linear_classifier

April 18, 2024

1 ITE4052 Assignment 3-1: Linear Classifiers

- This material draws from EECS 498-007/598-005 (Justin Johnson)

Before we start, please put your name and HYID in following format Firstname LASTNAME, #00000000 // e.g.) Sukmin Yun, #12345678

Your Answer:
Junwoo Park, #2021006253

1.1 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment, same as Assignment 2. You'll need to rerun this setup code each time you start the notebook.

First, run this cell load the autoreload extension. This allows us to edit .py source files, and re-import them into the notebook for a seamless editing and debugging experience.

[1]:
```
%load_ext autoreload
%autoreload 2
```

1.1.1 Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Now recall the path in your Google Drive where you uploaded this notebook, fill it in below. If everything is working correctly then running the folowing cell should print the filenames from the assignment:

```
['two_layer_net.ipynb', 'ite4052', 'two_layer_net.py', 'linear_classifier.py', 'linear_classifi
```

[3]:
```
import os

# TODO: Fill in the Google Drive path where you uploaded the assignment
# Example: If you create a ITE4052 folder and put all the files under A3␣
 ↪folder, then 'ITE4052/A3'
# GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'ITE4052/A3'
GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'ITE4052/A3'
GOOGLE_DRIVE_PATH = os.path.join('drive', 'My Drive',␣
 ↪GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
print(os.listdir(GOOGLE_DRIVE_PATH))
```

```
['collectSubmission.sh', 'makepdf.py', 'collect_submission.ipynb', 'ite4052',
'__pycache__', 'svm_best_model.pt', 'softmax_best_model.pt', 'nn_best_model.pt',
'two_layer_net.ipynb', 'two_layer_net.py', 'linear_classifier.py',
'linear_classifier.ipynb']
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run th following cell to allow us to import from the `.py` files of this assignment. If it works correctly, it should print the message:

```
Hello from linear_classifier.py!
```

as well as the last edit time for the file `linear_classifier.py`.

[4]:
```
import sys
sys.path.append(GOOGLE_DRIVE_PATH)

import time, os
os.environ["TZ"] = "US/Eastern"
time.tzset()

from linear_classifier import hello_linear_classifier
hello_linear_classifier()

linear_classifier_path = os.path.join(GOOGLE_DRIVE_PATH, 'linear_classifier.py')
linear_classifier_edit_time = time.ctime(os.path.
 ↪getmtime(linear_classifier_path))
print('linear_classifier.py last edited on %s' % linear_classifier_edit_time)
```

```
Hello from linear_classifier.py!
linear_classifier.py last edited on Thu Apr 18 00:16:28 2024
```

2 Data preprocessing

2.1 Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

[5]:
```
import ite4052
import torch
import torchvision
import matplotlib.pyplot as plt
import statistics
import random
import time
import math
%matplotlib inline


plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['font.size'] = 16
```

Starting in this assignment, we will use the GPU to accelerate our computation. Run this cell to make sure you are using a GPU.

[6]:
```
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

2.2 Load the CIFAR-10 dataset

Then, we will first load the CIFAR-10 dataset, same as knn. The utility function `ite4052.data.preprocess_cifar10()` returns the entire CIFAR-10 dataset as a set of six **Torch tensors**:

- `X_train` contains all training images (real numbers in the range $[0, 1]$)
- `y_train` contains all training labels (integers in the range $[0, 9]$)
- `X_val` contains all validation images
- `y_val` contains all validation labels
- `X_test` contains all test images
- `y_test` contains all test labels

In this notebook we will use the **bias trick**: By adding an extra constant feature of ones to each image, we avoid the need to keep track of a bias vector; the bias will be encoded as the part of the weight matrix that interacts with the constant ones in the input.

In the `two_layer_net.ipynb` notebook that follows this one, we will not use the bias trick.

We can learn more about the `ite4052.data.preprocess_cifar10` function by invoking the `help` command:

[7]:
```
import ite4052
help(ite4052.data.preprocess_cifar10)
```

```
Help on function preprocess_cifar10 in module ite4052.data:
```

```
preprocess_cifar10(cuda=True, show_examples=True, bias_trick=False,
flatten=True, validation_ratio=0.2, dtype=torch.float32)
    Returns a preprocessed version of the CIFAR10 dataset, automatically
    downloading if necessary. We perform the following steps:

    (0) [Optional] Visualize some images from the dataset
    (1) Normalize the data by subtracting the mean
    (2) Reshape each image of shape (3, 32, 32) into a vector of shape (3072,)
    (3) [Optional] Bias trick: add an extra dimension of ones to the data
    (4) Carve out a validation set from the training set

    Inputs:
    - cuda: If true, move the entire dataset to the GPU
    - validation_ratio: Float in the range (0, 1) giving the fraction of the
train
      set to reserve for validation
    - bias_trick: Boolean telling whether or not to apply the bias trick
    - show_examples: Boolean telling whether or not to visualize data samples
    - dtype: Optional, data type of the input image X

    Returns a dictionary with the following keys:
    - 'X_train': `dtype` tensor of shape (N_train, D) giving training images
    - 'X_val': `dtype` tensor of shape (N_val, D) giving val images
    - 'X_test': `dtype` tensor of shape (N_test, D) giving test images
    - 'y_train': int64 tensor of shape (N_train,) giving training labels
    - 'y_val': int64 tensor of shape (N_val,) giving val labels
    - 'y_test': int64 tensor of shape (N_test,) giving test labels

    N_train, N_val, and N_test are the number of examples in the train, val, and
    test sets respectively. The precise values of N_train and N_val are
determined
    by the input parameter validation_ratio. D is the dimension of the image
data;
    if bias_trick is False, then D = 32 * 32 * 3 = 3072;
    if bias_trick is True then D = 1 + 32 * 32 * 3 = 3073.
```

We can now run the `ite4052.data.preprocess` function to get our data:

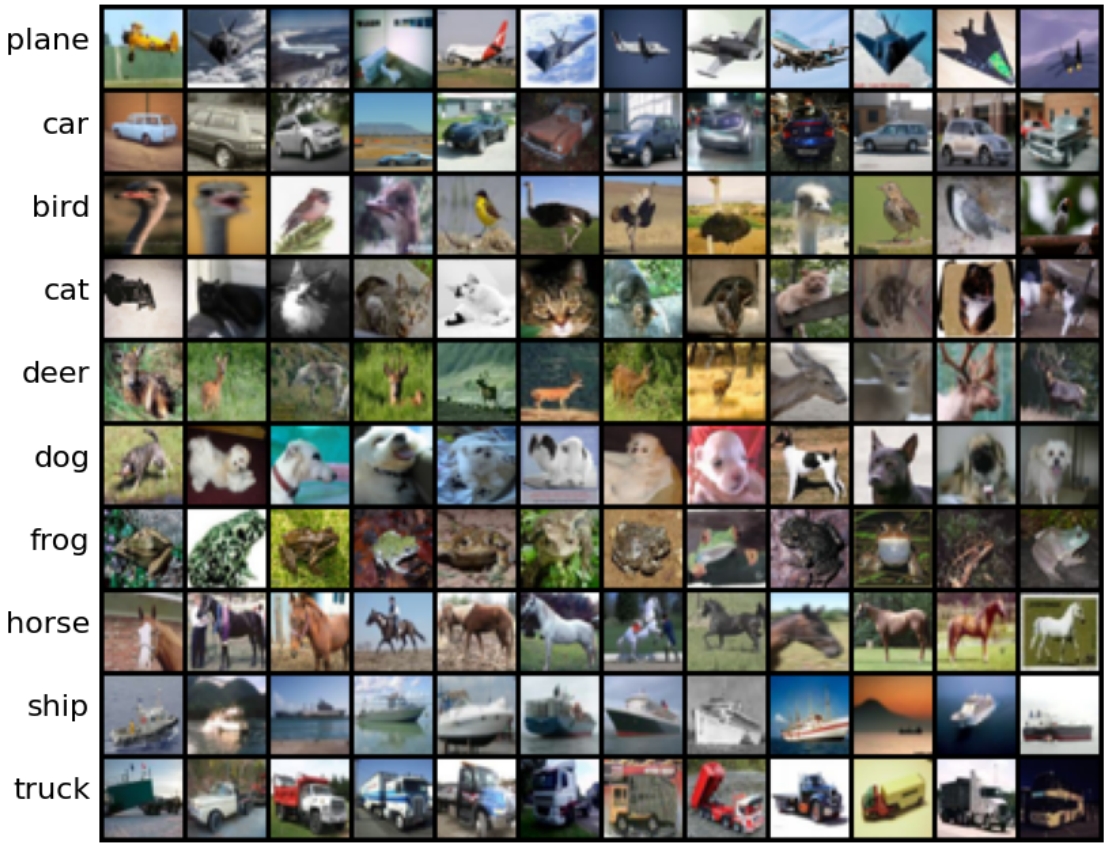
[8]:
```
# Invoke the above function to get our data.
import ite4052

ite4052.reset_seed(0)
data_dict = ite4052.data.preprocess_cifar10(bias_trick=True, cuda=True,␣
 ↪dtype=torch.float64)
print('Train data shape: ', data_dict['X_train'].shape)
print('Train labels shape: ', data_dict['y_train'].shape)
print('Validation data shape: ', data_dict['X_val'].shape)
```

```
print('Validation labels shape: ', data_dict['y_val'].shape)
print('Test data shape: ', data_dict['X_test'].shape)
print('Test labels shape: ', data_dict['y_test'].shape)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./cifar-10-python.tar.gz
```

```
100%|      | 170498071/170498071 [00:04<00:00, 35953285.77it/s]
```

```
Extracting ./cifar-10-python.tar.gz to .
```




```
Train data shape:  torch.Size([40000, 3073])
Train labels shape:  torch.Size([40000])
Validation data shape:  torch.Size([10000, 3073])
Validation labels shape:  torch.Size([10000])
Test data shape:  torch.Size([10000, 3073])
Test labels shape:  torch.Size([10000])
```

3 SVM Classifier

In this section, you will:

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

In Assignment 3, you SHOULD NOT use ".to()" or ".cuda()" in each implementation block. Otherwise, your implementation would gives you an error in Autograder end.

First, we will test the naive version of svm loss in `linear_classifier.py`. Let's first try the naive implementation of the loss we provided for you. You will get 9.000888. (Note: we've provided the `loss` part of the `svm_loss_naive` function, so you don't need to re-implement in `svm_loss_naive`. However, if your loss value doesn't match, then please report this to LMS)

[9]:
```
import ite4052
from linear_classifier import svm_loss_naive

ite4052.reset_seed(0)
# generate a random SVM weight tensor of small numbers
W = torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device) * 0.0001

loss, _grad_ = svm_loss_naive(W, data_dict['X_val'], data_dict['y_val'], 0.
 ↪000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.000888
```

The `_grad_` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`, by filing out the TODO blocks. You will find it helpful to interweave your new code inside the existing function.

To check that you have implemented the gradient correctly, we will use **numeric gradient checking**: we will use a finite differences approach to numerically estimate the gradient of the forward pass, and compare this numeric gradient to the analytic gradient that you implemented.

We have provided a function `ite4052.grad.grad_check_sparse` to help with numeric gradient checking. You can learn more about this function using the `help` command:

[10]:
```
import ite4052
help(ite4052.grad.grad_check_sparse)
```

```
Help on function grad_check_sparse in module ite4052.grad:

grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-07)
    Utility function to perform numeric gradient checking. We use the centered
    difference formula to compute a numeric derivative:
```

```
    f'(x) =~ (f(x + h) - f(x - h)) / (2h)

    Rather than computing a full numeric gradient, we sparsely sample a few
    dimensions along which to compute numeric derivatives.

    Inputs:
    - f: A function that inputs a torch tensor and returns a torch scalar
    - x: A torch tensor of the point at which to evaluate the numeric gradient
    - analytic_grad: A torch tensor giving the analytic gradient of f at x
    - num_checks: The number of dimensions along which to check
    - h: Step size for computing numeric derivatives
```

Now run the following to perform numeric gradient checking on the gradients of your SVM loss. You should see relative errors less than `1e-5`.

[13]:
```
import ite4052
from linear_classifier import svm_loss_naive

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

# Use a random W and a minibatch of data from the val set for gradient checking
# For numeric gradient checking it is a good idea to use 64-bit floating point
# numbers for increased numeric precision; however when actually training models
# we usually use 32-bit floating point numbers for increased speed.
ite4052.reset_seed(0)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)
batch_size = 64
X_batch = data_dict['X_val'][:batch_size]
y_batch = data_dict['y_val'][:batch_size]

# Compute the loss and its gradient at W.
# YOUR_TURN: implement the gradient part of 'svm_loss_naive' function in␣
 ↪"linear_classifier.py"
_, grad = svm_loss_naive(W, X_batch, y_batch, reg=0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should
# match almost exactly along all dimensions.
f = lambda w: svm_loss_naive(w, X_batch, y_batch, reg=0.0)[0]
grad_numerical = ite4052.grad.grad_check_sparse(f, W, grad)
```

```
numerical: 0.031599 analytic: -2.235970, relative error: 1.000000e+00
numerical: 0.111444 analytic: 0.547816, relative error: 6.619109e-01
numerical: 0.011204 analytic: -0.420664, relative error: 1.000000e+00
numerical: -0.046128 analytic: -0.713531, relative error: 8.785571e-01
```

```
numerical: 0.071948 analytic: -0.772067, relative error: 1.000000e+00
numerical: 0.025688 analytic: 0.135445, relative error: 6.811555e-01
numerical: 0.185388 analytic: 5.257542, relative error: 9.318792e-01
numerical: -0.021740 analytic: 2.436267, relative error: 1.000000e+00
numerical: -0.159613 analytic: -1.581438, relative error: 8.166476e-01
numerical: 0.092690 analytic: -2.905011, relative error: 1.000000e+00
```

Let's do the gradient check once again with regularization turned on. (You didn't forget the regularization gradient, did you?)

You should see relative errors less than `1e-5`.

