss: 0.911073
(Epoch 14 / 20) train acc: 0.665000; val_acc: 0.498000
(Iteration 6861 / 9800) loss: 0.973637
(Iteration 6871 / 9800) loss: 0.863585
(Iteration 6881 / 9800) loss: 1.042866
(Iteration 6891 / 9800) loss: 0.862814
(Iteration 6901 / 9800) loss: 0.996856
(Iteration 6911 / 9800) loss: 0.913571
(Iteration 6921 / 9800) loss: 0.947338
(Iteration 6931 / 9800) loss: 0.944871
(Iteration 6941 / 9800) loss: 1.008145
(Iteration 6951 / 9800) loss: 0.809979
(Iteration 6961 / 9800) loss: 0.980964
(Iteration 6971 / 9800) loss: 0.908692
(Iteration 6981 / 9800) loss: 0.855398
(Iteration 6991 / 9800) loss: 0.770794
(Iteration 7001 / 9800) loss: 0.891020
(Iteration 7011 / 9800) loss: 0.808356
(Iteration 7021 / 9800) loss: 0.822000
(Iteration 7031 / 9800) loss: 0.803978
(Iteration 7041 / 9800) loss: 0.841338
(Iteration 7051 / 9800) loss: 0.960743
(Iteration 7061 / 9800) loss: 0.859569
(Iteration 7071 / 9800) loss: 0.791502
(Iteration 7081 / 9800) loss: 0.963803
(Iteration 7091 / 9800) loss: 0.749703
(Iteration 7101 / 9800) loss: 0.856307
(Iteration 7111 / 9800) loss: 0.952457
```

```
(Iteration 7121 / 9800) loss: 0.850381
(Iteration 7131 / 9800) loss: 0.940020
(Iteration 7141 / 9800) loss: 0.957537
(Iteration 7151 / 9800) loss: 1.026274
(Iteration 7161 / 9800) loss: 1.150635
(Iteration 7171 / 9800) loss: 0.974438
(Iteration 7181 / 9800) loss: 0.978651
(Iteration 7191 / 9800) loss: 0.708581
(Iteration 7201 / 9800) loss: 0.946347
(Iteration 7211 / 9800) loss: 0.823376
(Iteration 7221 / 9800) loss: 0.710964
(Iteration 7231 / 9800) loss: 0.969588
(Iteration 7241 / 9800) loss: 0.855370
(Iteration 7251 / 9800) loss: 0.723886
(Iteration 7261 / 9800) loss: 0.809842
(Iteration 7271 / 9800) loss: 0.765288
(Iteration 7281 / 9800) loss: 0.830187
(Iteration 7291 / 9800) loss: 0.909273
(Iteration 7301 / 9800) loss: 1.023312
(Iteration 7311 / 9800) loss: 0.859834
(Iteration 7321 / 9800) loss: 0.537578
(Iteration 7331 / 9800) loss: 1.017617
(Iteration 7341 / 9800) loss: 1.020733
(Epoch 15 / 20) train acc: 0.709000; val_acc: 0.501000
(Iteration 7351 / 9800) loss: 0.836357
(Iteration 7361 / 9800) loss: 0.901268
(Iteration 7371 / 9800) loss: 0.765752
(Iteration 7381 / 9800) loss: 0.862170
(Iteration 7391 / 9800) loss: 0.813861
(Iteration 7401 / 9800) loss: 0.895636
(Iteration 7411 / 9800) loss: 0.903577
(Iteration 7421 / 9800) loss: 0.861770
(Iteration 7431 / 9800) loss: 0.733075
(Iteration 7441 / 9800) loss: 0.885267
(Iteration 7451 / 9800) loss: 0.814259
(Iteration 7461 / 9800) loss: 0.836020
(Iteration 7471 / 9800) loss: 0.693926
(Iteration 7481 / 9800) loss: 0.894061
(Iteration 7491 / 9800) loss: 0.836126
(Iteration 7501 / 9800) loss: 0.905481
(Iteration 7511 / 9800) loss: 0.787363
(Iteration 7521 / 9800) loss: 1.140097
(Iteration 7531 / 9800) loss: 0.821119
(Iteration 7541 / 9800) loss: 0.854043
(Iteration 7551 / 9800) loss: 0.787783
(Iteration 7561 / 9800) loss: 0.849964
(Iteration 7571 / 9800) loss: 0.787478
(Iteration 7581 / 9800) loss: 0.931193
(Iteration 7591 / 9800) loss: 0.822869
