

# softmax

April 17, 2020

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

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

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'hw3/cs231n/'

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

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

Go to this URL in a browser: [https://accounts.google.com/o/oauth2/auth?client\\_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect\\_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response\\_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly](https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly)

Enter your authorization code:

ûûûûûûûûûû

Mounted at /content/drive

/content/drive/My Drive

/content

/content/cs231n/datasets

--2020-04-17 21:06:45-- http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30

Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...

connected.

HTTP request sent, awaiting response... 200 OK

```
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz
```

```
cifar-10-python.tar 100%[=====>] 162.60M  66.3MB/s    in 2.5s
```

```
2020-04-17 21:06:48 (66.3 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]
```

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

## 1 Softmax exercise

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

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

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

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

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

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

```
%autoreload 2
```

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

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

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

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

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

```

X_test -= mean_image
X_dev -= mean_image

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

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

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

```

```

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

```

## 1.1 Softmax Classifier

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

```

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

from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

```

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

loss: 2.334785

sanity check: 2.302585

### Inline Question 1

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

*Your Answer :* As all elements in  $W$  are sampled randomly, each value of softmax function is same. In this case, the number of classes is 10, so the value of softmax is  $1/10$ . Therefore, the loss is  $-\log(0.1)$ .

[6]: *# Complete the implementation of softmax\_loss\_naive and implement a (naive) version of the gradient that uses nested loops.*

```
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)
```

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

```
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
# similar to SVM case, do another gradient check with regularization
```

```
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 1.006244 analytic: 1.006244, relative error: 6.284491e-08
numerical: -0.942011 analytic: -0.942011, relative error: 3.399031e-08
numerical: -0.271094 analytic: -0.271094, relative error: 1.843851e-07
numerical: -2.699511 analytic: -2.699511, relative error: 1.454334e-09
numerical: 0.448795 analytic: 0.448795, relative error: 9.323474e-08
numerical: 2.878742 analytic: 2.878742, relative error: 8.398954e-10
numerical: -1.426796 analytic: -1.426797, relative error: 1.117628e-08
numerical: 0.344628 analytic: 0.344628, relative error: 6.126935e-09
numerical: -0.527082 analytic: -0.527082, relative error: 3.277651e-08
numerical: 2.140627 analytic: 2.140627, relative error: 1.791637e-08
numerical: -2.879993 analytic: -2.879993, relative error: 1.651878e-08
numerical: -2.586599 analytic: -2.586599, relative error: 4.352961e-09
numerical: -1.382508 analytic: -1.382508, relative error: 2.651999e-08
numerical: 3.050477 analytic: 3.050477, relative error: 1.492817e-08
numerical: -1.890114 analytic: -1.890114, relative error: 8.598042e-09
numerical: -0.357089 analytic: -0.357089, relative error: 5.700806e-08
numerical: -3.559719 analytic: -3.559719, relative error: 9.203832e-09
numerical: 2.335448 analytic: 2.335448, relative error: 2.864716e-09
numerical: -1.633919 analytic: -1.633919, relative error: 2.227536e-09
numerical: -0.372789 analytic: -0.372789, relative error: 5.703762e-08
```

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

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

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

```
naive loss: 2.334785e+00 computed in 74.226979s
vectorized loss: 2.334785e+00 computed in 0.021271s
Loss difference: 0.000000
Gradient difference: 0.000000
```

