**cs231n - assignment1 - neural net 梯度推导 - 好记性不如烂笔头 - 博客频道 - CSDN.NET**

| | |
|---|---|
| **笔记本：** | 机器学习 |
| **创建时间：** | 4/10/2017 11:59 PM |
| **标签：** | 神经网络 |
| **URL：** | http://blog.csdn.net/vc461515457/article/details/51944683 |

cs231n

# cs231n - assignment1 - neural net 梯度推导



## 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.

可以先看看之前softmax的梯度推导方法，这里开始采用矩阵的形式来推导梯度，而且将逐级推导梯度，这种方式有很大的好处。
首先来回顾一下我们的网络结结构：输入层（D），全连接层-ReLu(H)，softmax(C)。网络输入 X[N×D]，groundtruth y[N×1]
网络参数： W1[D×H],b1[1×H],W2[H×C],b2[1×C]

**Propagation:**

FC1_out=X·W1+b1−−−(1)
H_out=maximum(0,FC1_out)−−−(2)
FC2_out=H_out·W2+b2−−−(3)
final_output=softmax(FC2_out)−−−(4)

**Backpropogation**

∂L∂FC2_out=final_output[N×C]−MaskMat[N×C]−−−（5)
MaskMat参见这里
∂L∂W2=∂FC2_out∂W2∂L∂FC2_out=H_outT·∂L∂FC2_out−−−(6)
∂L∂b2=∂FC2_out∂b2∂L∂FC2_out=[1...1][1×H]·∂L∂FC2_out−−−(7)
∂L∂H_out=∂L∂FC2_out∂FC2_out∂H_out=∂L∂FC2_outWT2,∂L∂H_out=maxmium(∂L∂H_out,0)−−−(8)
∂L∂W1=∂H_cout∂W1·∂L∂H_out=XT·∂L∂H_out−−−(9)
∂L∂b1=∂H_cout∂b1·∂L∂H_out=[1...1][1×N]·∂L∂H_out−−−(10)

```
# neural_net.py

import numpy as np
import matplotlib.pyplot as plt


class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
```

```
N, a hidden layer dimension of H, and performs classification over C classes.
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.
"""

def __init__(self, input_size, hidden_size, output_size, std=1e-4):
  """
  Initialize the model. Weights are initialized to small random values and
  biases are initialized to zero. Weights and biases are stored in the
  variable self.params, which is a dictionary with the following keys:

  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,)

  Inputs:
  - input_size: The dimension D of the input data.
  - hidden_size: The number of neurons H in the hidden layer.
  - output_size: The number of classes C.
  """
  self.params = {}
  self.params['W1'] = std * np.random.randn(input_size, hidden_size)
  self.params['b1'] = np.zeros(hidden_size)
  self.params['W2'] = std * np.random.randn(hidden_size, output_size)
  self.params['b2'] = np.zeros(output_size)

def loss(self, X, y=None, reg=0.0):
  """
  Compute the loss and gradients for a two layer fully connected neural
  network.

  Inputs:
  - X: Input data of shape (N, D). Each X[i] is a training sample.
  - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
    an integer in the range 0 <= y[i] < C. This parameter is optional; if it
    is not passed then we only return scores, and if it is passed then we
    instead return the loss and gradients.
  - reg: Regularization strength.

  Returns:
  If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
  the score for class c on input X[i].

  If y is not None, instead return a tuple of:
  - loss: Loss (data loss and regularization loss) for this batch of training
    samples.
  - grads: Dictionary mapping parameter names to gradients of those parameters
    with respect to the loss function; has the same keys as self.params.
```

```
"""
# Unpack variables from the params dictionary
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']
N, D = X.shape

# Compute the forward pass
scores = None
#############################################################################
# TODO: Perform the forward pass, computing the class scores for the input. #
# Store the result in the scores variable, which should be an array of      #
# shape (N, C).                                                             #
#############################################################################

# evaluate class scores, [N x K]
hidden_layer = np.maximum(0, np.dot(X,W1)+b1) # ReLU activation
scores = np.dot(hidden_layer, W2)+b2

#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################

# If the targets are not given then jump out, we're done
if y is None:
  return scores

# Compute the loss
loss = None
#############################################################################
# TODO: Finish the forward pass, and compute the loss. This should include  #
# both the data loss and L2 regularization for W1 and W2. Store the result  #
# in the variable loss, which should be a scalar. Use the Softmax           #
# classifier loss. So that your results match ours, multiply the            #
# regularization loss by 0.5                                                #
#############################################################################

# compute the class probabilities
#scores -= np.max(scores, axis = 1)[:, np.newaxis]
#exp_scores = np.exp(scores)
exp_scores = np.exp(scores-np.max(scores, axis=1, keepdims=True))
probs = exp_scores/np.sum(exp_scores, axis=1, keepdims=True) #[N X C]

correct_logprobs = -np.log(probs[range(N),y])
data_loss = np.sum(correct_logprobs)/N
reg_loss = 0.5 * reg * ( np.sum(W1*W1) + np.sum(W2*W2) )
