

This is the 2-layer neural network workbook for ECE 239AS Assignment #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyter notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training a two layer neural network.

```
In [1]: import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%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))))
```

Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass

```
In [2]: from nndl.neural_net import TwoLayerNet
```

```
In [3]: # 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()
```

Compute forward pass scores

```
In [4]: ## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
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:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231233889892e-08
```

Forward pass loss

```
In [5]: loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

Loss: 1.071696123862817
Difference between your loss and correct loss:
0.0
```

Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

```
In [6]: from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the back
ward 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, an
d 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('{} max relative error: {}'.format(param_name, rel_error(param
_grad_num, grads[param_name])))

W2 max relative error: 2.9632245016399034e-10
b2 max relative error: 1.2482624742512528e-09
W1 max relative error: 1.2832892417669998e-09
b1 max relative error: 3.172680285697327e-09
```

Training the network

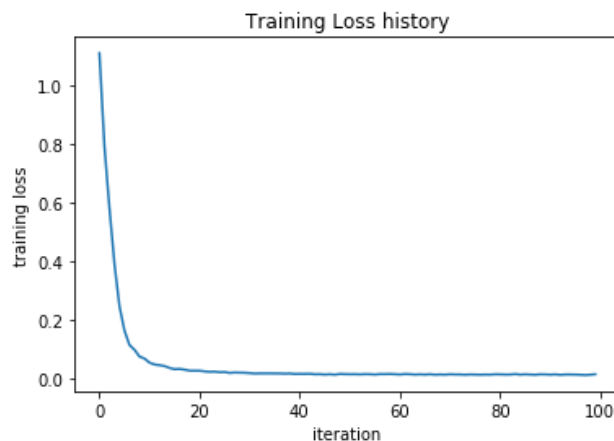
Implement `neural_net.train()` to train the network via stochastic gradient descent, much like the softmax and SVM.

```
In [7]: 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: 0.014497864587765906



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

```
In [8]: 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/CIFAR10'
    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,)
```

Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

```
In [9]: 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)

        # Save this net as the variable subopt_net for later comparison.
        subopt_net = net

iteration 0 / 1000: loss 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.2518259043164135
iteration 400 / 1000: loss 2.1889952350467756
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.990168862308394
iteration 800 / 1000: loss 2.0028276401246856
iteration 900 / 1000: loss 1.94651768178565
Validation accuracy: 0.283
```

Questions:

The training accuracy isn't great.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

```
In [10]: stats['train_acc_history']

Out[10]: [0.095, 0.15, 0.25, 0.25, 0.315]
```

```

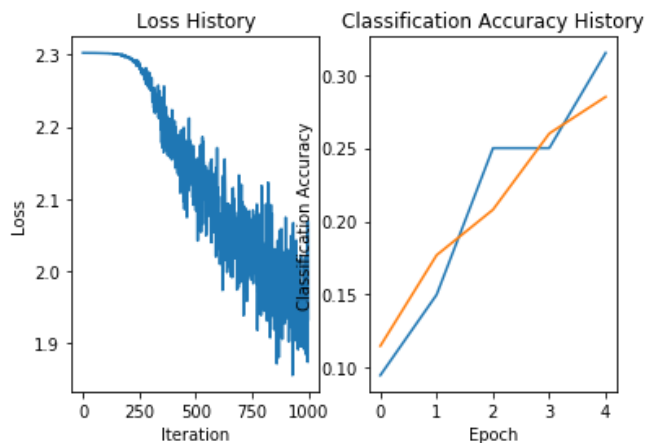
In [11]: # ===== #
# YOUR CODE HERE:
# Do some debugging to gain some insight into why the optimization
# isn't great.
# ===== #

# Plot the loss function and train / validation accuracies

plt.subplot(1, 2, 1)
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(1, 2, 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.show()
# ===== #
# END YOUR CODE HERE
# ===== #

```



Answers:

- (1) Because we have not optimized hyperparameters, the performance is not good.
- (2) We are going to optimize hyperparameters.

Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as `best_net`.

```

In [12]: best_net = None # store the best model into this

# ===== #
# YOUR CODE HERE:
# Optimize over your hyperparameters to arrive at the best neural
# network. You should be able to get over 50% validation accuracy.
# For this part of the notebook, we will give credit based on the
# accuracy you get. Your score on this question will be multiplied b
y:
# min(floor((X - 28%)) / %22, 1)
# where if you get 50% or higher validation accuracy, you get full
# points.
#
# Note, you need to use the same network structure (keep hidden_size =
50)!
# ===== #
best_val = -1
best_stats = None
learning_rates = [1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2]
regularization_strengths = [0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55]
results = {}
iters = 2000
for lr in learning_rates:
    for rs in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, num_classes)

        # Train the network
        stats = net.train(X_train, y_train, X_val, y_val,
                           num_iters=iters, batch_size=200,
                           learning_rate=lr, learning_rate_decay=0.95,
                           reg=rs)

        y_train_pred = net.predict(X_train)
        acc_train = np.mean(y_train == y_train_pred)
        y_val_pred = net.predict(X_val)
        acc_val = np.mean(y_val == y_val_pred)

        results[(lr, rs)] = (acc_train, acc_val)

        if best_val < acc_val:
            best_stats = stats
            best_val = acc_val
            best_net = net

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('learning rates %f regularization strengths %f train accurac
y: %f val accuracy: %f' % (lr, reg, train_accuracy, val_accuracy))