### 1.1.1 Stochastic Gradient Descent

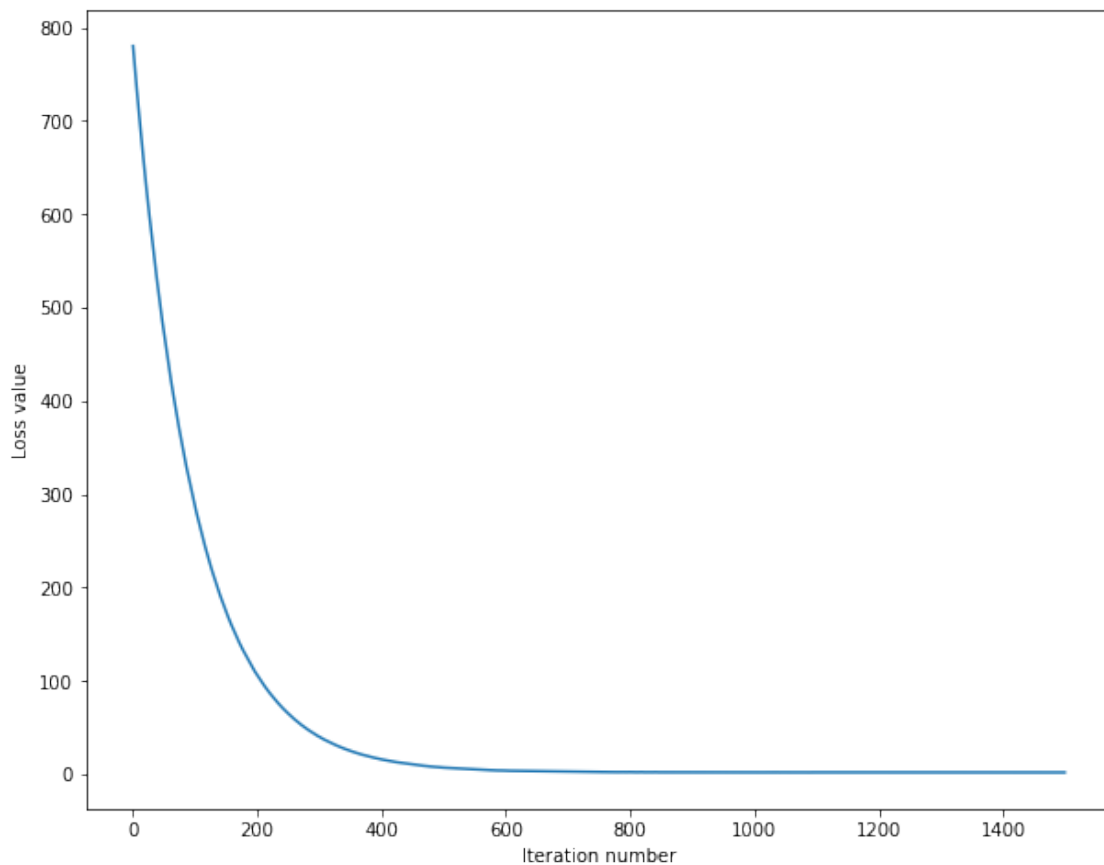
We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

```
[8]: # In the file linear_classifier.py, implement SGD in the function
      # LinearClassifier.train() and then run it with the code below.
      from cs231n.classifiers import Softmax
      softmax = Softmax()
      tic = time.time()
      loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
      num_iters=1500, verbose=True)
      toc = time.time()
      print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 780.333140
iteration 100 / 1500: loss 285.956509
iteration 200 / 1500: loss 105.835215
iteration 300 / 1500: loss 40.068869
```

```
iteration 400 / 1500: loss 16.005266
iteration 500 / 1500: loss 7.153357
iteration 600 / 1500: loss 3.984374
iteration 700 / 1500: loss 2.771053
iteration 800 / 1500: loss 2.366158
iteration 900 / 1500: loss 2.163061
iteration 1000 / 1500: loss 2.072370
iteration 1100 / 1500: loss 2.152889
iteration 1200 / 1500: loss 2.066083
iteration 1300 / 1500: loss 2.037175
iteration 1400 / 1500: loss 2.084858
That took 8.878294s
```

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



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

```
training accuracy: 0.329837
validation accuracy: 0.352000
```

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

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

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

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

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

      for lr in learning_rates:
          for reg in regularization_strengths:
              softmax = Softmax()
              softmax.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=1500,
                           verbose=False)

              train_accuracy = np.mean(y_train == softmax.predict(X_train))
              val_accuracy = np.mean(y_val == softmax.predict(X_val))
```



```

results[(lr, reg)] = (train_accuracy, val_accuracy)

if val_accuracy > best_val:
    best_val = val_accuracy
    best_softmax = softmax

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

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

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

```

```

lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.328959 val accuracy: 0.343000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.303082 val accuracy: 0.330000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.320306 val accuracy: 0.342000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.297531 val accuracy: 0.313000
best validation accuracy achieved during cross-validation: 0.343000

```

```

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

```

```
softmax on raw pixels final test set accuracy: 0.340000
```

### Inline Question 2 - True or False

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

*Your Answer :* True

*Your Explanation :* The SVM loss can be 0 if the gap between the largest score and the second is bigger than the margin. But the Softmax loss cannot be zero even if it can extremely close to zero. So, the Softmax loss will change if a new datapoint is added.

