two_layer_nn

January 31, 2022

0.1 This is the 2-layer neural network notebook for ECE C147/C247 Homework #3

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

Please print out the notebook entirely when completed.

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

[1]:
```
import random
import numpy as np
from utils.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))))
```

0.2 Toy example

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

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

[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()
```

0.2.1 Compute forward pass scores

[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:
3.381231204052648e-08
```

0.2.2 Forward pass loss

```
[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
```

```
[6]: print(loss)
```

```
1.071696123862817
```

0.2.3 Backward pass

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

```
[7]: from utils.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('{} max relative error: {}'.format(param_name,␣
 ↪rel_error(param_grad_num, grads[param_name])))
```

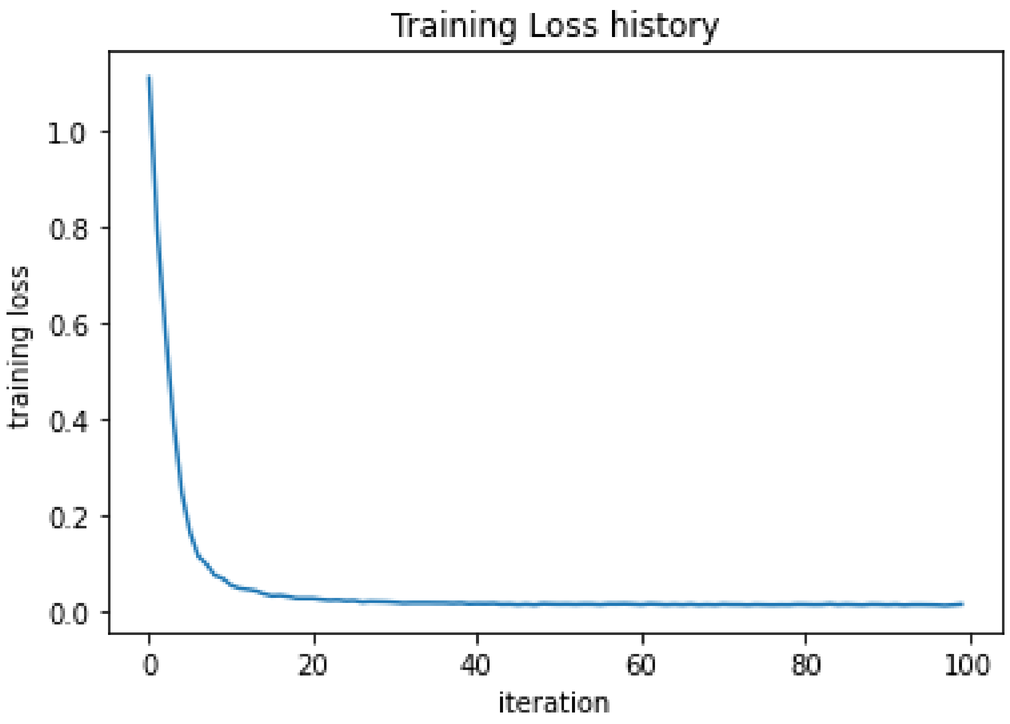
```
W2 max relative error: 2.9632227682005116e-10
b2 max relative error: 1.8392106647421603e-10
W1 max relative error: 1.283285235125835e-09
b1 max relative error: 3.172680092703762e-09
```

0.2.4 Training the network

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

[8]:
```python
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:  0.014497864587765 95
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[9]:
```python
from utils.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.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cifar-10-batches-py'
    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,)
```

0.3.1 Running SGD

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

[10]:
```
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 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.9901688623083942
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

0.4 Questions:

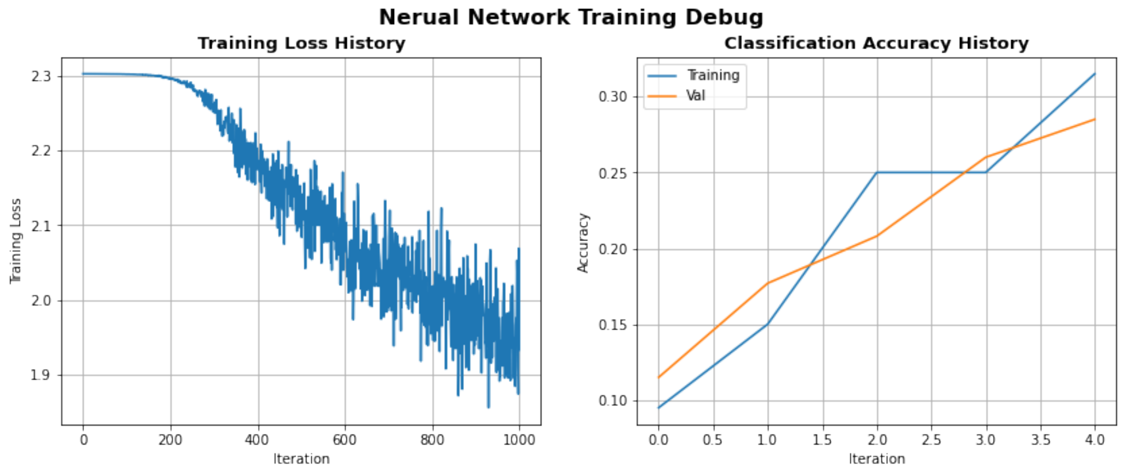
The training accuracy isn't great.