[14]:
```
import ite4052
from linear_classifier import svm_loss_naive

# Use a minibatch of data from the val set for gradient checking
ite4052.reset_seed(0)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)
batch_size = 64
X_batch = data_dict['X_val'][:batch_size]
y_batch = data_dict['y_val'][:batch_size]

# Compute the loss and its gradient at W.
# YOUR_TURN: check your 'svm_loss_naive' implementation with different 'reg'
_, grad = svm_loss_naive(W, X_batch, y_batch, reg=1e3)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should
# match almost exactly along all dimensions.
f = lambda w: svm_loss_naive(w, X_batch, y_batch, reg=1e3)[0]
grad_numerical = ite4052.grad.grad_check_sparse(f, W, grad)
```

```
numerical: 0.124849 analytic: -2.142721, relative error: 1.000000e+00
numerical: 0.168915 analytic: 0.605286, relative error: 5.636413e-01
numerical: 0.148752 analytic: -0.283115, relative error: 1.000000e+00
numerical: -0.024936 analytic: -0.692340, relative error: 9.304696e-01
numerical: -0.008570 analytic: -0.852586, relative error: 9.800954e-01
numerical: -0.103155 analytic: 0.006602, relative error: 1.000000e+00
numerical: -0.335573 analytic: 4.736580, relative error: 1.000000e+00
numerical: -0.222176 analytic: 2.235831, relative error: 1.000000e+00
numerical: 0.681163 analytic: -0.740662, relative error: 1.000000e+00
numerical: -0.004090 analytic: -3.001791, relative error: 9.972790e-01
```

Now, let's implement vectorized version of SVM: `svm_loss_vectorized`. It should compute the same inputs and outputs as the naive version before, but it should involve **no explicit loops**.

Let's first check the speed and performance bewteen the non-vectorized and the vectorized version. You should see a 15-120x speedup. PyTorch does some extra setup the first time you run CUDA code, so **you may need to run this cell more than once to see the desired speedup**.

(Note: It may have some difference, but should be less than 1e-6)

[19]:
```
import ite4052
from linear_classifier import svm_loss_naive, svm_loss_vectorized

# Use random weights and a minibatch of val data for gradient checking
ite4052.reset_seed(0)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)
X_batch = data_dict['X_val'][:128]
y_batch = data_dict['y_val'][:128]
reg = 0.000005

# Run and time the naive version
torch.cuda.synchronize()
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_naive = 1000.0 * (toc - tic)
print('Naive loss: %e computed in %.2fms' % (loss_naive, ms_naive))

# Run and time the vectorized version
torch.cuda.synchronize()
tic = time.time()
# YOUR_TURN: implement the loss part of 'svm_loss_vectorized' function in␣
 ↪"linear_classifier.py"
loss_vec, _ = svm_loss_vectorized(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_vec = 1000.0 * (toc - tic)
print('Vectorized loss: %e computed in %.2fms' % (loss_vec, ms_vec))

# The losses should match but your vectorized implementation should be much␣
 ↪faster.
print('Difference: %.2e' % (loss_naive - loss_vec))
print('Speedup: %.2fX' % (ms_naive / ms_vec))
```

```
Naive loss: 9.002394e+00 computed in 307.53ms
Vectorized loss: 9.002394e+00 computed in 2.60ms
Difference: -5.33e-15
Speedup: 118.18X
```

Then, let's compute the gradient of the loss function. We can check the difference of gradient as well. (The error should be less than 1e-6)

Now implement a vectorized version of the gradient computation in `svm_loss_vectorize` above. Run the cell below to compare the gradient of your naive and vectorized implementations. The difference between the gradients should be less than `1e-6`, and the vectorized version should run

15-120x faster.

[28]:
```
# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.

import ite4052
from linear_classifier import svm_loss_naive, svm_loss_vectorized

# Use random weights and a minibatch of val data for gradient checking
ite4052.reset_seed(0)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)
X_batch = data_dict['X_val'][:128]
y_batch = data_dict['y_val'][:128]
reg = 0.000005

# Run and time the naive version
torch.cuda.synchronize()
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_batch, y_batch, 0.000005)
torch.cuda.synchronize()
toc = time.time()
ms_naive = 1000.0 * (toc - tic)
print('Naive loss and gradient: computed in %.2fms' % ms_naive)

# Run and time the vectorized version
torch.cuda.synchronize()
tic = time.time()
# YOUR_TURN: implement the gradient part of 'svm_loss_vectorized' function in␣
 ↪"linear_classifier.py"
_, grad_vec = svm_loss_vectorized(W, X_batch, y_batch, 0.000005)
torch.cuda.synchronize()
toc = time.time()
ms_vec = 1000.0 * (toc - tic)
print('Vectorized loss and gradient: computed in %.2fms' % ms_vec)

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a tensor, so
# we use the Frobenius norm to compare them.
grad_difference = torch.norm(grad_naive - grad_vec, p='fro')
print('Gradient difference: %.2e' % grad_difference)
print('Speedup: %.2fX' % (ms_naive / ms_vec))
```

```
Naive loss and gradient: computed in 464.59ms
Vectorized loss and gradient: computed in 3.53ms
Gradient difference: 6.46e+02
Speedup: 131.70X
```

Now that we have an efficient vectorized implementation of the SVM loss and its gradient, we can implement a training pipeline for linear classifiers.

Please complete the implementation of `train_linear_classifier` in `linear_classifer.py`.

Once you have implemented the training function, run the following cell to train a linear classifier using some default hyperparameters:

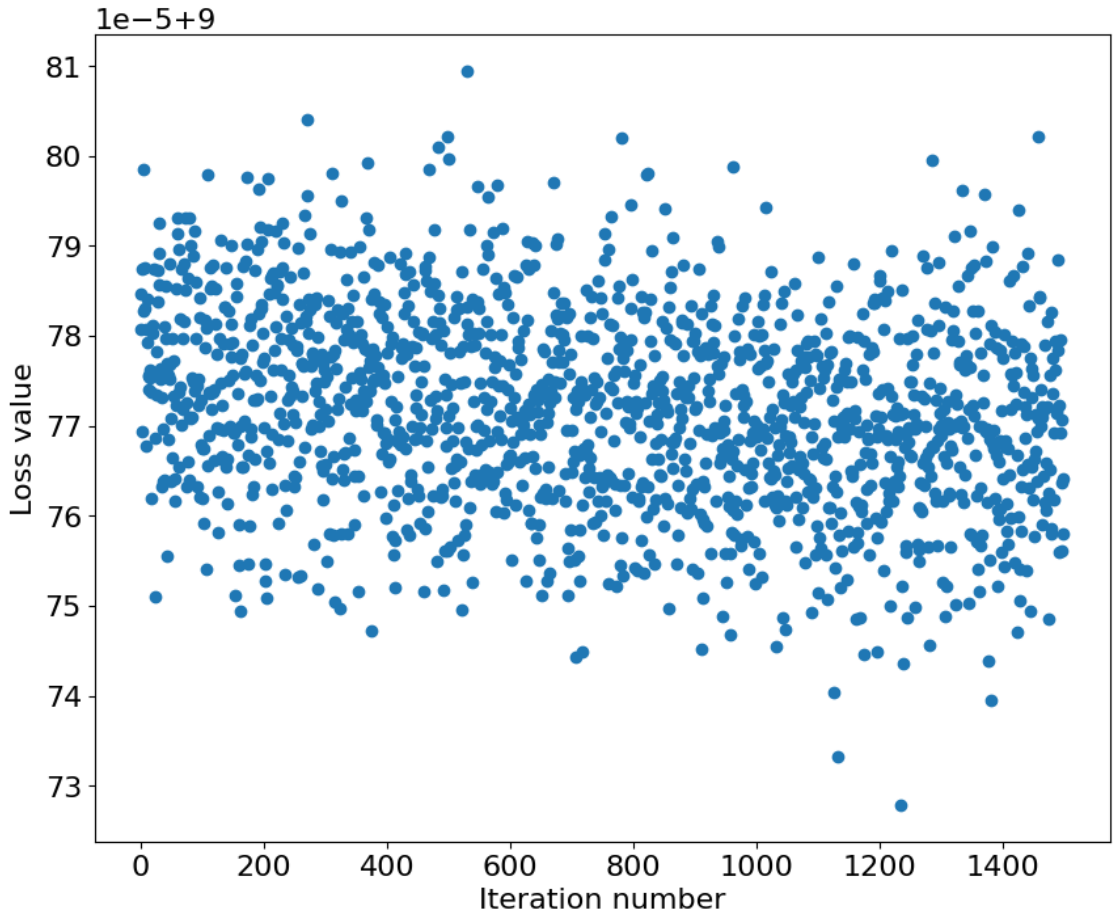
(You should see a final loss close to 9.0, and your training loop should run in about two seconds)

[29]:
```
import ite4052
from linear_classifier import svm_loss_vectorized, train_linear_classifier

# fix random seed before we perform this operation
ite4052.reset_seed(0)

torch.cuda.synchronize()
tic = time.time()

# YOUR_TURN: Implement how to construct the batch,
#            and how to update the weight in 'train_linear_classifier'
W, loss_hist = train_linear_classifier(svm_loss_vectorized, None,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=3e-11, reg=2.5e4,
                                       num_iters=1500, verbose=True)

torch.cuda.synchronize()
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 9.000785
iteration 100 / 1500: loss 9.000762
iteration 200 / 1500: loss 9.000777
iteration 300 / 1500: loss 9.000766
iteration 400 / 1500: loss 9.000778
iteration 500 / 1500: loss 9.000771
iteration 600 / 1500: loss 9.000772
iteration 700 / 1500: loss 9.000770
iteration 800 / 1500: loss 9.000772
iteration 900 / 1500: loss 9.000772
iteration 1000 / 1500: loss 9.000770
iteration 1100 / 1500: loss 9.000789
iteration 1200 / 1500: loss 9.000787
iteration 1300 / 1500: loss 9.000769
iteration 1400 / 1500: loss 9.000778
That took 2.739383s
```

A useful debugging strategy is to plot the loss as a function of iteration number. In this case it seems our hyperparameters are not good, since the training loss is not decreasing very fast.

[30]:
```
plt.plot(loss_hist, 'o')
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



Then, let's move on to the prediction stage. We can evaluate the performance our trained model on both the training and validation set. You should see validation accuracy less than 20%.

[32]:
```
import ite4052
from linear_classifier import predict_linear_classifier

# fix random seed before we perform this operation
ite4052.reset_seed(0)

# evaluate the performance on both the training and validation set
# YOUR_TURN: Implement how to make a prediction with the trained weight
#            in 'predict_linear_classifier'
y_train_pred = predict_linear_classifier(W, data_dict['X_train'])
```

```
train_acc = 100.0 * (data_dict['y_train'] == y_train_pred).double().mean().
 ↪item()
print('Training accuracy: %.2f%%' % train_acc)

y_val_pred = predict_linear_classifier(W, data_dict['X_val'])
val_acc = 100.0 * (data_dict['y_val'] == y_val_pred).double().mean().item()
print('Validation accuracy: %.2f%%' % val_acc)
```

```
Training accuracy: 9.24%
Validation accuracy: 9.00%
```

Unfortunately, the performance of our initial model is quite bad. To find a better hyperparamters, we first modulized the functions that we've implemented as LinearSVM.

Now, please use the validation set to tune hyperparameters (regularization strength and learning rate). You should experiment with different ranges for the learning rates and regularization strengths.

To get full credit for the assignment your best model found through cross-validation should achieve an accuracy of at least 37% on the validation set.

(Our best model got over 38.1% – did you beat us?)

[33]:
```
import os
import ite4052
from linear_classifier import LinearSVM, svm_get_search_params,␣
 ↪test_one_param_set

# YOUR_TURN: find the best learning_rates and regularization_strengths␣
 ↪combination
#            in 'svm_get_search_params'
learning_rates, regularization_strengths = svm_get_search_params()
num_models = len(learning_rates) * len(regularization_strengths)

####
# It is okay to comment out the following conditions when you are working on␣
 ↪svm_get_search_params.
# But, please do not forget to reset back to the original setting once you are␣
 ↪done.
if num_models > 25:
  raise Exception("Please do not test/submit more than 25 items at once")
elif num_models < 5:
  raise Exception("Please present at least 5 parameter sets in your final␣
 ↪ipynb")
####


i = 0
# results is dictionary mapping tuples of the form
```

```
# (learning_rate, regularization_strength) to tuples of the form
# (train_acc, val_acc).
results = {}
best_val = -1.0   # The highest validation accuracy that we have seen so far.
best_svm_model = None # The LinearSVM object that achieved the highest␣
 ↪validation rate.
num_iters = 2000 # number of iterations

for lr in learning_rates:
  for reg in regularization_strengths:
    i += 1
    print('Training SVM %d / %d with learning_rate=%e and reg=%e'
          % (i, num_models, lr, reg))

    ite4052.reset_seed(0)
    # YOUR_TURN: implement a function that gives the trained model with
    #            train/validation accuracies in 'test_one_param_set'
    #            (note: this function will be used in Softmax Classifier␣
 ↪section as well)
    cand_svm_model, cand_train_acc, cand_val_acc =␣
 ↪test_one_param_set(LinearSVM(), data_dict, lr, reg, num_iters)

    if cand_val_acc > best_val:
      best_val = cand_val_acc
      best_svm_model = cand_svm_model # save the svm
    results[(lr, reg)] = (cand_train_acc, cand_val_acc)