```

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

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

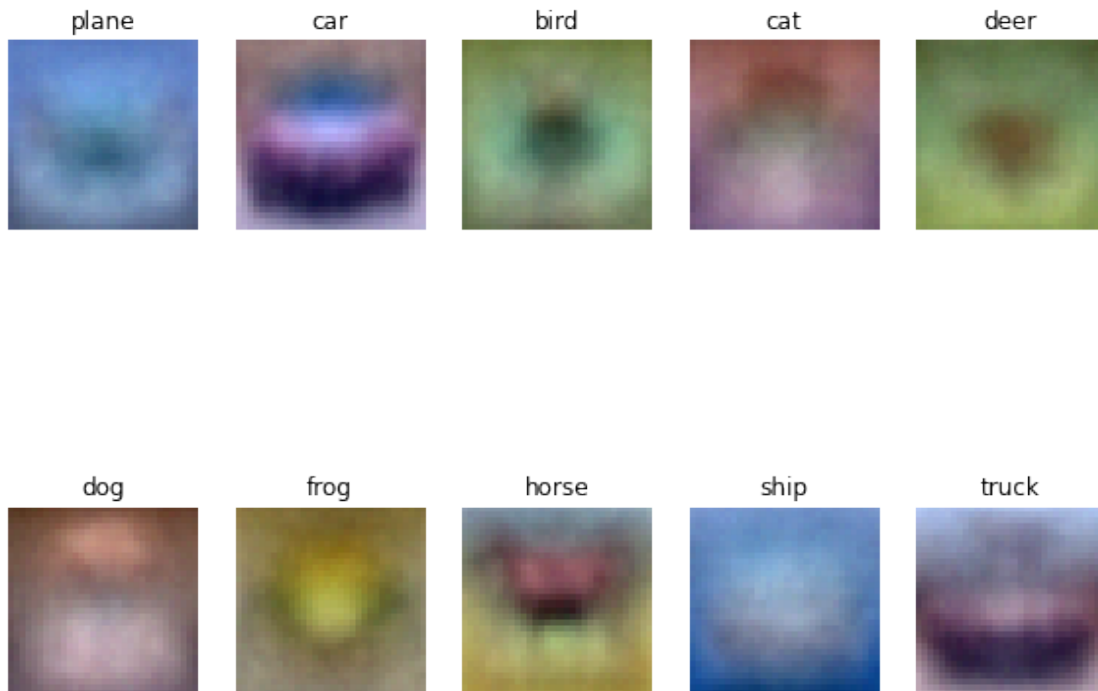
```

```

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

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

```



[0]:

## 2 IMPORTANT

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

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

```
[0]: import os

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

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

# two\_layer\_net

April 17, 2020

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

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

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
# e.g. 'cs231n/assignments/assignment1/cs231n/'
FOLDERNAME = 'hw3/cs231n/'

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

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

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/cs231n/datasets
--2020-04-17 20:29:57-- http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: cifar-10-python.tar.gz

cifar-10-python.tar 100%[=====>] 162.60M  16.0MB/s   in 12s

2020-04-17 20:30:09 (14.1 MB/s) - cifar-10-python.tar.gz saved
[170498071/170498071]
```

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

## 1 Implementing a Neural Network

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

```
[0]: # A bit of setup  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
from cs231n.classifiers.neural_net import TwoLayerNet  
  
%matplotlib inline  
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots  
plt.rcParams['image.interpolation'] = 'nearest'  
plt.rcParams['image.cmap'] = 'gray'  
  
# for auto-reloading external modules  
# see http://stackoverflow.com/questions/1907993/  
#   → autoreload-of-modules-in-ipython  
%load_ext autoreload  
%autoreload 2  
  
def rel_error(x, y):  
    """ returns relative error """  
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

```
[0]: # Create a small net and some toy data to check your implementations.  
# Note that we set the random seed for repeatable experiments.  
  
input_size = 4  
hidden_size = 10  
num_classes = 3
```

```

num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()

```

## 2 Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

```

[0]: scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-0.81233741, -1.27654624, -0.70335995],
    [-0.17129677, -1.18803311, -0.47310444],
    [-0.51590475, -1.01354314, -0.8504215 ],
    [-0.15419291, -0.48629638, -0.52901952],
    [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))

```

Your scores:

```

[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]]

```

```
[-0.00618733 -0.12435261 -0.15226949]]
```

correct scores:

```
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]
```

Difference between your scores and correct scores:

```
3.6802720745909845e-08
```