# ===== #
# END YOUR CODE HERE
# ===== #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)

```

```
/home/dennis/Documents/PY_PROGRAM/UCLA_C247/HW3-code/HW3-code/nndl/neural
_net.py:120: RuntimeWarning: overflow encountered in exp
  softmax = np.exp(scores) / np.sum(np.exp(scores),axis=1).reshape(N,1)
/home/dennis/Documents/PY_PROGRAM/UCLA_C247/HW3-code/HW3-code/nndl/neural
_net.py:120: RuntimeWarning: invalid value encountered in true_divide
  softmax = np.exp(scores) / np.sum(np.exp(scores),axis=1).reshape(N,1)
/home/dennis/Documents/PY_PROGRAM/UCLA_C247/HW3-code/HW3-code/nndl/neural
_net.py:121: RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(softmax[range(N), list(y)]))
/home/dennis/Documents/PY_PROGRAM/UCLA_C247/HW3-code/HW3-code/nndl/neural
_net.py:141: RuntimeWarning: invalid value encountered in greater
  dz = dh * (score_l1 > 0).T
```



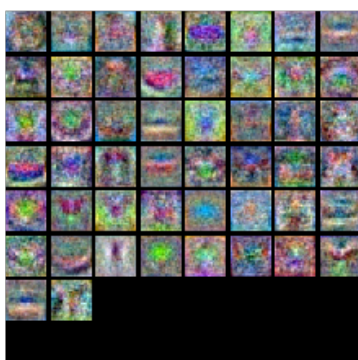
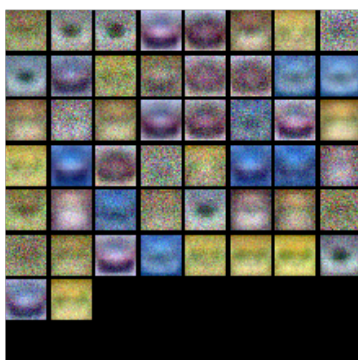
```
learning rates 0.000100 regularization_strengths 0.200000 train accuracy:
0.359857 val accuracy: 0.369000
learning rates 0.000100 regularization_strengths 0.250000 train accuracy:
0.359469 val accuracy: 0.363000
learning rates 0.000100 regularization_strengths 0.300000 train accuracy:
0.361286 val accuracy: 0.366000
learning rates 0.000100 regularization_strengths 0.350000 train accuracy:
0.357061 val accuracy: 0.354000
learning rates 0.000100 regularization_strengths 0.400000 train accuracy:
0.359327 val accuracy: 0.358000
learning rates 0.000100 regularization_strengths 0.450000 train accuracy:
0.359653 val accuracy: 0.369000
learning rates 0.000100 regularization_strengths 0.500000 train accuracy:
0.354306 val accuracy: 0.345000
learning rates 0.000100 regularization_strengths 0.550000 train accuracy:
0.357653 val accuracy: 0.356000
learning rates 0.000500 regularization_strengths 0.200000 train accuracy:
0.500571 val accuracy: 0.477000
learning rates 0.000500 regularization_strengths 0.250000 train accuracy:
0.497347 val accuracy: 0.475000
learning rates 0.000500 regularization_strengths 0.300000 train accuracy:
0.496796 val accuracy: 0.480000
learning rates 0.000500 regularization_strengths 0.350000 train accuracy:
0.497592 val accuracy: 0.484000
learning rates 0.000500 regularization_strengths 0.400000 train accuracy:
0.490061 val accuracy: 0.472000
learning rates 0.000500 regularization_strengths 0.450000 train accuracy:
0.496918 val accuracy: 0.475000
learning rates 0.000500 regularization_strengths 0.500000 train accuracy:
0.495633 val accuracy: 0.475000
learning rates 0.000500 regularization_strengths 0.550000 train accuracy:
0.492347 val accuracy: 0.478000
learning rates 0.001000 regularization_strengths 0.200000 train accuracy:
0.531714 val accuracy: 0.504000
learning rates 0.001000 regularization_strengths 0.250000 train accuracy:
0.530571 val accuracy: 0.508000
learning rates 0.001000 regularization_strengths 0.300000 train accuracy:
0.527102 val accuracy: 0.483000
learning rates 0.001000 regularization_strengths 0.350000 train accuracy:
0.524714 val accuracy: 0.497000
learning rates 0.001000 regularization_strengths 0.400000 train accuracy:
0.521490 val accuracy: 0.489000
learning rates 0.001000 regularization_strengths 0.450000 train accuracy:
0.513837 val accuracy: 0.492000
learning rates 0.001000 regularization_strengths 0.500000 train accuracy:
0.523367 val accuracy: 0.491000
learning rates 0.001000 regularization_strengths 0.550000 train accuracy:
0.523694 val accuracy: 0.502000
learning rates 0.005000 regularization_strengths 0.200000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.250000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.300000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.350000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.400000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.450000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.500000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.005000 regularization_strengths 0.550000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.010000 regularization_strengths 0.200000 train accuracy:
0.100265 val accuracy: 0.087000
learning rates 0.010000 regularization_strengths 0.250000 train accuracy:
0.100265 val accuracy: 0.087000
```

```
In [13]: from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.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(subopt_net)
show_net_weights(best_net)
```



Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

Answer:

(1) From suboptimal one I can only see blurry patterns, but from the best one I arrived at I can see clearer patterns.

Evaluate on test set

```
In [14]: test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

Test accuracy: 0.482