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

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

# save the best model
path = os.path.join(GOOGLE_DRIVE_PATH, 'svm_best_model.pt')
best_svm_model.save(path)
```

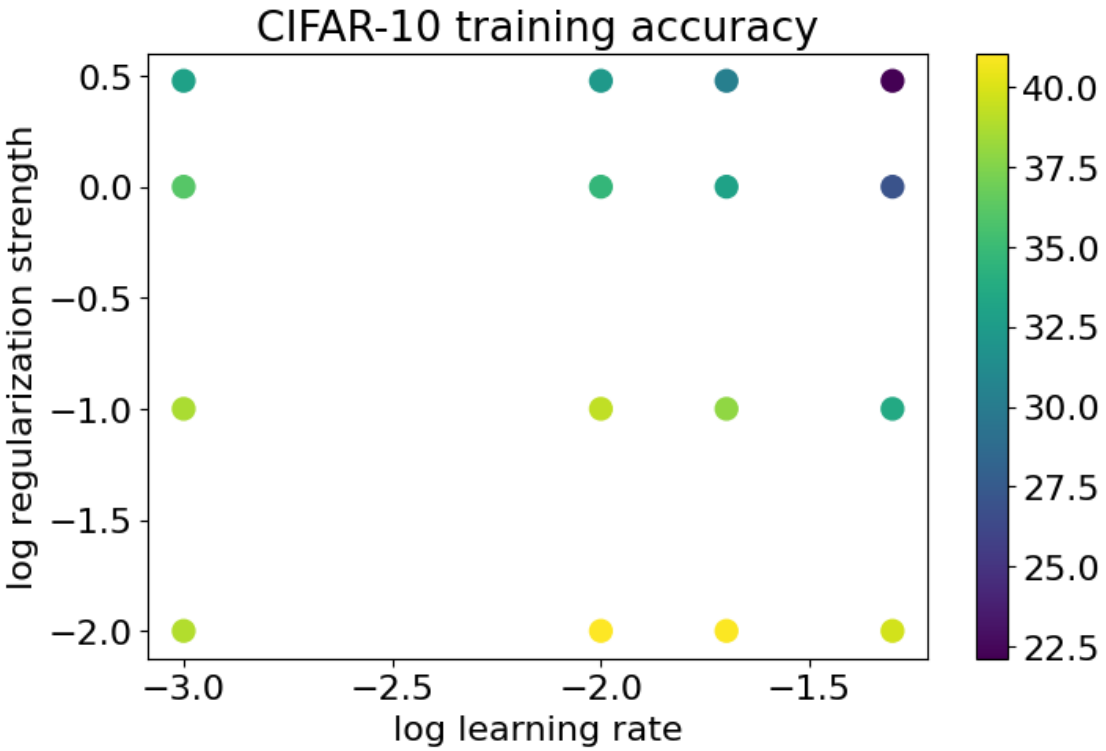
```
Training SVM 1 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-02
Training SVM 2 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-02
Training SVM 3 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-01
Training SVM 4 / 25 with learning_rate=1.000000e-03 and reg=1.000000e+00
Training SVM 5 / 25 with learning_rate=1.000000e-03 and reg=3.000000e+00
Training SVM 6 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-02
```

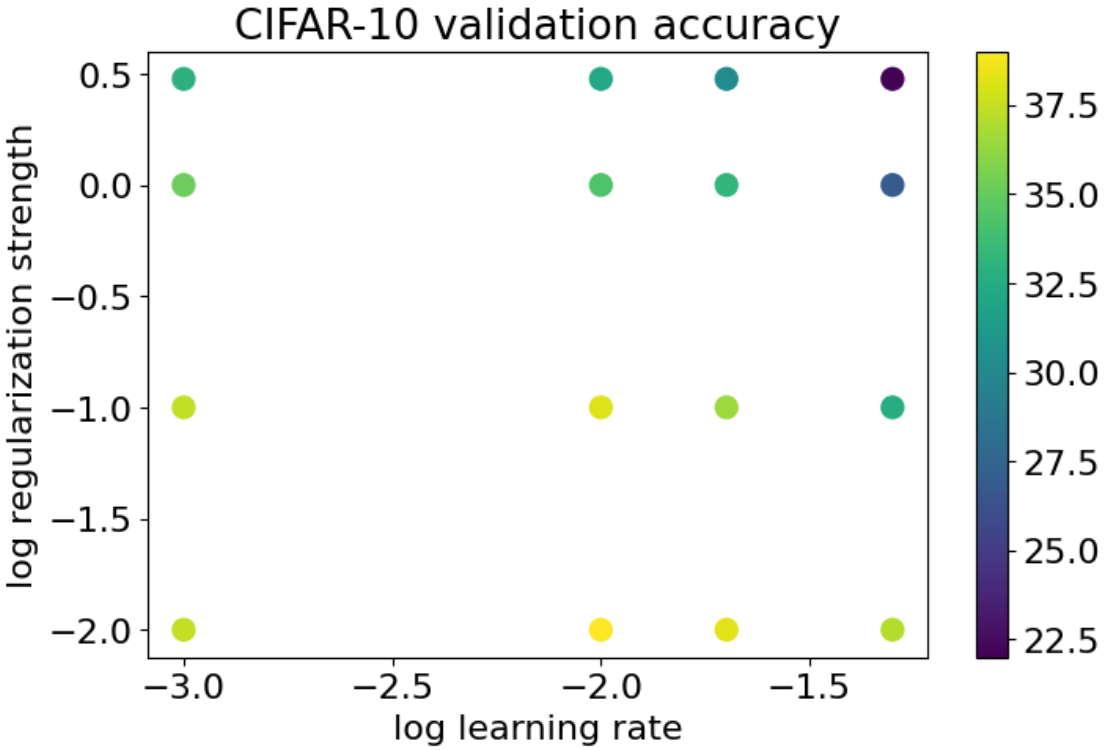
```
Training SVM 7 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-02
Training SVM 8 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-01
Training SVM 9 / 25 with learning_rate=1.000000e-03 and reg=1.000000e+00
Training SVM 10 / 25 with learning_rate=1.000000e-03 and reg=3.000000e+00
Training SVM 11 / 25 with learning_rate=2.000000e-02 and reg=1.000000e-02
Training SVM 12 / 25 with learning_rate=2.000000e-02 and reg=1.000000e-02
Training SVM 13 / 25 with learning_rate=2.000000e-02 and reg=1.000000e-01
Training SVM 14 / 25 with learning_rate=2.000000e-02 and reg=1.000000e+00
Training SVM 15 / 25 with learning_rate=2.000000e-02 and reg=3.000000e+00
Training SVM 16 / 25 with learning_rate=5.000000e-02 and reg=1.000000e-02
Training SVM 17 / 25 with learning_rate=5.000000e-02 and reg=1.000000e-02
Training SVM 18 / 25 with learning_rate=5.000000e-02 and reg=1.000000e-01
Training SVM 19 / 25 with learning_rate=5.000000e-02 and reg=1.000000e+00
Training SVM 20 / 25 with learning_rate=5.000000e-02 and reg=3.000000e+00
Training SVM 21 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-02
Training SVM 22 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-02
Training SVM 23 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-01
Training SVM 24 / 25 with learning_rate=1.000000e-02 and reg=1.000000e+00
Training SVM 25 / 25 with learning_rate=1.000000e-02 and reg=3.000000e+00
lr 1.000000e-03 reg 1.000000e-02 train accuracy: 38.850000 val accuracy:
37.480000
lr 1.000000e-03 reg 1.000000e-01 train accuracy: 38.687500 val accuracy:
37.350000
lr 1.000000e-03 reg 1.000000e+00 train accuracy: 36.050000 val accuracy:
35.210000
lr 1.000000e-03 reg 3.000000e+00 train accuracy: 32.957500 val accuracy:
32.850000
lr 1.000000e-02 reg 1.000000e-02 train accuracy: 41.060000 val accuracy:
39.010000
lr 1.000000e-02 reg 1.000000e-01 train accuracy: 39.242500 val accuracy:
38.080000
lr 1.000000e-02 reg 1.000000e+00 train accuracy: 34.682500 val accuracy:
34.270000
lr 1.000000e-02 reg 3.000000e+00 train accuracy: 32.300000 val accuracy:
32.180000
lr 2.000000e-02 reg 1.000000e-02 train accuracy: 40.932500 val accuracy:
38.220000
lr 2.000000e-02 reg 1.000000e-01 train accuracy: 37.920000 val accuracy:
36.470000
lr 2.000000e-02 reg 1.000000e+00 train accuracy: 33.047500 val accuracy:
33.100000
lr 2.000000e-02 reg 3.000000e+00 train accuracy: 30.205000 val accuracy:
30.140000
lr 5.000000e-02 reg 1.000000e-02 train accuracy: 39.795000 val accuracy:
37.050000
lr 5.000000e-02 reg 1.000000e-01 train accuracy: 33.570000 val accuracy:
32.620000
lr 5.000000e-02 reg 1.000000e+00 train accuracy: 26.972500 val accuracy:
```

```
26.850000
lr 5.000000e-02 reg 3.000000e+00 train accuracy: 22.122500 val accuracy:
21.990000
best validation accuracy achieved during cross-validation: 39.010000
Saved in drive/My Drive/ITE4052/A3/svm_best_model.pt
```

Visualize the cross-validation results. You can use this as a debugging tool – after examining the cross-validation results here, you may want to go back and rerun your cross-validation from above.

[34]:
```python
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap='viridis')
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')
plt.gcf().set_size_inches(8, 5)
plt.show()

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap='viridis')
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.gcf().set_size_inches(8, 5)
plt.show()
```


CIFAR-10 training accuracy
log regularization strength
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3.0
−2.5
−2.0
−1.5
log learning rate
40.0
37.5
35.0
32.5
30.0
27.5
25.0
22.5


CIFAR-10 validation accuracy
log regularization strength
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3.0
−2.5
−2.0
−1.5
log learning rate
37.5
35.0
32.5
30.0
27.5
25.0
22.5

Evaluate the best svm on test set. To get full credit for the assignment you should achieve a test-set accuracy above 35%.

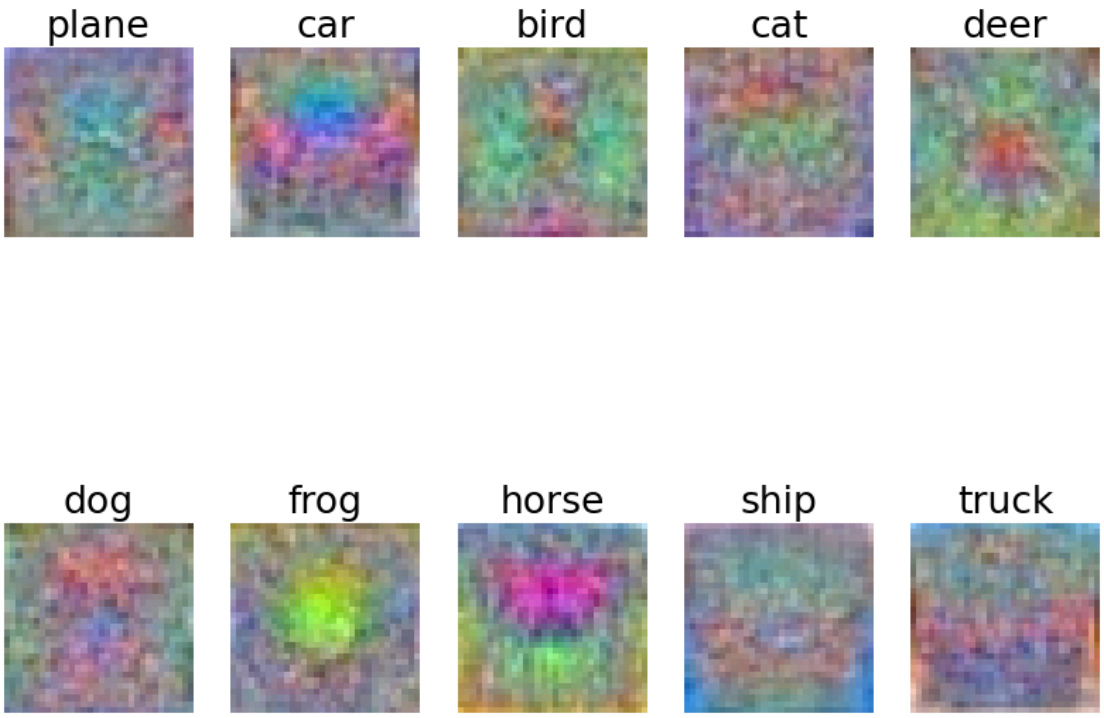
(Our best was over 39.1% – did you beat us?)

[35]:
```
import ite4052

ite4052.reset_seed(0)
y_test_pred = best_svm_model.predict(data_dict['X_test'])
test_accuracy = torch.mean((data_dict['y_test'] == y_test_pred).double())
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.388900
```

Visualize the learned weights for each class. Depending on your choice of learning rate and regularization strength, these may or may not be nice to look at.

[36]:
```
w = best_svm_model.W[:-1,:] # strip out the bias
w = w.reshape(3, 32, 32, 10)
w = w.transpose(0, 2).transpose(1, 0)

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

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



4 Softmax Classifier

Similar to the SVM, you will:

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

As noted in the SVM section, you SHOULD NOT use ".to()" or ".cuda()" in each implementation block.

First, let's start from implementing the naive softmax loss function with nested loops in `softmax_loss_naive` function.

As a sanity check to see whether we have implemented the loss correctly, run the softmax classifier with a small random weight matrix and no regularization. You should see loss near $\log(10) = 2.3$

[37]:
```
import ite4052
from linear_classifier import softmax_loss_naive

ite4052.reset_seed(0)
# Generate a random softmax weight tensor and use it to compute the loss.
```

```
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)

X_batch = data_dict['X_val'][:128]
y_batch = data_dict['y_val'][:128]

# YOUR_TURN: Complete the implementation of softmax_loss_naive and implement
# a (naive) version of the gradient that uses nested loops.
loss, _ = softmax_loss_naive(W, X_batch, y_batch, reg=0.0)

# As a rough sanity check, our loss should be something close to log(10.0).
print('loss: %f' % loss)
print('sanity check: %f' % (math.log(10.0)))
```

```
loss: 2.302826
sanity check: 2.302585
```

Next, we use gradient checking to debug the analytic gradient of our naive softmax loss function. If you've implemented the gradient correctly, you should see relative errors less than `1e-5`.

[38]:
```
import ite4052
from linear_classifier import softmax_loss_naive

ite4052.reset_seed(0)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)

X_batch = data_dict['X_val'][:128]
y_batch = data_dict['y_val'][:128]

# YOUR_TURN: Complete the implementation of softmax_loss_naive and implement
# a (naive) version of the gradient that uses nested loops.
_, grad = softmax_loss_naive(W, X_batch, y_batch, reg=0.0)

f = lambda w: softmax_loss_naive(w, X_batch, y_batch, reg=0.0)[0]
ite4052.grad.grad_check_sparse(f, W, grad, 10)
```

```
numerical: 0.003046 analytic: 0.003046, relative error: 6.468497e-07
numerical: 0.006308 analytic: 0.006308, relative error: 1.234992e-07
numerical: 0.005392 analytic: 0.005392, relative error: 4.593635e-07
numerical: 0.002581 analytic: 0.002581, relative error: 3.442300e-08
numerical: 0.007512 analytic: 0.007512, relative error: 8.122736e-07
numerical: 0.006417 analytic: 0.006417, relative error: 2.286038e-08
numerical: 0.011391 analytic: 0.011391, relative error: 1.960823e-07
numerical: 0.001822 analytic: 0.001822, relative error: 2.932218e-06
numerical: -0.014710 analytic: -0.014710, relative error: 8.967622e-08
numerical: -0.005153 analytic: -0.005153, relative error: 4.012889e-07
```

Let's perform another gradient check with regularization enabled. Again you should see relative errors less than `1e-5`.