(Iteration 7601 / 9800) loss: 0.609961
(Iteration 7611 / 9800) loss: 0.639507
(Iteration 7621 / 9800) loss: 0.794223
(Iteration 7631 / 9800) loss: 0.822842
(Iteration 7641 / 9800) loss: 0.736065
(Iteration 7651 / 9800) loss: 0.731141
(Iteration 7661 / 9800) loss: 0.864216
```

```
(Iteration 7671 / 9800) loss: 0.687184
(Iteration 7681 / 9800) loss: 0.863336
(Iteration 7691 / 9800) loss: 0.731150
(Iteration 7701 / 9800) loss: 0.972840
(Iteration 7711 / 9800) loss: 0.825283
(Iteration 7721 / 9800) loss: 0.622582
(Iteration 7731 / 9800) loss: 0.768965
(Iteration 7741 / 9800) loss: 0.695182
(Iteration 7751 / 9800) loss: 1.018550
(Iteration 7761 / 9800) loss: 0.984811
(Iteration 7771 / 9800) loss: 0.720340
(Iteration 7781 / 9800) loss: 0.908348
(Iteration 7791 / 9800) loss: 0.846390
(Iteration 7801 / 9800) loss: 1.169997
(Iteration 7811 / 9800) loss: 0.824165
(Iteration 7821 / 9800) loss: 0.892967
(Iteration 7831 / 9800) loss: 0.688355
(Epoch 16 / 20) train acc: 0.720000; val_acc: 0.518000
(Iteration 7841 / 9800) loss: 0.680372
(Iteration 7851 / 9800) loss: 0.942279
(Iteration 7861 / 9800) loss: 0.556268
(Iteration 7871 / 9800) loss: 0.807672
(Iteration 7881 / 9800) loss: 0.821774
(Iteration 7891 / 9800) loss: 0.973928
(Iteration 7901 / 9800) loss: 0.943179
(Iteration 7911 / 9800) loss: 0.919872
(Iteration 7921 / 9800) loss: 0.895609
(Iteration 7931 / 9800) loss: 0.637186
(Iteration 7941 / 9800) loss: 0.775248
(Iteration 7951 / 9800) loss: 0.887665
(Iteration 7961 / 9800) loss: 0.752427
(Iteration 7971 / 9800) loss: 0.738051
(Iteration 7981 / 9800) loss: 0.841195
(Iteration 7991 / 9800) loss: 0.859497
(Iteration 8001 / 9800) loss: 0.703537
(Iteration 8011 / 9800) loss: 0.837302
(Iteration 8021 / 9800) loss: 0.652753
(Iteration 8031 / 9800) loss: 0.828653
(Iteration 8041 / 9800) loss: 0.810520
(Iteration 8051 / 9800) loss: 0.995530
(Iteration 8061 / 9800) loss: 0.784568
(Iteration 8071 / 9800) loss: 0.891749
(Iteration 8081 / 9800) loss: 0.785259
(Iteration 8091 / 9800) loss: 0.942846
(Iteration 8101 / 9800) loss: 0.753412
(Iteration 8111 / 9800) loss: 0.685459
(Iteration 8121 / 9800) loss: 0.795099
(Iteration 8131 / 9800) loss: 0.737759
(Iteration 8141 / 9800) loss: 0.627024
(Iteration 8151 / 9800) loss: 0.658368
(Iteration 8161 / 9800) loss: 0.815970
(Iteration 8171 / 9800) loss: 0.785000
(Iteration 8181 / 9800) loss: 0.689833
(Iteration 8191 / 9800) loss: 0.694218
(Iteration 8201 / 9800) loss: 0.803620
(Iteration 8211 / 9800) loss: 0.803563
```

```
(Iteration 8221 / 9800) loss: 0.728366
(Iteration 8231 / 9800) loss: 0.610136
(Iteration 8241 / 9800) loss: 0.798945
(Iteration 8251 / 9800) loss: 0.935357
(Iteration 8261 / 9800) loss: 0.757640
(Iteration 8271 / 9800) loss: 0.762017
(Iteration 8281 / 9800) loss: 0.860132
(Iteration 8291 / 9800) loss: 0.709334
(Iteration 8301 / 9800) loss: 0.640532
(Iteration 8311 / 9800) loss: 0.729611
(Iteration 8321 / 9800) loss: 0.626456
(Epoch 17 / 20) train acc: 0.741000; val_acc: 0.494000
(Iteration 8331 / 9800) loss: 0.813293
(Iteration 8341 / 9800) loss: 0.745445