(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)?

[ ]:
```
stats['train_acc_history']
```

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


fig2,(ax1,ax2) = plt.subplots(1,2,figsize=(14,5))
fig2.suptitle("Nerual Network Training Debug",fontweight='bold',fontsize = 16)

# Loss Function
ax1.plot(stats['loss_history'])
ax1.set_title('Training Loss History',fontweight='bold',fontsize = 13)
ax1.set_xlabel('Iteration')
ax1.set_ylabel('Training Loss')
ax1.grid()

# Accuracies
ax2.plot(stats['train_acc_history'])
ax2.plot(stats['val_acc_history'])
ax2.set_title('Classification Accuracy History',fontweight='bold',fontsize = 13)
ax2.set_xlabel('Iteration')
ax2.legend(['Training','Val'])
ax2.set_ylabel('Accuracy')
ax2.grid()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```



0.5 Answers:

(1) The primary issue appears to be **a low learning rate contributing to a lack of convergence in the current number of training iterations**. Because the learning rate is likely too low, we see that the training loss is nearly flat for the first ≈ 200 iterations. Similarly, we do not see any sign of convergence or leveling off in the two classification accuracies.

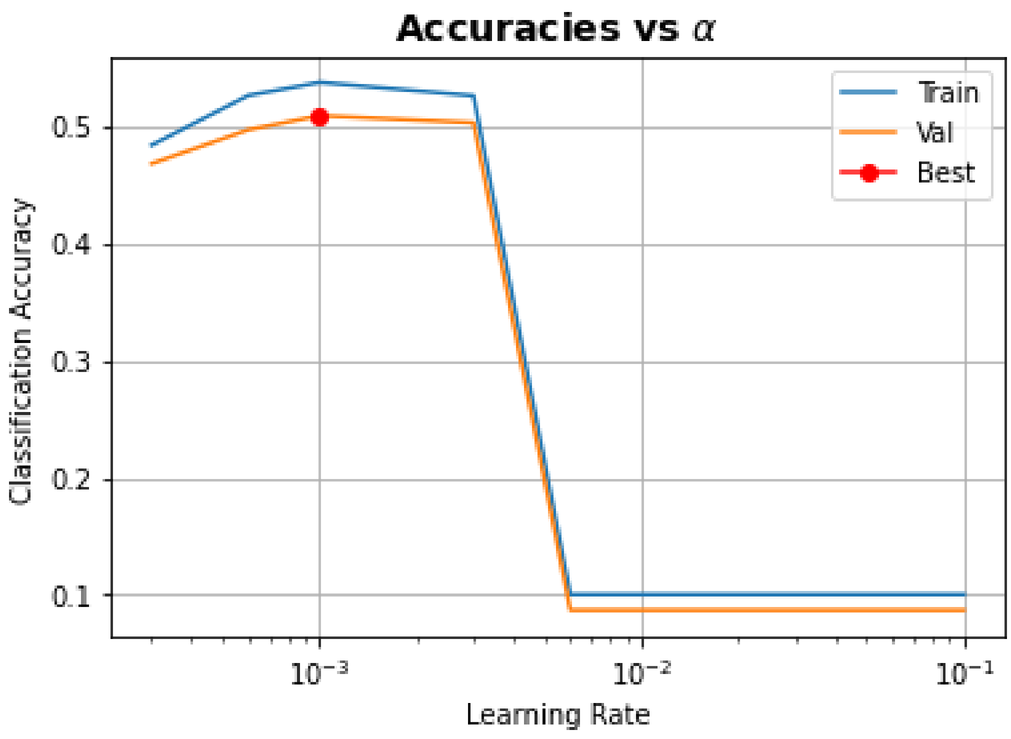
(2) The above analyis indicates we need **a higher learning rate and/or more iterations of training. Both will be tried below**.

0.6 Optimize the neural network

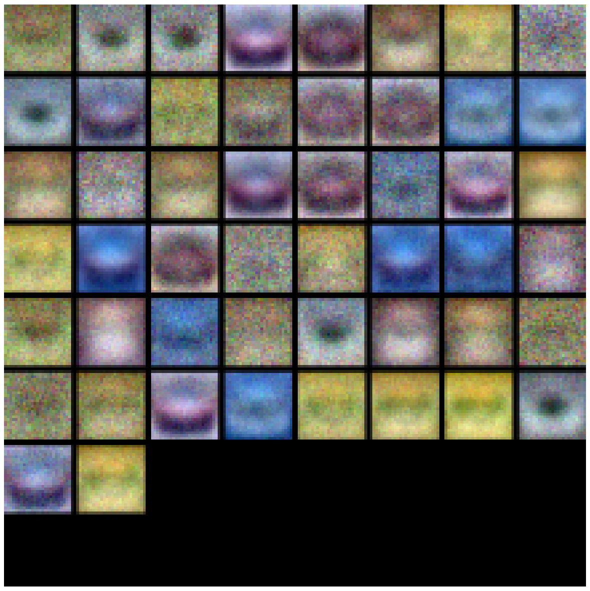
Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