[39]:
```
import ite4052
from linear_classifier import softmax_loss_naive

ite4052.reset_seed(128)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)
reg = 10.0

X_batch = data_dict['X_val'][:128]
y_batch = data_dict['y_val'][:128]

# YOUR_TURN: Complete the gradient compuation part of softmax_loss_naive
_, grad = softmax_loss_naive(W, X_batch, y_batch, reg)

f = lambda w: softmax_loss_naive(w, X_batch, y_batch, reg)[0]
ite4052.grad.grad_check_sparse(f, W, grad, 10)
```

```
numerical: 0.004914 analytic: 0.004914, relative error: 2.907815e-08
numerical: 0.005887 analytic: 0.005887, relative error: 8.060044e-07
numerical: 0.006309 analytic: 0.006309, relative error: 1.643123e-07
numerical: 0.001580 analytic: 0.001580, relative error: 1.790195e-06
numerical: 0.005839 analytic: 0.005839, relative error: 6.556458e-07
numerical: 0.006800 analytic: 0.006800, relative error: 5.004632e-07
numerical: 0.011465 analytic: 0.011465, relative error: 2.593154e-07
numerical: 0.002314 analytic: 0.002314, relative error: 4.575594e-07
numerical: -0.016813 analytic: -0.016813, relative error: 9.876027e-08
numerical: -0.006673 analytic: -0.006673, relative error: 1.337888e-08
```

Then, let's move on to the vectorized form: `softmax_loss_vectorized`.

Now that we have a naive implementation of the softmax loss function and its gradient, implement a vectorized version in softmax_loss_vectorized. The two versions should compute the same results, but the vectorized version should be much faster.

The differences between the naive and vectorized losses and gradients should both be less than `1e-6`, and your vectorized implementation should be at least 20x faster than the naive implementation.

[40]:
```
import ite4052
from linear_classifier import softmax_loss_naive, softmax_loss_vectorized

ite4052.reset_seed(0)
W = 0.0001 * torch.randn(3073, 10, dtype=data_dict['X_val'].dtype,␣
 ↪device=data_dict['X_val'].device)
reg = 0.05

X_batch = data_dict['X_val'][:128]
```

```
y_batch = data_dict['y_val'][:128]

# Run and time the naive version
torch.cuda.synchronize()
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_naive = 1000.0 * (toc - tic)
print('naive loss: %e computed in %fs' % (loss_naive, ms_naive))

# Run and time the vectorized version
# YOUR_TURN: Complete the implementation of softmax_loss_vectorized
torch.cuda.synchronize()
tic = time.time()
loss_vec, grad_vec = softmax_loss_vectorized(W, X_batch, y_batch, reg)
torch.cuda.synchronize()
toc = time.time()
ms_vec = 1000.0 * (toc - tic)
print('vectorized loss: %e computed in %fs' % (loss_vec, ms_vec))

# we use the Frobenius norm to compare the two versions of the gradient.
loss_diff = (loss_naive - loss_vec).abs().item()
grad_diff = torch.norm(grad_naive - grad_vec, p='fro')
print('Loss difference: %.2e' % loss_diff)
print('Gradient difference: %.2e' % grad_diff)
print('Speedup: %.2fX' % (ms_naive / ms_vec))
```

```
naive loss: 2.302841e+00 computed in 242.712498s
vectorized loss: 2.302841e+00 computed in 3.100872s
Loss difference: 0.00e+00
Gradient difference: 6.94e-16
Speedup: 78.27X
```

Let's check that your implementation of the softmax loss is numerically stable.

If either of the following print **nan** then you should double-check the numeric stability of your implementations.

[41]:
```
import ite4052
from linear_classifier import softmax_loss_naive, softmax_loss_vectorized

ite4052.reset_seed(0)
device = data_dict['X_train'].device
dtype = data_dict['X_train'].dtype
D = data_dict['X_train'].shape[1]
C = 10
```

```
# YOUR_TURN??: train_linear_classifier should be same as what you've␣
 ↪implemented in the SVM section
W_ones = torch.ones(D, C, device=device, dtype=dtype)
W, loss_hist = train_linear_classifier(softmax_loss_naive, W_ones,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-8, reg=2.5e4,
                                       num_iters=1, verbose=True)


W_ones = torch.ones(D, C, device=device, dtype=dtype)
W, loss_hist = train_linear_classifier(softmax_loss_vectorized, W_ones,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-8, reg=2.5e4,
                                       num_iters=1, verbose=True)
```

```
iteration 0 / 1: loss 768250002.302585
iteration 0 / 1: loss 768250002.302585
```

Now lets train a softmax classifier with some default hyperparameters:

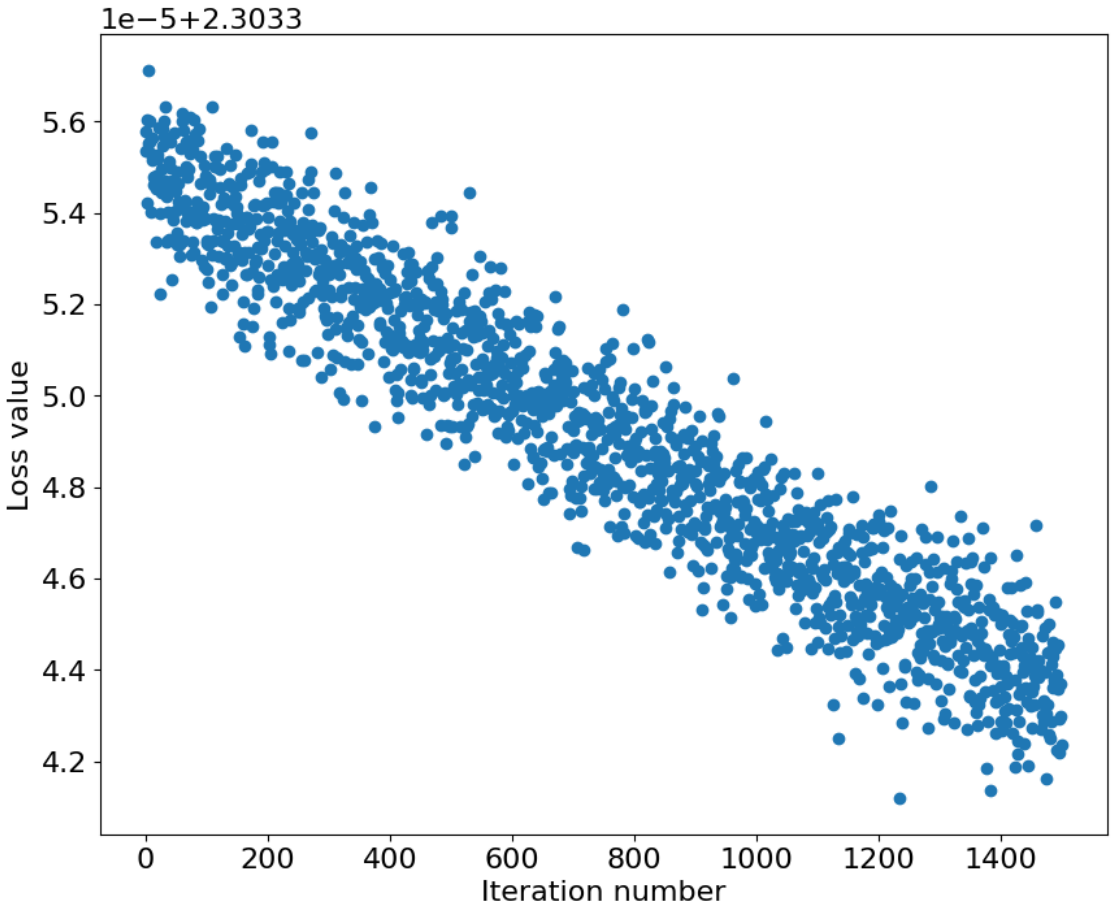
[42]:
```
import ite4052
from linear_classifier import softmax_loss_vectorized

ite4052.reset_seed(0)

torch.cuda.synchronize()
tic = time.time()

# YOUR_TURN??: train_linear_classifier should be same as what you've␣
 ↪implemented in the SVM section
W, loss_hist = train_linear_classifier(softmax_loss_vectorized, None,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-10, reg=2.5e4,
                                       num_iters=1500, verbose=True)

torch.cuda.synchronize()
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 2.303356
iteration 100 / 1500: loss 2.303353
iteration 200 / 1500: loss 2.303354
iteration 300 / 1500: loss 2.303352
iteration 400 / 1500: loss 2.303352
iteration 500 / 1500: loss 2.303351
```

```
iteration 600 / 1500: loss 2.303350
iteration 700 / 1500: loss 2.303349
iteration 800 / 1500: loss 2.303349
iteration 900 / 1500: loss 2.303348
iteration 1000 / 1500: loss 2.303347
iteration 1100 / 1500: loss 2.303348
iteration 1200 / 1500: loss 2.303347
iteration 1300 / 1500: loss 2.303345
iteration 1400 / 1500: loss 2.303345
That took 2.839731s
```

Plot the loss curve:

[43]:
```
plt.plot(loss_hist, 'o')
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



Let's compute the accuracy of current model. It should be less than 10%.

[44]:
```
import ite4052
from linear_classifier import predict_linear_classifier

ite4052.reset_seed(0)

# evaluate the performance on both the training and validation set
# YOUR_TURN??: predict_linear_classifier should be same as what you've␣
 ↪implemented before, in the SVM section
y_train_pred = predict_linear_classifier(W, data_dict['X_train'])
train_acc = 100.0 * (data_dict['y_train'] == y_train_pred).double().mean().
 ↪item()
print('training accuracy: %.2f%%' % train_acc)
y_val_pred = predict_linear_classifier(W, data_dict['X_val'])
val_acc = 100.0 * (data_dict['y_val'] == y_val_pred).double().mean().item()
print('validation accuracy: %.2f%%' % val_acc)
```

```
training accuracy: 8.90%
validation accuracy: 8.54%
```

Now use the validation set to tune hyperparameters (regularization strength and learning rate). You should experiment with different ranges for the learning rates and regularization strengths.

To get full credit for the assignment, your best model found through cross-validation should achieve an accuracy above 0.37 on the validation set.

(Our best model was above 39.8% – did you beat us?)

[45]:
```
import os
import ite4052
from linear_classifier import Softmax, softmax_get_search_params,␣
 ↪test_one_param_set

# YOUR_TURN: find the best learning_rates and regularization_strengths␣
 ↪combination
#            in 'softmax_get_search_params'
learning_rates, regularization_strengths = softmax_get_search_params()
num_models = len(learning_rates) * len(regularization_strengths)

####
# It is okay to comment out the following conditions when you are working on␣
 ↪svm_get_search_params.
# But, please do not forget to reset back to the original setting once you are␣
 ↪done.
if num_models > 25:
  raise Exception("Please do not test/submit more than 25 items at once")
elif num_models < 5:
  raise Exception("Please present at least 5 parameter sets in your final␣
 ↪ipynb")
```

```
####


i = 0
# As before, store your cross-validation results in this dictionary.
# The keys should be tuples of (learning_rate, regularization_strength) and
# the values should be tuples (train_acc, val_acc)
results = {}
best_val = -1.0   # The highest validation accuracy that we have seen so far.
best_softmax_model = None # The Softmax object that achieved the highest␣
 ↪validation rate.
num_iters = 2000 # number of iterations

for lr in learning_rates:
  for reg in regularization_strengths:
    i += 1
    print('Training Softmax %d / %d with learning_rate=%e and reg=%e'
          % (i, num_models, lr, reg))

    ite4052.reset_seed(0)
    cand_softmax_model, cand_train_acc, cand_val_acc =␣
 ↪test_one_param_set(Softmax(), data_dict, lr, reg, num_iters)

    if cand_val_acc > best_val:
      best_val = cand_val_acc
      best_softmax_model = cand_softmax_model # save the classifier
    results[(lr, reg)] = (cand_train_acc, cand_val_acc)