### 3 Forward pass: compute loss

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

```
[0]: loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

Difference between your loss and correct loss:

```
1.7985612998927536e-13
```

### 4 Backward pass

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

```
[0]: from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward
    ↳pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],
    ↳verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
    ↳grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447625e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738420e-09
```

## 5 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 `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

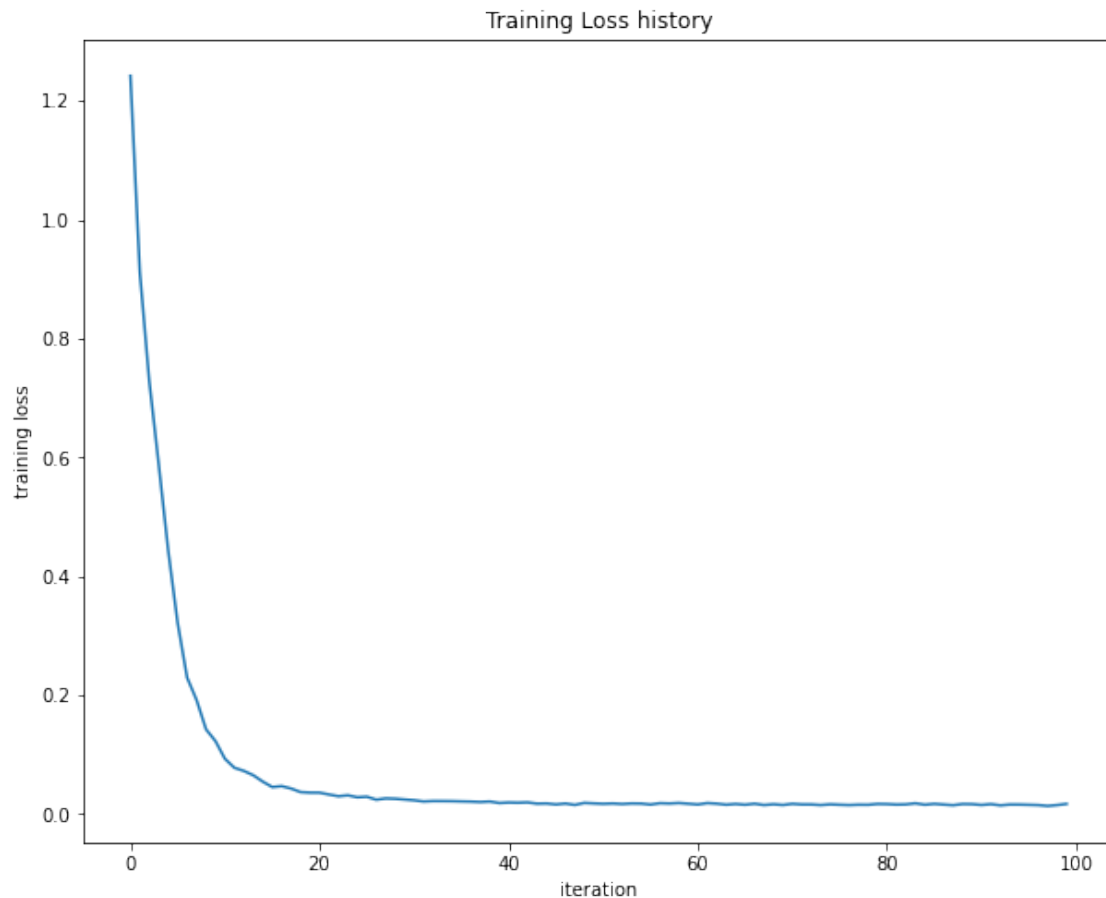
```
[0]: net = init_toy_model()
stats = net.train(X, y, X, y,
                  learning_rate=1e-1, reg=5e-6,
                  num_iters=100, verbose=False)

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

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

```
Final training loss: 0.017149607938732093
```





## 6 Load the data

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

```
[0]: from cs231n.data_utils import load_CIFAR10

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

    # Cleaning up variables to prevent loading data multiple times (which may
    → cause memory issue)
```

```

try:
    del X_train, y_train
    del X_test, y_test
    print('Clear previously loaded data.')
except:
    pass

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

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

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

# Reshape data to rows
X_train = X_train.reshape(num_training, -1)
X_val = X_val.reshape(num_validation, -1)
X_test = X_test.reshape(num_test, -1)

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

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