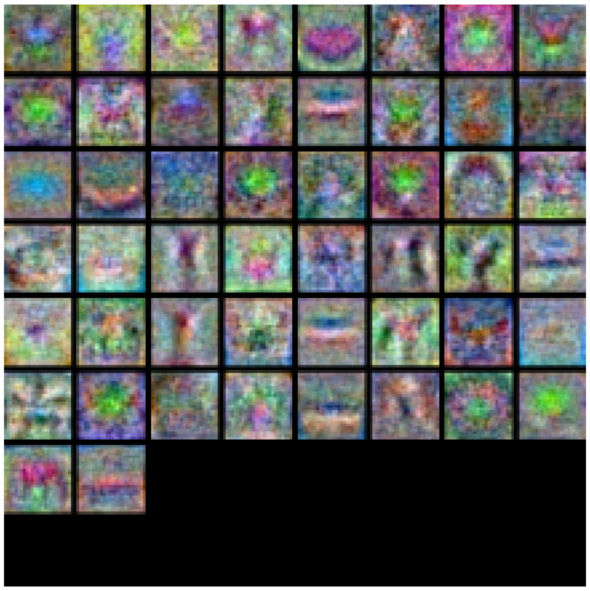
[12]:
```
best_net = None # store the best model into this

# ================================================================ #
# 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 by:
#      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)!
# ================================================================ #
NUM_ITERS = 3000 # Try going up to 3k iterations with various learning rates␣
 ↪(alphas)

best_acc = -1
best_i = -1
ALPHAS = [1e-1, 6e-2, 3e-2, 1e-2, 6e-3, 3e-3, 1e-3,6e-4, 3e-4]
accuracies = np.empty([len(ALPHAS),2])
accuracies[:] = np.nan
for i,alpha in enumerate(ALPHAS):
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=NUM_ITERS, batch_size=200,
                learning_rate=alpha, learning_rate_decay=0.95,
                reg=0.55)

    train_acc = np.mean(y_train == net.predict(X_train))
    val_acc = np.mean(y_val == net.predict(X_val))
    accuracies[i,:] = np.array([train_acc, val_acc])

    print ('Val Accuracy: {:.2%} for alpha = {:.4f}'.format(val_acc, alpha))
    if val_acc > best_acc:
        best_i = i
        best_acc = val_acc
        best_net = net
        best_stats = stats

print ('Best validation accuracy is alpha = {:.4f}: {:.1%} with train accuracy:␣
 ↪{:.1%}'.format(ALPHAS[best_i],accuracies[best_i,1],accuracies[best_i,0]))

plt.plot(ALPHAS,accuracies)
plt.title(r"Accuracies vs $\alpha$",fontweight='bold',fontsize = 14)
plt.xscale('log')
plt.grid()
plt.plot(ALPHAS[best_i],accuracies[best_i,1],marker='o',c='r')
plt.legend(['Train','Val','Best'])
plt.xlabel('Learning Rate')
plt.ylabel('Classification Accuracy')

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

```
/Users/sunaybhat/Documents/GitHub/C247-NNs-DL/HW
3/hw3-code/nndl/neural_net.py:112: RuntimeWarning: overflow encountered in exp
  softmax_loss = np.sum(np.log(np.sum(np.exp(scores), axis = 1)) -
scores[np.arange(N), y])
/Users/sunaybhat/Documents/GitHub/C247-NNs-DL/HW
3/hw3-code/nndl/neural_net.py:134: RuntimeWarning: overflow encountered in exp
  dL_dz2 = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)
/Users/sunaybhat/Documents/GitHub/C247-NNs-DL/HW
3/hw3-code/nndl/neural_net.py:134: RuntimeWarning: invalid value encountered in
true_divide
  dL_dz2 = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)

Val Accuracy: 8.70% for alpha = 0.1000
Val Accuracy: 8.70% for alpha = 0.0600
Val Accuracy: 8.70% for alpha = 0.0300
Val Accuracy: 8.70% for alpha = 0.0100
Val Accuracy: 8.70% for alpha = 0.0060
Val Accuracy: 50.30% for alpha = 0.0030
Val Accuracy: 50.90% for alpha = 0.0010
Val Accuracy: 49.70% for alpha = 0.0006
Val Accuracy: 46.80% for alpha = 0.0003
Best validation accuracy is alpha = 0.0010: 50.9% with train accuracy: 53.7%
Validation accuracy:  0.509
```



[13]:
```
from utils.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)
```





0.7 Question:

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

0.8 Answer:

(1) In the suboptimal net, we notice there is **less variation than the best net. Many of the weights look nearly identical. This implies more training, and a higher learning rate, found weights that vary more and produce a better set of descriminating features**.

0.9 Evaluate on test set

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

```
Test accuracy:  0.513
```