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

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

# save the best model
path = os.path.join(GOOGLE_DRIVE_PATH, 'softmax_best_model.pt')
best_softmax_model.save(path)
```

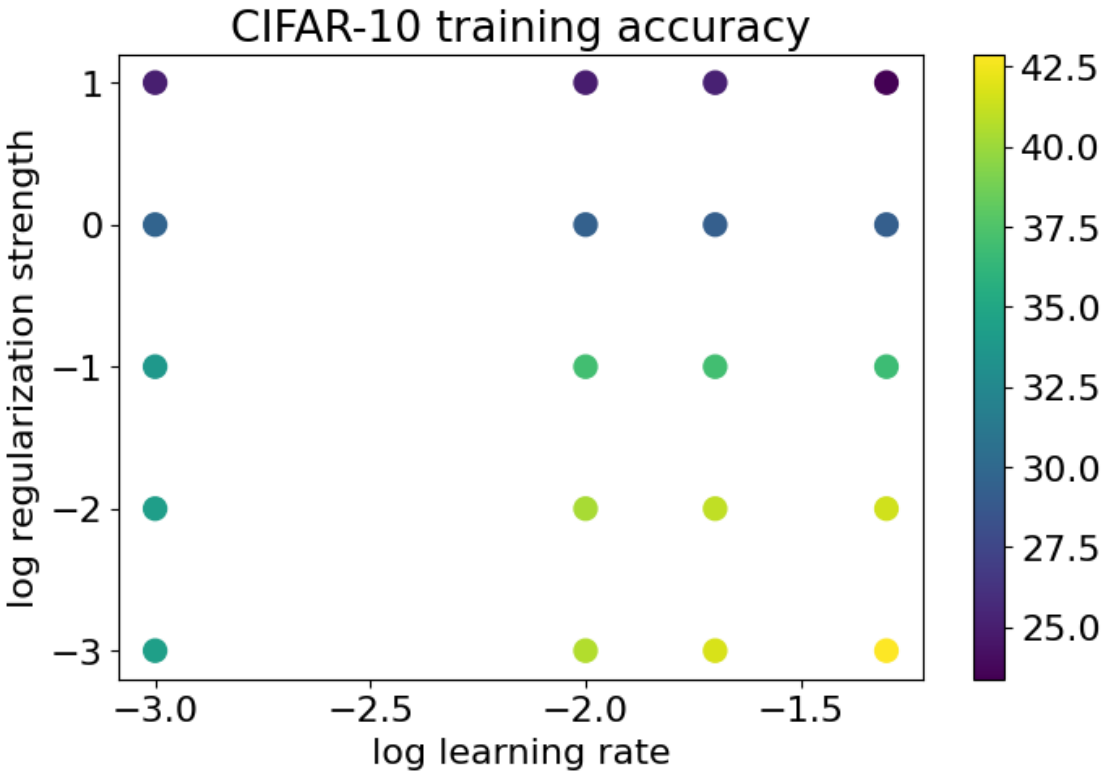
```
Training Softmax 1 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-03
Training Softmax 2 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-02
Training Softmax 3 / 25 with learning_rate=1.000000e-03 and reg=1.000000e-01
Training Softmax 4 / 25 with learning_rate=1.000000e-03 and reg=1.000000e+00
Training Softmax 5 / 25 with learning_rate=1.000000e-03 and reg=1.000000e+01
```

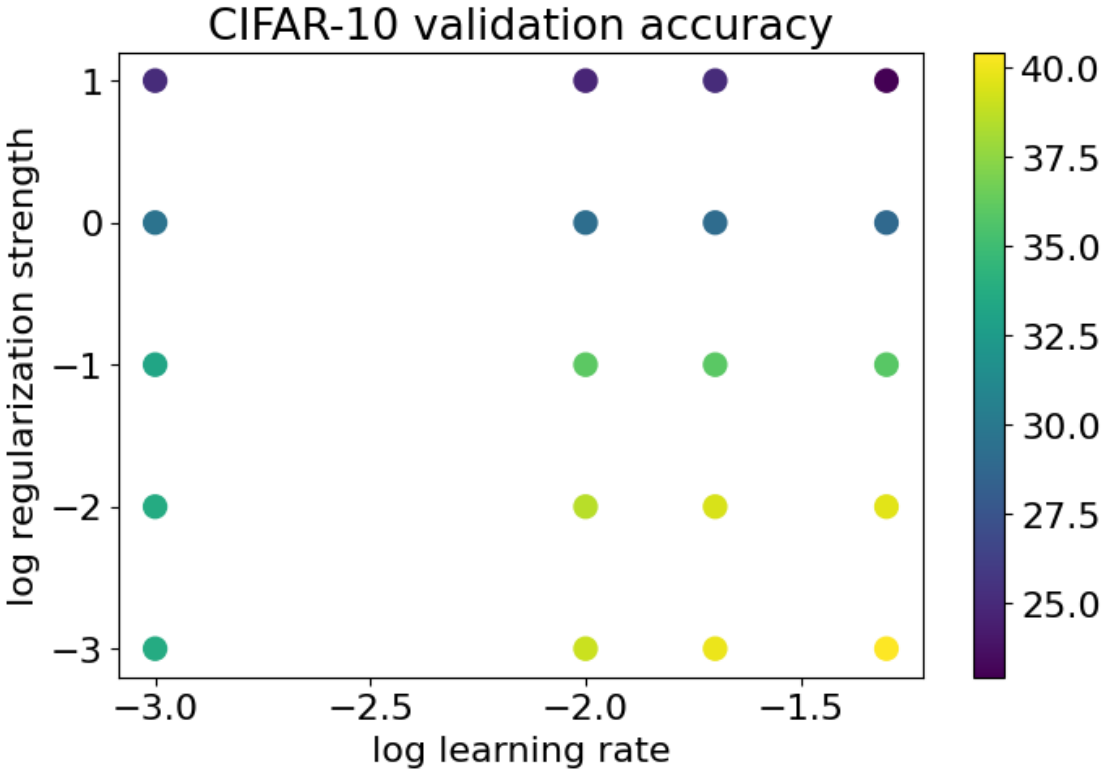
```
Training Softmax 6 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-03
Training Softmax 7 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-02
Training Softmax 8 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-01
Training Softmax 9 / 25 with learning_rate=1.000000e-02 and reg=1.000000e+00
Training Softmax 10 / 25 with learning_rate=1.000000e-02 and reg=1.000000e+01
Training Softmax 11 / 25 with learning_rate=5.000000e-02 and reg=1.000000e-03
Training Softmax 12 / 25 with learning_rate=5.000000e-02 and reg=1.000000e-02
Training Softmax 13 / 25 with learning_rate=5.000000e-02 and reg=1.000000e-01
Training Softmax 14 / 25 with learning_rate=5.000000e-02 and reg=1.000000e+00
Training Softmax 15 / 25 with learning_rate=5.000000e-02 and reg=1.000000e+01
Training Softmax 16 / 25 with learning_rate=2.000000e-02 and reg=1.000000e-03
Training Softmax 17 / 25 with learning_rate=2.000000e-02 and reg=1.000000e-02
Training Softmax 18 / 25 with learning_rate=2.000000e-02 and reg=1.000000e-01
Training Softmax 19 / 25 with learning_rate=2.000000e-02 and reg=1.000000e+00
Training Softmax 20 / 25 with learning_rate=2.000000e-02 and reg=1.000000e+01
Training Softmax 21 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-03
Training Softmax 22 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-02
Training Softmax 23 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-01
Training Softmax 24 / 25 with learning_rate=1.000000e-02 and reg=1.000000e+00
Training Softmax 25 / 25 with learning_rate=1.000000e-02 and reg=1.000000e+01
lr 1.000000e-03 reg 1.000000e-03 train accuracy: 34.322500 val accuracy:
33.640000
lr 1.000000e-03 reg 1.000000e-02 train accuracy: 34.260000 val accuracy:
33.620000
lr 1.000000e-03 reg 1.000000e-01 train accuracy: 33.785000 val accuracy:
33.220000
lr 1.000000e-03 reg 1.000000e+00 train accuracy: 29.605000 val accuracy:
29.620000
lr 1.000000e-03 reg 1.000000e+01 train accuracy: 24.980000 val accuracy:
25.070000
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 40.665000 val accuracy:
39.030000
lr 1.000000e-02 reg 1.000000e-02 train accuracy: 40.365000 val accuracy:
38.550000
lr 1.000000e-02 reg 1.000000e-01 train accuracy: 36.945000 val accuracy:
35.980000
lr 1.000000e-02 reg 1.000000e+00 train accuracy: 29.352500 val accuracy:
29.150000
lr 1.000000e-02 reg 1.000000e+01 train accuracy: 24.945000 val accuracy:
24.720000
lr 2.000000e-02 reg 1.000000e-03 train accuracy: 41.730000 val accuracy:
39.940000
lr 2.000000e-02 reg 1.000000e-02 train accuracy: 41.052500 val accuracy:
39.390000
lr 2.000000e-02 reg 1.000000e-01 train accuracy: 36.987500 val accuracy:
35.920000
lr 2.000000e-02 reg 1.000000e+00 train accuracy: 29.182500 val accuracy:
29.060000
```

```
lr 2.000000e-02 reg 1.000000e+01 train accuracy: 25.155000 val accuracy:
25.080000
lr 5.000000e-02 reg 1.000000e-03 train accuracy: 42.882500 val accuracy:
40.420000
lr 5.000000e-02 reg 1.000000e-02 train accuracy: 41.437500 val accuracy:
39.660000
lr 5.000000e-02 reg 1.000000e-01 train accuracy: 36.767500 val accuracy:
35.870000
lr 5.000000e-02 reg 1.000000e+00 train accuracy: 29.217500 val accuracy:
28.850000
lr 5.000000e-02 reg 1.000000e+01 train accuracy: 23.372500 val accuracy:
22.930000
best validation accuracy achieved during cross-validation: 40.420000
Saved in drive/My Drive/ITE4052/A3/softmax_best_model.pt
```

Run the following to visualize your cross-validation results:

[46]:
```
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap='viridis')
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')
plt.gcf().set_size_inches(8, 5)
plt.show()

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap='viridis')
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.gcf().set_size_inches(8, 5)
plt.show()
```


CIFAR-10 training accuracy
log regularization strength
log learning rate
1
0
−1
−2
−3
−3.0
−2.5
−2.0
−1.5
42.5
40.0
37.5
35.0
32.5
30.0
27.5
25.0


CIFAR-10 validation accuracy
log regularization strength
log learning rate
1
0
−1
−2
−3
−3.0
−2.5
−2.0
−1.5
40.0
37.5
35.0
32.5
30.0
27.5
25.0

Them, evaluate the performance of your best model on test set. To get full credit for this assignment you should achieve a test-set accuracy above 0.36.

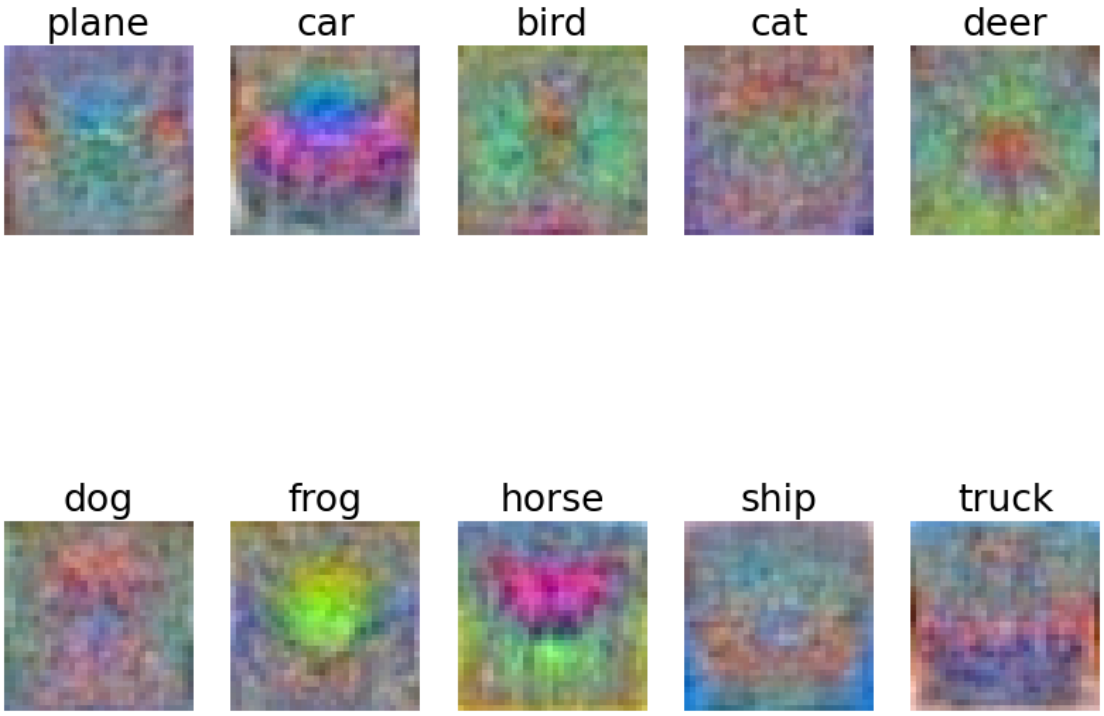
(Our best was just around 39.9% – did you beat us?)

[47]:
```
y_test_pred = best_softmax_model.predict(data_dict['X_test'])
test_accuracy = torch.mean((data_dict['y_test'] == y_test_pred).double())
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.405300
```

Finally, let's visualize the learned weights for each class

[48]:
```
w = best_softmax_model.W[:-1,:] # strip out the bias
w = w.reshape(3, 32, 32, 10)
w = w.transpose(0, 2).transpose(1, 0)

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

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

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


plane
car
bird
cat
deer
dog
frog
horse
ship
truck

two_layer_net

April 18, 2024

1 ITE4052 Assignment 3-2: Two Layer Neural Network

- This material draws from EECS 498-007/598-005 (Justin Johnson)

Before we start, please put your name and HYID in following format Firstname LASTNAME, #00000000 // e.g.) Sukmin Yun, #12345678

Your Answer:
Junwoo Park, #2021006253

2 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment, same as Assignment 2. You'll need to rerun this setup code each time you start the notebook.

First, run this cell load the autoreload extension. This allows us to edit .py source files, and re-import them into the notebook for a seamless editing and debugging experience.