```

```

Train data shape: (49000, 3072)
Train labels shape: (49000,)
Validation data shape: (1000, 3072)
Validation labels shape: (1000,)

```

```
Test data shape: (1000, 3072)
Test labels shape: (1000,)
```

## 7 Train a network

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

```
[0]: input_size = 32 * 32 * 3
      hidden_size = 50
      num_classes = 10
      net = TwoLayerNet(input_size, hidden_size, num_classes)

      # Train the network
      stats = net.train(X_train, y_train, X_val, y_val,
                        num_iters=1000, batch_size=200,
                        learning_rate=1e-4, learning_rate_decay=0.95,
                        reg=0.25, verbose=True)

      # Predict on the validation set
      val_acc = (net.predict(X_val) == y_val).mean()
      print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy: 0.287
```

## 8 Debug the training

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

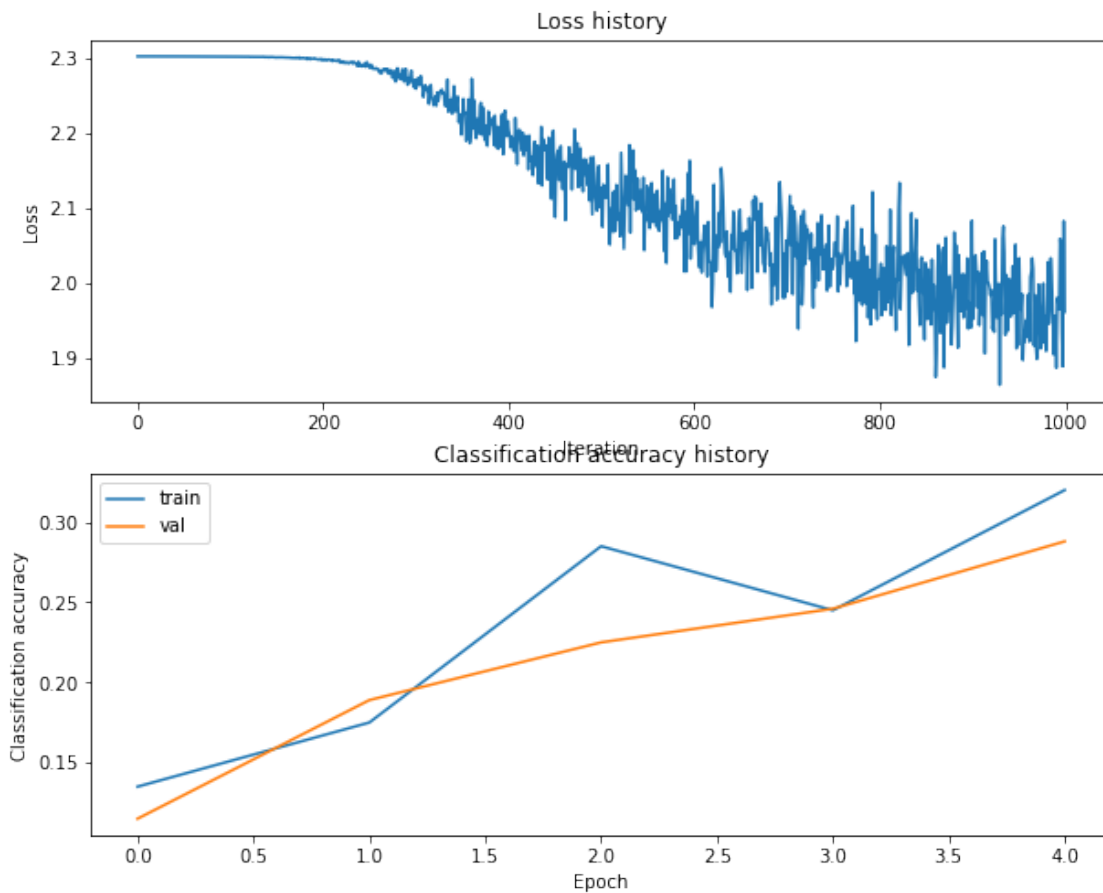
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