(Iteration 8351 / 9800) loss: 0.817568
(Iteration 8361 / 9800) loss: 0.859755
(Iteration 8371 / 9800) loss: 0.742721
(Iteration 8381 / 9800) loss: 0.856391
(Iteration 8391 / 9800) loss: 0.708345
(Iteration 8401 / 9800) loss: 0.679482
(Iteration 8411 / 9800) loss: 0.787465
(Iteration 8421 / 9800) loss: 0.680812
(Iteration 8431 / 9800) loss: 0.674628
(Iteration 8441 / 9800) loss: 0.776797
(Iteration 8451 / 9800) loss: 0.850261
(Iteration 8461 / 9800) loss: 0.804208
(Iteration 8471 / 9800) loss: 0.632700
(Iteration 8481 / 9800) loss: 0.807253
(Iteration 8491 / 9800) loss: 0.961541
(Iteration 8501 / 9800) loss: 0.848391
(Iteration 8511 / 9800) loss: 0.795042
(Iteration 8521 / 9800) loss: 0.795588
(Iteration 8531 / 9800) loss: 0.796428
(Iteration 8541 / 9800) loss: 0.932628
(Iteration 8551 / 9800) loss: 0.865986
(Iteration 8561 / 9800) loss: 0.589160
(Iteration 8571 / 9800) loss: 0.806985
(Iteration 8581 / 9800) loss: 0.875445
(Iteration 8591 / 9800) loss: 0.866215
(Iteration 8601 / 9800) loss: 0.589695
(Iteration 8611 / 9800) loss: 0.757142
(Iteration 8621 / 9800) loss: 0.690333
(Iteration 8631 / 9800) loss: 0.584328
(Iteration 8641 / 9800) loss: 0.833069
(Iteration 8651 / 9800) loss: 0.661724
(Iteration 8661 / 9800) loss: 0.919725
(Iteration 8671 / 9800) loss: 0.752340
(Iteration 8681 / 9800) loss: 0.872869
(Iteration 8691 / 9800) loss: 0.678446
(Iteration 8701 / 9800) loss: 0.769153
(Iteration 8711 / 9800) loss: 0.761410
(Iteration 8721 / 9800) loss: 0.825859
(Iteration 8731 / 9800) loss: 0.864991
(Iteration 8741 / 9800) loss: 0.756636
(Iteration 8751 / 9800) loss: 0.647477
(Iteration 8761 / 9800) loss: 0.714159
```

```
(Iteration 8771 / 9800) loss: 0.961289
(Iteration 8781 / 9800) loss: 0.621662
(Iteration 8791 / 9800) loss: 0.607564
(Iteration 8801 / 9800) loss: 0.726925
(Iteration 8811 / 9800) loss: 0.933206
(Epoch 18 / 20) train acc: 0.749000; val_acc: 0.494000
(Iteration 8821 / 9800) loss: 0.893531
(Iteration 8831 / 9800) loss: 0.829545
(Iteration 8841 / 9800) loss: 0.899619
(Iteration 8851 / 9800) loss: 0.692633
(Iteration 8861 / 9800) loss: 0.669621
(Iteration 8871 / 9800) loss: 0.821403
(Iteration 8881 / 9800) loss: 0.778491
(Iteration 8891 / 9800) loss: 0.728066
(Iteration 8901 / 9800) loss: 0.907919
(Iteration 8911 / 9800) loss: 1.001103
(Iteration 8921 / 9800) loss: 0.764052
(Iteration 8931 / 9800) loss: 0.866806
(Iteration 8941 / 9800) loss: 0.886883
(Iteration 8951 / 9800) loss: 0.835594
(Iteration 8961 / 9800) loss: 0.822416
(Iteration 8971 / 9800) loss: 0.833234
(Iteration 8981 / 9800) loss: 0.871290
(Iteration 8991 / 9800) loss: 0.644278
(Iteration 9001 / 9800) loss: 0.688012
(Iteration 9011 / 9800) loss: 0.698948
(Iteration 9021 / 9800) loss: 0.720713
(Iteration 9031 / 9800) loss: 0.801622
(Iteration 9041 / 9800) loss: 0.762067
(Iteration 9051 / 9800) loss: 0.799787
(Iteration 9061 / 9800) loss: 0.852556
(Iteration 9071 / 9800) loss: 0.838691
(Iteration 9081 / 9800) loss: 0.762687
(Iteration 9091 / 9800) loss: 0.707170
(Iteration 9101 / 9800) loss: 0.841585