[1]:
```
%load_ext autoreload
%autoreload 2
```

2.1 Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Now recall the path in your Google Drive where you uploaded this notebook, fill it in below. If everything is working correctly then running the folowing cell should print the filenames from the assignment:

```
['two_layer_net.ipynb', 'ite4052', 'two_layer_net.py', 'linear_classifier.py', 'linear_classifi
```

[3]:
```
import os

# TODO: Fill in the Google Drive path where you uploaded the assignment
# Example: If you create a ITE4052 folder and put all the files under A3␣
 ↪folder, then 'ITE4052/A3'
# GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'ITE4052/A3'
GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'ITE4052/A3'
GOOGLE_DRIVE_PATH = os.path.join('drive', 'My Drive',␣
 ↪GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
print(os.listdir(GOOGLE_DRIVE_PATH))
```

```
['collectSubmission.sh', 'makepdf.py', 'collect_submission.ipynb', 'ite4052',
'__pycache__', 'linear_classifier.py', 'svm_best_model.pt',
'softmax_best_model.pt', 'linear_classifier.ipynb', 'nn_best_model.pt',
'two_layer_net.ipynb', 'two_layer_net.py']
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run th following cell to allow us to import from the `.py` files of this assignment. If it works correctly, it should print the message:

```
Hello from two_layer_net.py!
Hello from a3_helpers.py!
```

as well as the last edit time for the file `two_layer_net.py`.

[4]:
```
import sys
sys.path.append(GOOGLE_DRIVE_PATH)

import time, os
os.environ["TZ"] = "US/Eastern"
time.tzset()

from two_layer_net import hello_two_layer_net
hello_two_layer_net()

from ite4052.a3_helpers import hello_helper
hello_helper()

two_layer_net_path = os.path.join(GOOGLE_DRIVE_PATH, 'two_layer_net.py')
two_layer_net_edit_time = time.ctime(os.path.getmtime(two_layer_net_path))
print('two_layer_net.py last edited on %s' % two_layer_net_edit_time)
```

```
Hello from two_layer_net.py!
Hello from a3_helpers.py!
two_layer_net.py last edited on Thu Apr 18 01:24:05 2024
```

2.2 Miscellaneous

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

[5]:
```
import ite4052
import torch
import matplotlib.pyplot as plt
import statistics
import random
import time
%matplotlib inline


plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['font.size'] = 16
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

[6]:
```
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

3 Implementing a Neural Network

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

We train the network with a softmax loss function and L2 regularization on the weight matrices. The network uses a ReLU nonlinearity after the first fully connected layer.

In other words, the network has the following architecture:

input - fully connected layer - ReLU - fully connected layer - softmax

The outputs of the second fully-connected layer are the scores for each class.

Note: When you implment the regularization over W, **please DO NOT multiply the regularization term by 1/2** (no coefficient).

3.1 Play with a toy data

The inputs to our network will be a batch of N (`num_inputs`) D-dimensional vectors (`input_size`); the hidden layer will have H hidden units (`hidden_size`), and we will predict classification scores

for C categories (`num_classes`). This means that the learnable weights and biases of the network will have the following shapes:

- W1: First layer weights; has shape (D, H)
- b1: First layer biases; has shape (H,)
- W2: Second layer weights; has shape (H, C)
- b2: Second layer biases; has shape (C,)

We will use `a3_helpers.get_toy_data` function to generate random weights for a small toy model while we implement the model.

3.1.1 Forward pass: compute scores

Like in the Linear Classifiers exercise, we want to write a function that takes as input the model weights and a batch of images and labels, and returns the loss and the gradient of the loss with respect to each model parameter.

However rather than attempting to implement the entire function at once, we will take a staged approach and ask you to implement the full forward and backward pass one step at a time.

First we will implement the forward pass of the network which uses the weights and biases to compute scores for all inputs in `nn_forward_pass`.

Compute the scores and compare with the answer. The distance gap should be smaller than 1e-10.

[7]:
```
import ite4052
from ite4052.a3_helpers import get_toy_data
from two_layer_net import nn_forward_pass

ite4052.reset_seed(0)
toy_X, toy_y, params = get_toy_data()

# YOUR_TURN: Implement the score computation part of nn_forward_pass
scores, _ = nn_forward_pass(params, toy_X)
print('Your scores:')
print(scores)
print(scores.dtype)
print()
print('correct scores:')
correct_scores = torch.tensor([
        [ 9.7003e-08, -1.1143e-07, -3.9961e-08],
        [-7.4297e-08,  1.1502e-07,  1.5685e-07],
        [-2.5860e-07,  2.2765e-07,  3.2453e-07],
        [-4.7257e-07,  9.0935e-07,  4.0368e-07],
        [-1.8395e-07,  7.9303e-08,  6.0360e-07]], dtype=torch.float32,␣
 ↪device=scores.device)
print(correct_scores)
print()

# The difference should be very small. We get < 1e-10
```

```
scores_diff = (scores - correct_scores).abs().sum().item()
print('Difference between your scores and correct scores: %.2e' % scores_diff)
```

```
Your scores:
tensor([[ 9.7003e-08, -1.1143e-07, -3.9961e-08],
        [-7.4297e-08,  1.1502e-07,  1.5685e-07],
        [-2.5860e-07,  2.2765e-07,  3.2453e-07],
        [-4.7257e-07,  9.0935e-07,  4.0368e-07],
        [-1.8395e-07,  7.9303e-08,  6.0360e-07]], device='cuda:0')
torch.float32

correct scores:
tensor([[ 9.7003e-08, -1.1143e-07, -3.9961e-08],
        [-7.4297e-08,  1.1502e-07,  1.5685e-07],
        [-2.5860e-07,  2.2765e-07,  3.2453e-07],
        [-4.7257e-07,  9.0935e-07,  4.0368e-07],
        [-1.8395e-07,  7.9303e-08,  6.0360e-07]], device='cuda:0')

Difference between your scores and correct scores: 2.24e-11
```

3.1.2 Forward pass: compute loss

Now, we implement the first part of `nn_forward_backward` that computes the data and regularization loss.

For the data loss, we will use the softmax loss. For the regularization loss we will use L2 regularization on the weight matrices `W1` and `W2`; we will not apply regularization loss to the bias vectors `b1` and `b2`.

First, Let's run the following to check your implementation.

We compute the loss for the toy data, and compare with the answer computed by our implementation. The difference between the correct and computed loss should be less than `1e-4`.

[8]:
```
import ite4052
from ite4052.a3_helpers import get_toy_data
from two_layer_net import nn_forward_backward

ite4052.reset_seed(0)
toy_X, toy_y, params = get_toy_data()

# YOUR_TURN: Implement the loss computation part of nn_forward_backward
loss, _ = nn_forward_backward(params, toy_X, toy_y, reg=0.05)
print('Your loss: ', loss.item())
correct_loss = 1.0986121892929077
print('Correct loss: ', correct_loss)
diff = (correct_loss - loss).item()

# should be very small, we get < 1e-4
```

```
print('Difference: %.4e' % diff)
```

```
Your loss:  1.0986121892929077
Correct loss:  1.0986121892929077
Difference: 0.0000e+00
```

3.1.3 Backward pass

Now implement the backward pass for the entire network in `nn_forward_backward`.

After doing so, we will use numeric gradient checking to see whether the analytic gradient computed by our backward pass mateches a numeric gradient.

We will use the functions `ite4052.grad.compute_numeric_gradient` and `ite4052.grad.rel_error` to help with numeric gradient checking. We can learn more about these functions using the `help` command:

[9]:
```
help(ite4052.grad.compute_numeric_gradient)
print('-' * 80)
help(ite4052.grad.rel_error)
```

```
Help on function compute_numeric_gradient in module ite4052.grad:

compute_numeric_gradient(f, x, dLdf=None, h=1e-07)
    Compute the numeric gradient of f at x using a finite differences
    approximation. We use the centered difference:

    df    f(x + h) - f(x - h)
    -- ~= -------------------
    dx           2 * h

    Function can also expand this easily to intermediate layers using the
    chain rule:

    dL   df   dL
    -- = -- * --
    dx   dx   df

    Inputs:
    - f: A function that inputs a torch tensor and returns a torch scalar
    - x: A torch tensor giving the point at which to compute the gradient
    - dLdf: optional upstream gradient for intermediate layers
    - h: epsilon used in the finite difference calculation
    Returns:
    - grad: A tensor of the same shape as x giving the gradient of f at x

--------------------------------------------------------------------------------
Help on function rel_error in module ite4052.grad:
```

```
rel_error(x, y, eps=1e-10)
    Compute the relative error between a pair of tensors x and y,
    which is defined as:

                              max_i |x_i - y_i]|
    rel_error(x, y) = ------------------------------
                      max_i |x_i| + max_i |y_i| + eps

    Inputs:
    - x, y: Tensors of the same shape
    - eps: Small positive constant for numeric stability

    Returns:
    - rel_error: Scalar giving the relative error between x and y
```

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

You should see relative errors less than `1e-4` for all parameters.

[10]:
```
import ite4052
from ite4052.a3_helpers import get_toy_data
from two_layer_net import nn_forward_backward

ite4052.reset_seed(0)

reg = 0.05
toy_X, toy_y, params = get_toy_data(dtype=torch.float64)

# YOUR_TURN: Implement the gradient computation part of nn_forward_backward
#            When you implement the gradient computation part, you may need to
#            implement the `hidden` output in nn_forward_pass, as well.
loss, grads = nn_forward_backward(params, toy_X, toy_y, reg=reg)

for param_name, grad in grads.items():
  param = params[param_name]
  f = lambda w: nn_forward_backward(params, toy_X, toy_y, reg=reg)[0]
  grad_numeric = ite4052.grad.compute_numeric_gradient(f, param)
  error = ite4052.grad.rel_error(grad, grad_numeric)
  print('%s max relative error: %e' % (param_name, error))
```

```
W2 max relative error: 1.261262e-06
b2 max relative error: 3.122771e-09
W1 max relative error: 1.441750e-06
b1 max relative error: 8.239303e-06
```

3.1.4 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers.

Look at the function `nn_train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers.

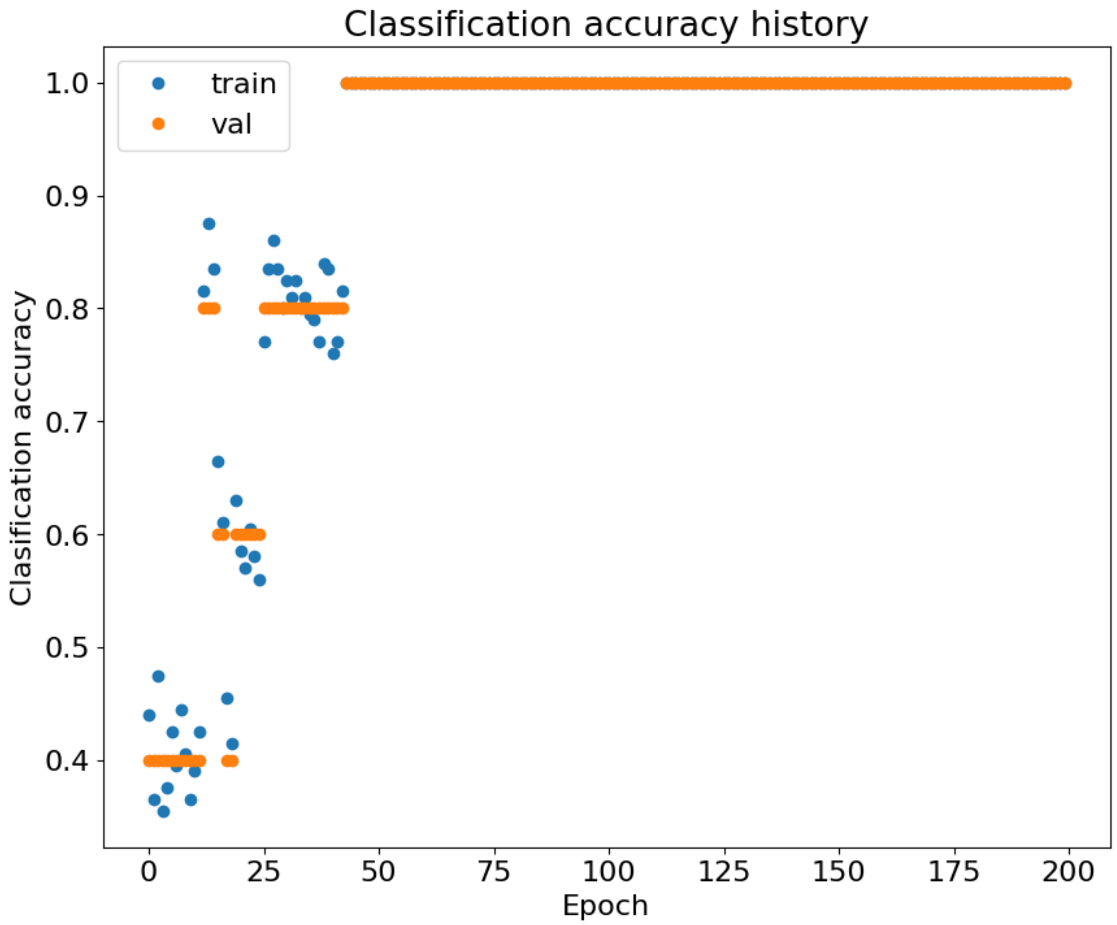
You will also have to implement `nn_predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. Your final training loss should be less than 1.0.

[11]:
```
import ite4052
from ite4052.a3_helpers import get_toy_data
from two_layer_net import nn_forward_backward, nn_train, nn_predict

ite4052.reset_seed(0)
toy_X, toy_y, params = get_toy_data()

# YOUR_TURN: Implement the nn_train function.
#            You may need to check nn_predict function (the "pred_func") as␣
 ↪well.
stats = nn_train(params, nn_forward_backward, nn_predict, toy_X, toy_y, toy_X,␣
 ↪toy_y,
                 learning_rate=1e-1, reg=1e-6,
                 num_iters=200, verbose=False)

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

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

```
Final training loss:  0.5211756229400635
```

[12]:
```
# Plot the loss function and train / validation accuracies
plt.plot(stats['train_acc_history'], 'o', label='train')
plt.plot(stats['val_acc_history'], 'o', label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```

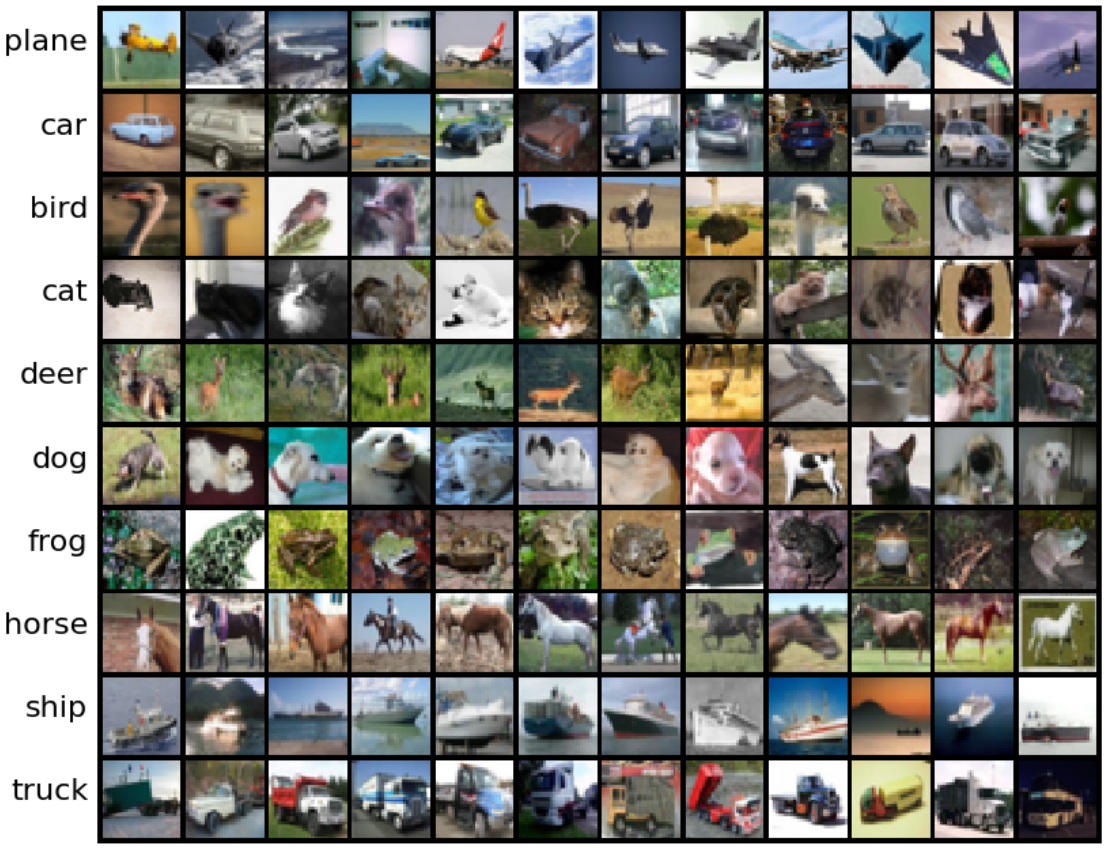


3.2 Testing our NN on a real dataset: CIFAR-10

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

[13]:
```
import ite4052