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

```
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



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

# Visualize the weights of the network

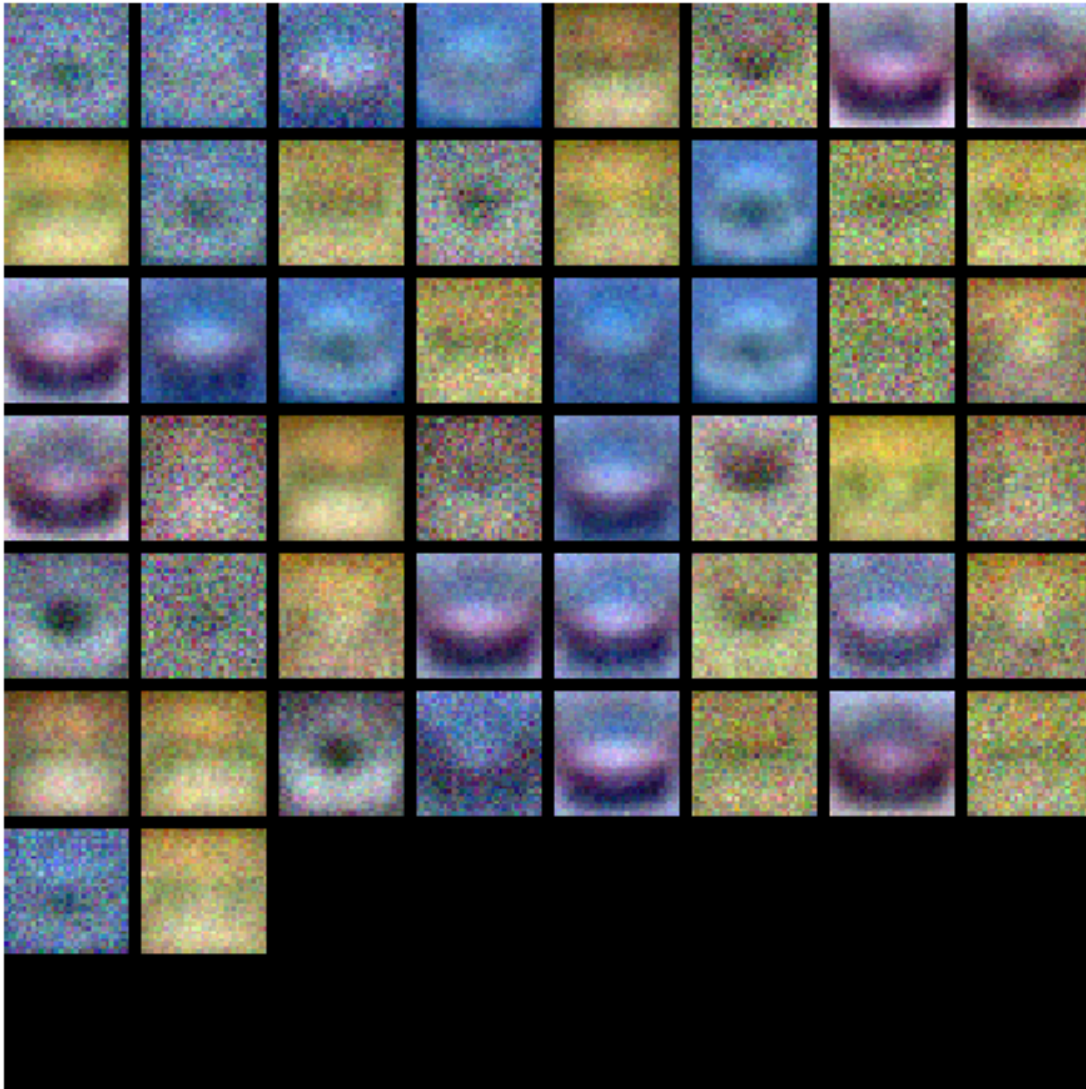
def show_net_weights(net):
```

```

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

show_net_weights(net)

```



## 9 Tune your hyperparameters

**What's wrong?.** Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is

no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

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

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

**Explain your hyperparameter tuning process below.**

*Your Answer :* I used Grid search method. At first, I choosed random hyperparameters, and heuristically choosed some intervals. Then divided the intervals for equal length and checked all that points.

```
[0]: best_net = None # store the best model into this

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