(Iteration 9111 / 9800) loss: 0.723128
(Iteration 9121 / 9800) loss: 0.641105
(Iteration 9131 / 9800) loss: 0.847532
(Iteration 9141 / 9800) loss: 0.852135
(Iteration 9151 / 9800) loss: 0.774783
(Iteration 9161 / 9800) loss: 0.491491
(Iteration 9171 / 9800) loss: 0.732769
(Iteration 9181 / 9800) loss: 0.746013
(Iteration 9191 / 9800) loss: 0.657886
(Iteration 9201 / 9800) loss: 0.733258
(Iteration 9211 / 9800) loss: 0.815558
(Iteration 9221 / 9800) loss: 0.665228
(Iteration 9231 / 9800) loss: 0.672510
(Iteration 9241 / 9800) loss: 0.795343
(Iteration 9251 / 9800) loss: 0.719686
(Iteration 9261 / 9800) loss: 0.604749
(Iteration 9271 / 9800) loss: 0.683748
(Iteration 9281 / 9800) loss: 1.042196
(Iteration 9291 / 9800) loss: 0.813385
(Iteration 9301 / 9800) loss: 0.756924
(Epoch 19 / 20) train acc: 0.750000; val_acc: 0.491000
```

```
(Iteration 9311 / 9800) loss: 0.677680
(Iteration 9321 / 9800) loss: 0.913582
(Iteration 9331 / 9800) loss: 0.725741
(Iteration 9341 / 9800) loss: 0.723819
(Iteration 9351 / 9800) loss: 0.606713
(Iteration 9361 / 9800) loss: 0.973883
(Iteration 9371 / 9800) loss: 0.677397
(Iteration 9381 / 9800) loss: 0.804836
(Iteration 9391 / 9800) loss: 0.953421
(Iteration 9401 / 9800) loss: 0.659982
(Iteration 9411 / 9800) loss: 0.594430
(Iteration 9421 / 9800) loss: 0.824651
(Iteration 9431 / 9800) loss: 0.774406
(Iteration 9441 / 9800) loss: 0.711237
(Iteration 9451 / 9800) loss: 0.747398
(Iteration 9461 / 9800) loss: 0.685345
(Iteration 9471 / 9800) loss: 0.856583
(Iteration 9481 / 9800) loss: 0.657758
(Iteration 9491 / 9800) loss: 0.701971
(Iteration 9501 / 9800) loss: 0.707321
(Iteration 9511 / 9800) loss: 0.881134
(Iteration 9521 / 9800) loss: 0.662964
(Iteration 9531 / 9800) loss: 0.623758
(Iteration 9541 / 9800) loss: 0.560932
(Iteration 9551 / 9800) loss: 0.673809
(Iteration 9561 / 9800) loss: 1.000659
(Iteration 9571 / 9800) loss: 0.683729
(Iteration 9581 / 9800) loss: 0.755238
(Iteration 9591 / 9800) loss: 0.857589
(Iteration 9601 / 9800) loss: 0.592243
(Iteration 9611 / 9800) loss: 0.832795
(Iteration 9621 / 9800) loss: 0.710457
(Iteration 9631 / 9800) loss: 0.762787
(Iteration 9641 / 9800) loss: 0.729542
(Iteration 9651 / 9800) loss: 0.640706
(Iteration 9661 / 9800) loss: 0.842940
(Iteration 9671 / 9800) loss: 0.819048
(Iteration 9681 / 9800) loss: 0.660386
(Iteration 9691 / 9800) loss: 0.676220
(Iteration 9701 / 9800) loss: 0.810101
(Iteration 9711 / 9800) loss: 0.768903
(Iteration 9721 / 9800) loss: 0.878861
(Iteration 9731 / 9800) loss: 0.950525
(Iteration 9741 / 9800) loss: 0.622398
(Iteration 9751 / 9800) loss: 0.744435
(Iteration 9761 / 9800) loss: 0.797743
(Iteration 9771 / 9800) loss: 0.490029
(Iteration 9781 / 9800) loss: 0.603168
(Iteration 9791 / 9800) loss: 0.654702
(Epoch 20 / 20) train acc: 0.741000; val_acc: 0.478000
```

Test Your Model!

Run your best model on the validation and test sets. You should achieve at least 50% accuracy on the validation set and the test set.

```
Validation set accuracy:  0.519
Test set accuracy:  0.488
```