# Invoke the above function to get our data.
ite4052.reset_seed(0)
data_dict = ite4052.data.preprocess_cifar10(dtype=torch.float64)
print('Train data shape: ', data_dict['X_train'].shape)
print('Train labels shape: ', data_dict['y_train'].shape)
print('Validation data shape: ', data_dict['X_val'].shape)
print('Validation labels shape: ', data_dict['y_val'].shape)
print('Test data shape: ', data_dict['X_test'].shape)
print('Test labels shape: ', data_dict['y_test'].shape)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./cifar-10-python.tar.gz
100%|      | 170498071/170498071 [00:14<00:00, 11758630.92it/s]
Extracting ./cifar-10-python.tar.gz to .
```




```
Train data shape:  torch.Size([40000, 3072])
Train labels shape:  torch.Size([40000])
Validation data shape:  torch.Size([10000, 3072])
Validation labels shape:  torch.Size([10000])
Test data shape:  torch.Size([10000, 3072])
Test labels shape:  torch.Size([10000])
```

3.2.1 Wrap all function into a Class

We will use the class `TwoLayerNet` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are PyTorch tensors.

3.2.2 Train a network

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

[14]:
```
import ite4052
from two_layer_net import TwoLayerNet

input_size = 3 * 32 * 32
hidden_size = 36
num_classes = 10

# fix random seed before we generate a set of parameters
ite4052.reset_seed(0)
net = TwoLayerNet(input_size, hidden_size, num_classes,␣
 ↪dtype=data_dict['X_train'].dtype, device=data_dict['X_train'].device)

# Train the network
stats = net.train(data_dict['X_train'], data_dict['y_train'],
                  data_dict['X_val'], data_dict['y_val'],
                  num_iters=500, batch_size=1000,
                  learning_rate=1e-2, learning_rate_decay=0.95,
                  reg=0.25, verbose=True)

# Predict on the validation set
y_val_pred = net.predict(data_dict['X_val'])
val_acc = 100.0 * (y_val_pred == data_dict['y_val']).double().mean().item()
print('Validation accuracy: %.2f%%' % val_acc)
```

```
iteration 0 / 500: loss 2.302864
iteration 100 / 500: loss 2.302695
iteration 200 / 500: loss 2.302669
iteration 300 / 500: loss 2.302552
iteration 400 / 500: loss 2.302571
Validation accuracy: 9.77%
```

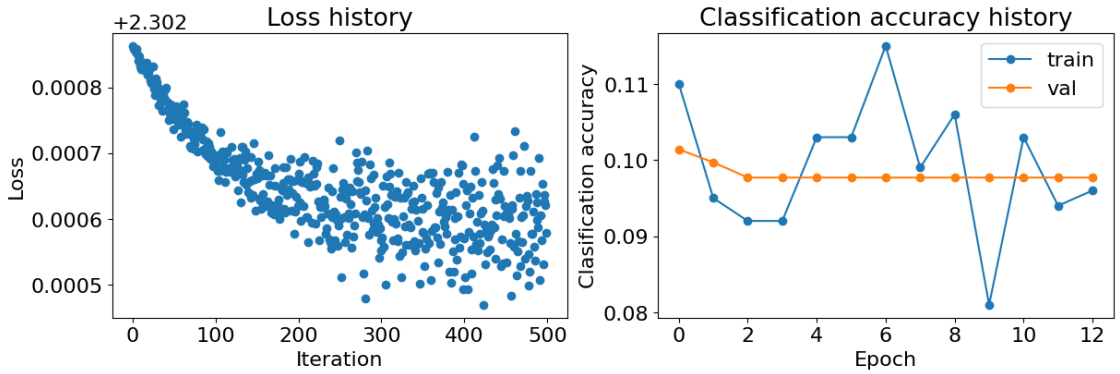
3.2.3 Debug the training

With the default parameters we provided above, you should get a validation accuracy less than 10% 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.

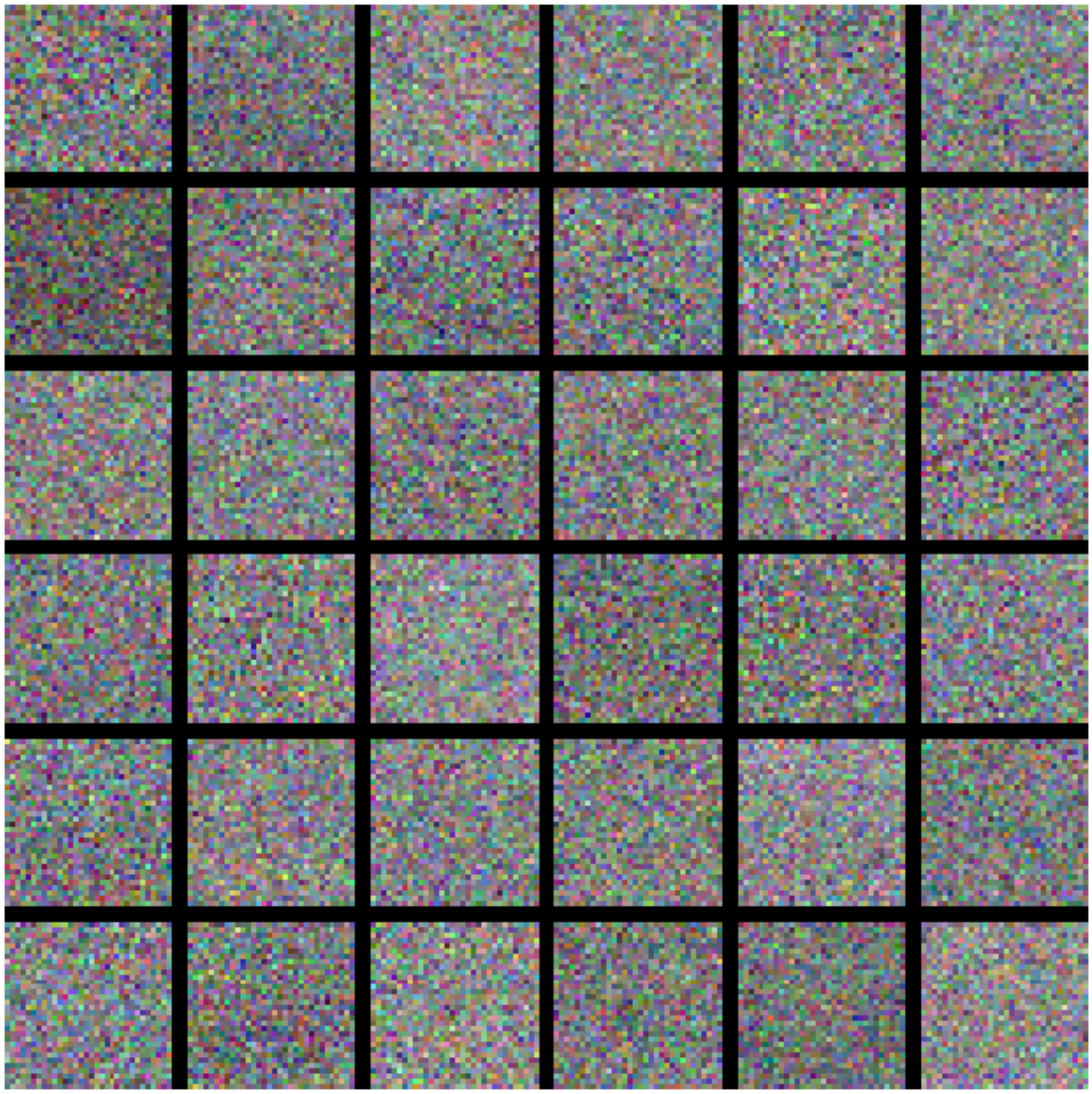
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[15]:
```
# Plot the loss function and train / validation accuracies
from ite4052.a3_helpers import plot_stats

plot_stats(stats)
```



Similar to SVM and Softmax classifier, let's visualize the weights.

[16]:
```
from ite4052.a3_helpers import show_net_weights

show_net_weights(net)
```



3.2.4 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.

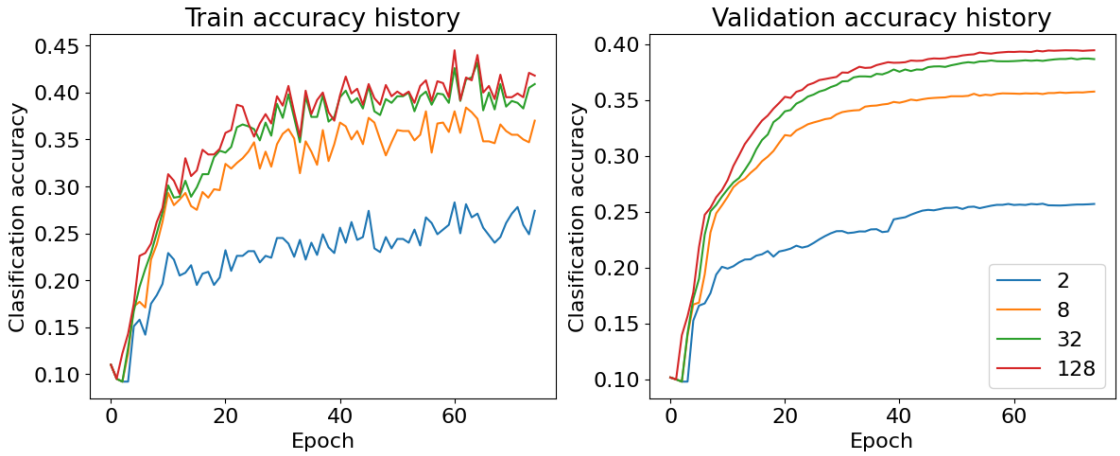
Capacity? Our initial model has very similar performance on the training and validation sets. This suggests that the model is underfitting, and that its performance might improve if we were to increase its capacity.

One way we can increase the capacity of a neural network model is to increase the size of its hidden

layer. Here we investigate the effect of increasing the size of the hidden layer. The performance (as measured by validation-set accuracy) should increase as the size of the hidden layer increases; however it may show diminishing returns for larger layer sizes.

[17]:
```
import ite4052
from ite4052.a3_helpers import plot_acc_curves
from two_layer_net import TwoLayerNet

hidden_sizes = [2, 8, 32, 128]
lr = 0.1
reg = 0.001

stat_dict = {}
for hs in hidden_sizes:
  print('train with hidden size: {}'.format(hs))
  # fix random seed before we generate a set of parameters
  ite4052.reset_seed(0)
  net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device,␣
 ↪dtype=data_dict['X_train'].dtype)
  stats = net.train(data_dict['X_train'], data_dict['y_train'],␣
 ↪data_dict['X_val'], data_dict['y_val'],
            num_iters=3000, batch_size=1000,
            learning_rate=lr, learning_rate_decay=0.95,
            reg=reg, verbose=False)
  stat_dict[hs] = stats

plot_acc_curves(stat_dict)
```

```
train with hidden size: 2
train with hidden size: 8
train with hidden size: 32
train with hidden size: 128
```



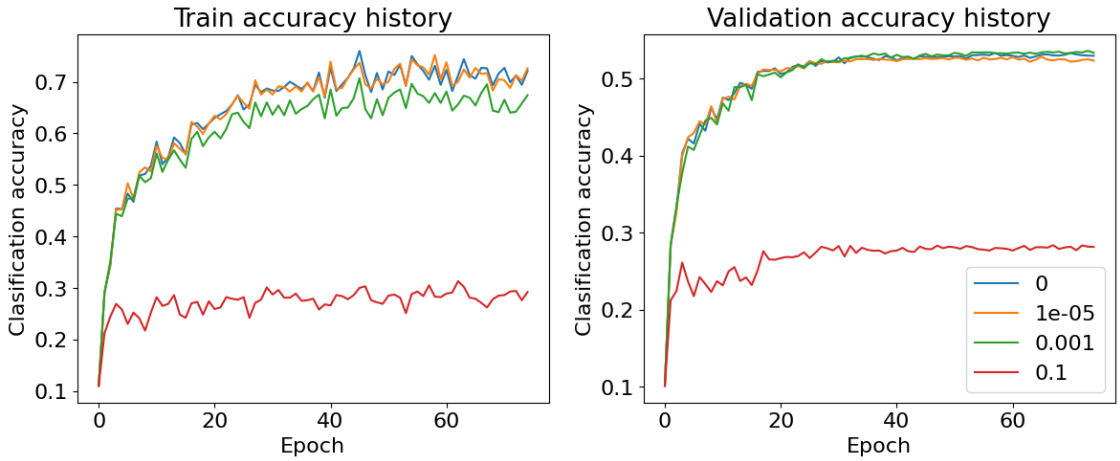
Regularization? Another possible explanation for the small gap we saw between the train and validation accuracies of our model is regularization. In particular, if the regularization coefficient were too high then the model may be unable to fit the training data.

We can investigate the phenomenon empirically by training a set of models with varying regularization strengths while fixing other hyperparameters.

You should see that setting the regularization strength too high will harm the validation-set performance of the model:

[18]:
```
import ite4052
from ite4052.a3_helpers import plot_acc_curves
from two_layer_net import TwoLayerNet