```

best_acc = 0.0
best_stats = []
best_params = {}

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

input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = np.linspace(70, 130, num=3, dtype=int)
learning_rates = np.linspace(0.001, 0.003, num=3, dtype=float)
learning_rate_decays = np.linspace(0.95, 1.0, num=2, dtype=float)
regs = np.linspace(0.001, 0.01, num=3, dtype=float)

i = 0
for hidden_size in hidden_sizes:
    for learning_rate in learning_rates:
        for learning_rate_decay in learning_rate_decays:
            for reg in regs:
                i += 1
                net = TwoLayerNet(input_size, hidden_size, num_classes)
                stats = net.train(X_train, y_train, X_val, y_val,
                                learning_rate=learning_rate,
                                learning_rate_decay=learning_rate_decay,
                                reg=reg,
                                num_iters=1000,
                                batch_size=200,
                                verbose=False)

                val_acc = stats['val_acc_history'][-1]
                if val_acc > best_acc:
                    best_acc = val_acc
                    best_net = net
                    best_stats = stats
                    best_params['hidden_size'] = hidden_size
                    best_params['learning_rate'] = learning_rate
                    best_params['learning_rate_decay'] = learning_rate_decay
                    best_params['reg'] = reg
                print(i, "accuracy =", val_acc, "/ best accuracy =", best_acc)

print("best accuracy = ", best_acc)

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

```

```

1 accuracy = 0.475 / best accuracy = 0.475
2 accuracy = 0.491 / best accuracy = 0.491
3 accuracy = 0.461 / best accuracy = 0.491
4 accuracy = 0.466 / best accuracy = 0.491

```

5 accuracy = 0.467 / best accuracy = 0.491  
6 accuracy = 0.48 / best accuracy = 0.491  
7 accuracy = 0.48 / best accuracy = 0.491  
8 accuracy = 0.476 / best accuracy = 0.491  
9 accuracy = 0.478 / best accuracy = 0.491  
10 accuracy = 0.472 / best accuracy = 0.491  
11 accuracy = 0.478 / best accuracy = 0.491  
12 accuracy = 0.47 / best accuracy = 0.491  
13 accuracy = 0.473 / best accuracy = 0.491  
14 accuracy = 0.457 / best accuracy = 0.491  
15 accuracy = 0.45 / best accuracy = 0.491  
16 accuracy = 0.399 / best accuracy = 0.491  
17 accuracy = 0.402 / best accuracy = 0.491  
18 accuracy = 0.351 / best accuracy = 0.491  
19 accuracy = 0.496 / best accuracy = 0.496  
20 accuracy = 0.477 / best accuracy = 0.496  
21 accuracy = 0.485 / best accuracy = 0.496  
22 accuracy = 0.493 / best accuracy = 0.496  
23 accuracy = 0.455 / best accuracy = 0.496  
24 accuracy = 0.492 / best accuracy = 0.496  
25 accuracy = 0.465 / best accuracy = 0.496  
26 accuracy = 0.469 / best accuracy = 0.496  
27 accuracy = 0.489 / best accuracy = 0.496  
28 accuracy = 0.443 / best accuracy = 0.496  
29 accuracy = 0.479 / best accuracy = 0.496  
30 accuracy = 0.473 / best accuracy = 0.496  
31 accuracy = 0.454 / best accuracy = 0.496  
32 accuracy = 0.418 / best accuracy = 0.496  
33 accuracy = 0.466 / best accuracy = 0.496  
34 accuracy = 0.437 / best accuracy = 0.496  
35 accuracy = 0.407 / best accuracy = 0.496  
36 accuracy = 0.442 / best accuracy = 0.496  
37 accuracy = 0.471 / best accuracy = 0.496  
38 accuracy = 0.473 / best accuracy = 0.496  
39 accuracy = 0.46 / best accuracy = 0.496  
40 accuracy = 0.48 / best accuracy = 0.496  
41 accuracy = 0.501 / best accuracy = 0.501  
42 accuracy = 0.49 / best accuracy = 0.501  
43 accuracy = 0.479 / best accuracy = 0.501  
44 accuracy = 0.467 / best accuracy = 0.501  
45 accuracy = 0.47 / best accuracy = 0.501  
46 accuracy = 0.453 / best accuracy = 0.501  
47 accuracy = 0.482 / best accuracy = 0.501  
48 accuracy = 0.497 / best accuracy = 0.501  
49 accuracy = 0.422 / best accuracy = 0.501  
50 accuracy = 0.448 / best accuracy = 0.501  
51 accuracy = 0.466 / best accuracy = 0.501  
52 accuracy = 0.37 / best accuracy = 0.501

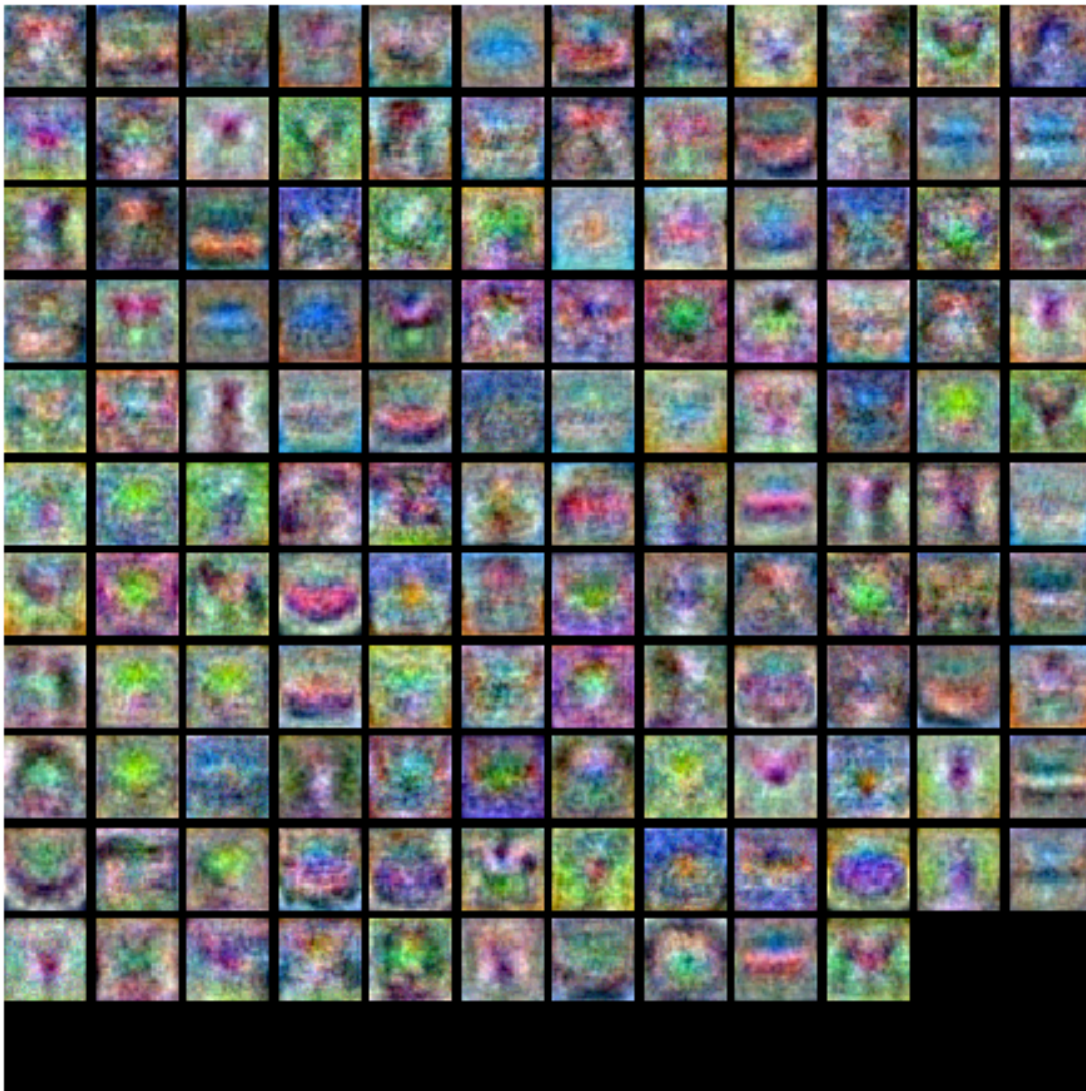


```
53 accuracy = 0.439 / best accuracy = 0.501
54 accuracy = 0.397 / best accuracy = 0.501
best accuracy = 0.501
```

```
[0]: # Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

Validation accuracy: 0.481

```
[0]: # Visualize the weights of the best network
show_net_weights(best_net)
```



## 10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

```
[0]: # Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

Test accuracy: 0.492

### Inline Question

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

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

*Your Answer :* 1, 3

*Your Explanation :* This is the result of overfitting. If I can use larger dataset, then the model can learn from new data. Likewise, the regularization also makes the model not to overfit by decreasing the effect of training data.

---

## 11 IMPORTANT

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

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

```
[0]: import os

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

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