hs = 128
lr = 1.0
regs = [0, 1e-5, 1e-3, 1e-1]

stat_dict = {}
for reg in regs:
  print('train with regularization: {}'.format(reg))
  # fix random seed before we generate a set of parameters
  ite4052.reset_seed(0)
  net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device,␣
 ↪dtype=data_dict['X_train'].dtype)
  stats = net.train(data_dict['X_train'], data_dict['y_train'],␣
 ↪data_dict['X_val'], data_dict['y_val'],
            num_iters=3000, batch_size=1000,
            learning_rate=lr, learning_rate_decay=0.95,
            reg=reg, verbose=False)
  stat_dict[reg] = stats

plot_acc_curves(stat_dict)
```

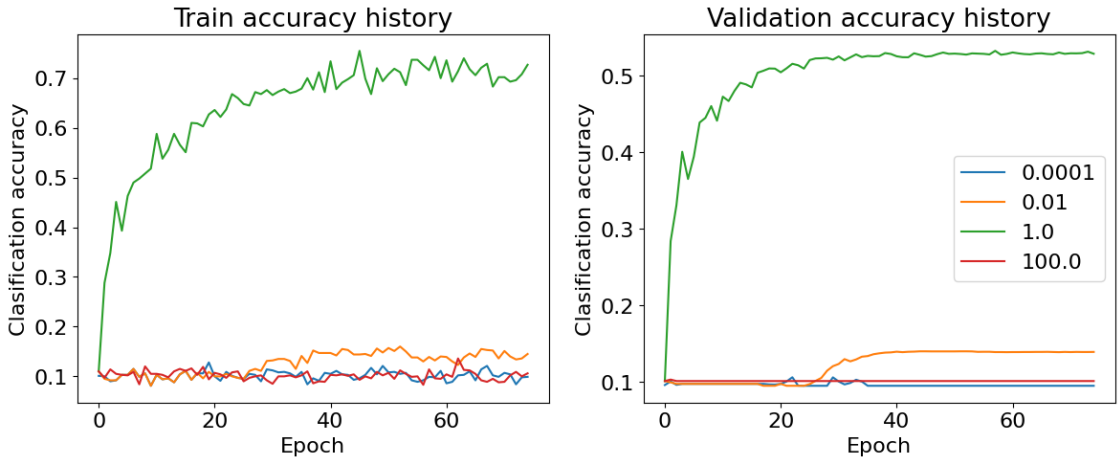
```
train with regularization: 0
train with regularization: 1e-05
train with regularization: 0.001
train with regularization: 0.1
```



Learning Rate? Last but not least, we also want to see the effect of learning rate with respect to the performance.

[19]:
```
import ite4052
from ite4052.a3_helpers import plot_acc_curves
from two_layer_net import TwoLayerNet

hs = 128
lrs = [1e-4, 1e-2, 1e0, 1e2]
reg = 1e-4

stat_dict = {}
for lr in lrs:
  print('train with learning rate: {}'.format(lr))
  # fix random seed before we generate a set of parameters
  ite4052.reset_seed(0)
  net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device,␣
 ↪dtype=data_dict['X_train'].dtype)
  stats = net.train(data_dict['X_train'], data_dict['y_train'],␣
 ↪data_dict['X_val'], data_dict['y_val'],
            num_iters=3000, batch_size=1000,
            learning_rate=lr, learning_rate_decay=0.95,
            reg=reg, verbose=False)
  stat_dict[lr] = stats

plot_acc_curves(stat_dict)
```

```
train with learning rate: 0.0001
train with learning rate: 0.01
train with learning rate: 1.0
train with learning rate: 100.0
```



3.2.5 Tune your hyperparameters

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, number 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.

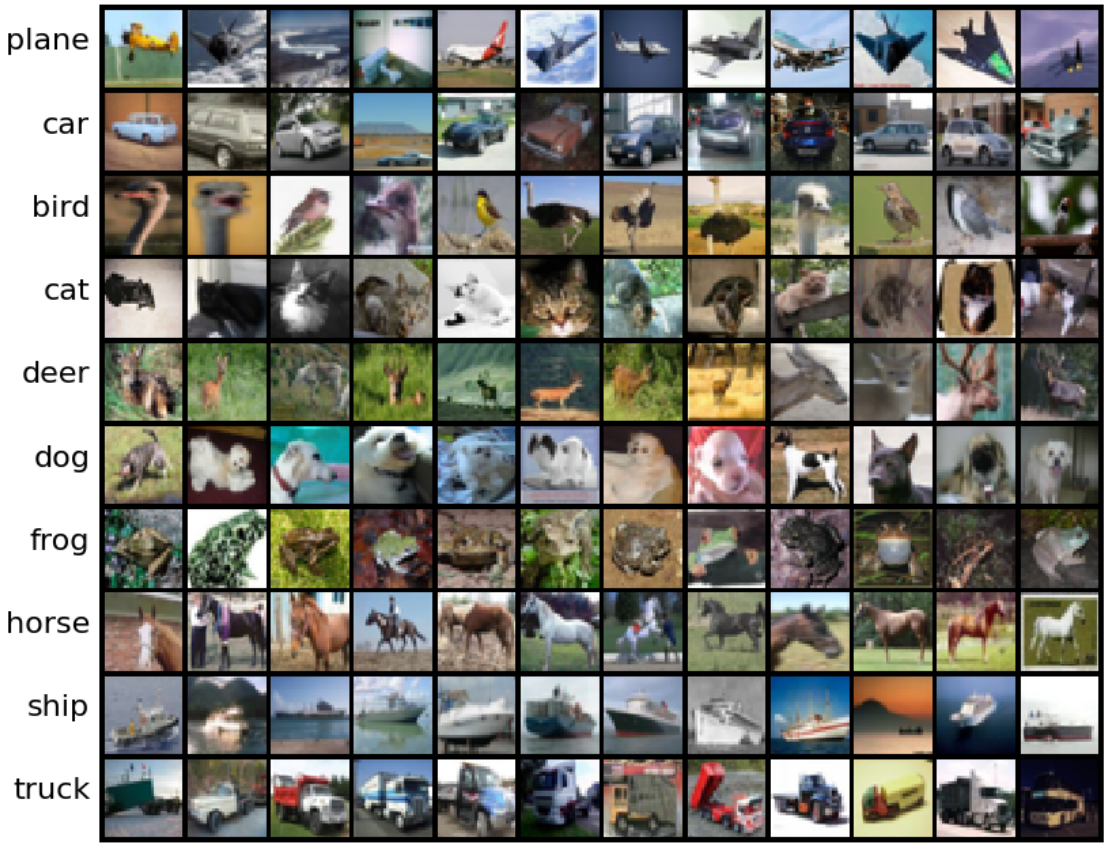
Plots. To guide your hyperparameter search, you might consider making auxiliary plots of training and validation performance as above, or plotting the results arising from different hyperparameter combinations as we did in the Linear Classifier notebook. You should feel free to plot any auxiliary results you need in order to find a good network, but we don't require any particular plots from you.

Approximate results. To get full credit for the assignment, you should achieve a classification accuracy above 50% on the validation set.

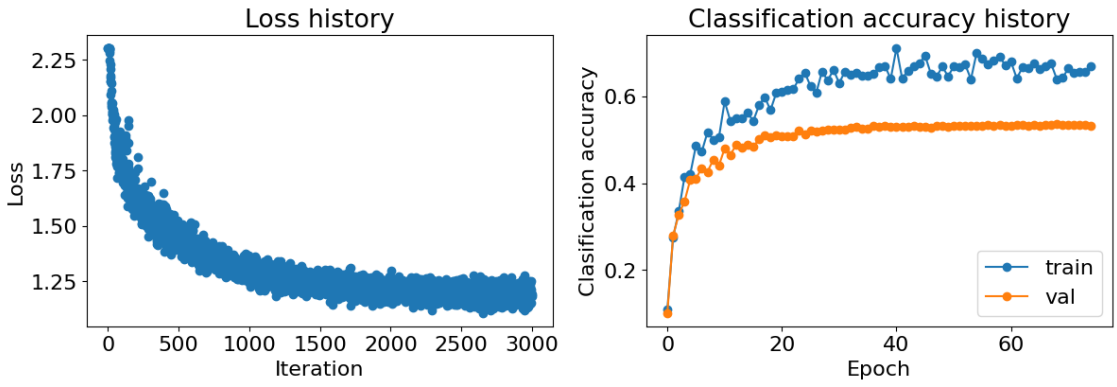
(Our best model gets a validation-set accuracy 56.44% – did you beat us?)

[20]:
```
import os
import ite4052
from two_layer_net import TwoLayerNet, find_best_net, nn_get_search_params

# running this model on float64 may needs more time, so set it as float32
ite4052.reset_seed(0)
data_dict = ite4052.data.preprocess_cifar10(dtype=torch.float32)

# store the best model into this
ite4052.reset_seed(0)
best_net, best_stat, best_val_acc = find_best_net(data_dict,␣
 ↪nn_get_search_params)
print(best_val_acc)
```

```
plot_stats(best_stat)

# save the best model
path = os.path.join(GOOGLE_DRIVE_PATH, 'nn_best_model.pt')
best_net.save(path)
```




```
0.5351999998092651
```

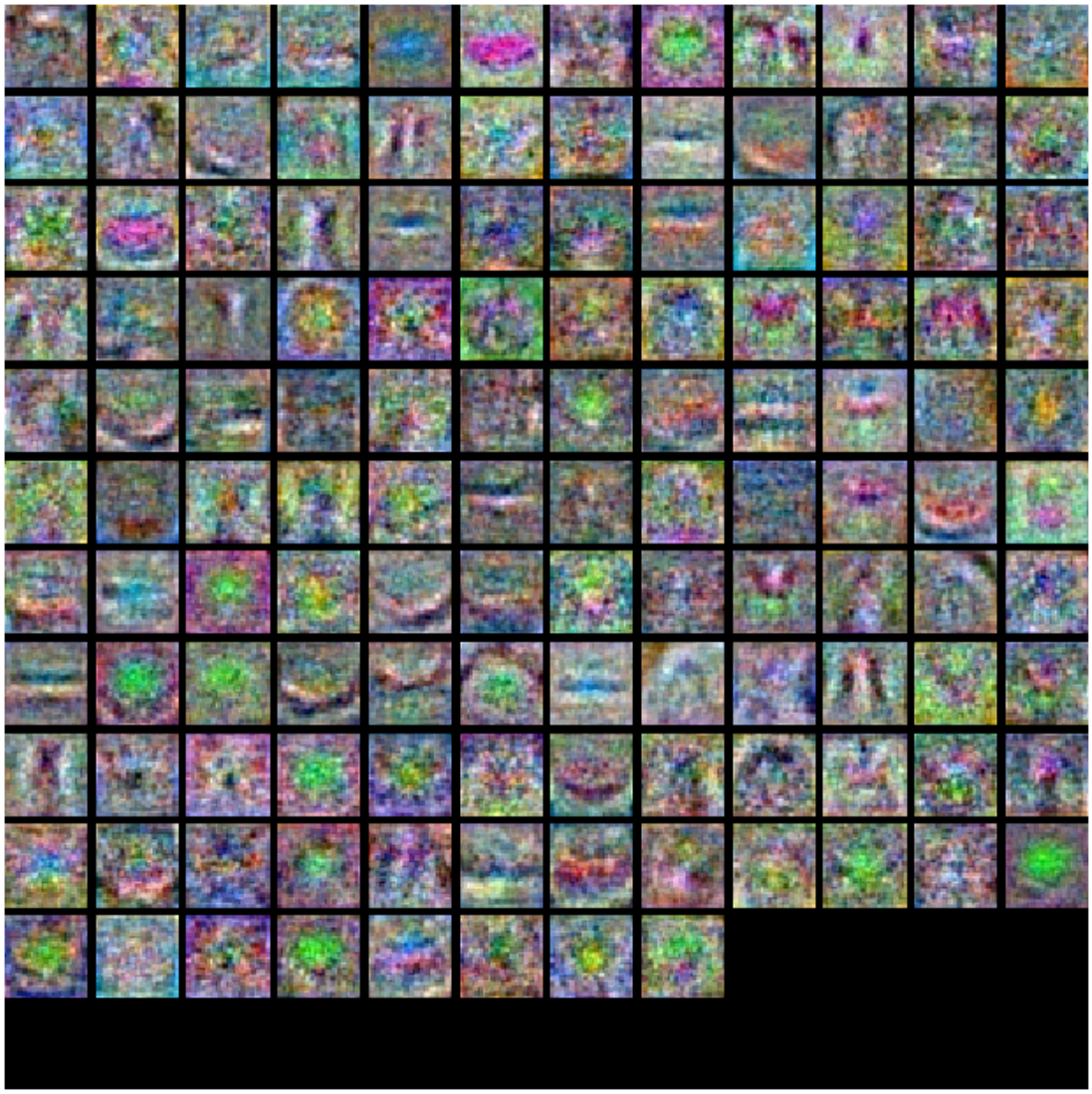


```
Saved in drive/My Drive/ITE4052/A3/nn_best_model.pt
```

[21]:
```
# Check the validation-set accuracy of your best model
y_val_preds = best_net.predict(data_dict['X_val'])
val_acc = 100 * (y_val_preds == data_dict['y_val']).double().mean().item()
print('Best val-set accuracy: %.2f%%' % val_acc)
```

```
Best val-set accuracy: 53.39%
```

[22]:
```
from ite4052.a3_helpers import show_net_weights
# visualize the weights of the best network
show_net_weights(best_net)
```



3.2.6 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set. To get full credit for the assignment, you should achieve over 50% classification accuracy on the test set.

(Our best model gets 56.03% test-set accuracy – did you beat us?)

[23]:
```
y_test_preds = best_net.predict(data_dict['X_test'])
test_acc = 100 * (y_test_preds == data_dict['y_test']).double().mean().item()
print('Test accuracy: %.2f%%' % test_acc)
```

```
Test accuracy: 53.20%
```