loss = data_loss + reg_loss

#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################

# Backward pass: compute gradients
grads = {}
#############################################################################
# TODO: Compute the backward pass, computing the derivatives of the weights #
# and biases. Store the results in the grads dictionary. For example,       #
```

```
    # grads['W1'] should store the gradient on W1, and be a matrix of same size #
    ############################################################################

    # compute the gradient on scores
    dscores = probs
    dscores[range(N),y] -= 1
    dscores /= N

    # backpropate the gradient to the parameters
    # first backprop into parameters W2 and b2
    dW2 = np.dot(hidden_layer.T, dscores)
    db2 = np.sum(dscores, axis=0, keepdims=False)
    # next backprop into hidden layer
    dhidden = np.dot(dscores, W2.T)
    # backprop the ReLU non-linearity
    dhidden[hidden_layer <= 0] = 0
    # finally into W,b
    dW1 = np.dot(X.T, dhidden)
    db1 = np.sum(dhidden, axis=0, keepdims=False)

    # add regularization gradient contribution
    dW2 += reg * W2
    dW1 += reg * W1

    grads['W1'] = dW1
    grads['W2'] = dW2
    grads['b1'] = db1
    grads['b2'] = db2
    #print dW1.shape, dW2.shape, db1.shape, db2.shape
    ############################################################################
    #                              END OF YOUR CODE                            #
    ############################################################################

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """
```

```
num_train = X.shape[0]
iterations_per_epoch = max(num_train / batch_size, 1)

# Use SGD to optimize the parameters in self.model
loss_history = []
train_acc_history = []
val_acc_history = []

for it in xrange(num_iters):
  X_batch = None
  y_batch = None

  #########################################################################
  # TODO: Create a random minibatch of training data and labels, storing  #
  # them in X_batch and y_batch respectively.                             #
  #########################################################################
  sample_index = np.random.choice(num_train, batch_size, replace=True)
  X_batch = X[sample_index, :]
  y_batch = y[sample_index]

  #########################################################################
  #                             END OF YOUR CODE                          #
  #########################################################################

  # Compute loss and gradients using the current minibatch
  loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
  loss_history.append(loss)

  #########################################################################
  # TODO: Use the gradients in the grads dictionary to update the         #
  # parameters of the network (stored in the dictionary self.params)      #
  # using stochastic gradient descent. You'll need to use the gradients   #
  # stored in the grads dictionary defined above.                         #
  #########################################################################
  dW1 = grads['W1']
  dW2 = grads['W2']
  db1 = grads['b1']
  db2 = grads['b2']
  self.params['W1'] -= learning_rate*dW1
  self.params['W2'] -= learning_rate*dW2
  self.params['b1'] -= learning_rate*db1
  self.params['b2'] -= learning_rate*db2

  #########################################################################
  #                             END OF YOUR CODE                          #
  #########################################################################

  if verbose and it % 100 == 0:
    print 'iteration %d / %d: loss %f' % (it, num_iters, loss)

  # Every epoch, check train and val accuracy and decay learning rate.
  if it % iterations_per_epoch == 0:
    # Check accuracy
    train_acc = (self.predict(X_batch) == y_batch).mean()
    val_acc = (self.predict(X_val) == y_val).mean()
    train_acc_history.append(train_acc)
```

```
      val_acc_history.append(val_acc)

      # Decay learning rate
      learning_rate *= learning_rate_decay

  return {
    'loss_history': loss_history,
    'train_acc_history': train_acc_history,
    'val_acc_history': val_acc_history,
  }

def predict(self, X):
  """
  Use the trained weights of this two-layer network to predict labels for
  data points. For each data point we predict scores for each of the C
  classes, and assign each data point to the class with the highest score.

  Inputs:
  - X: A numpy array of shape (N, D) giving N D-dimensional data points to
    classify.

  Returns:
  - y_pred: A numpy array of shape (N,) giving predicted labels for each of
    the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
    to have class c, where 0 <= c < C.
  """
  y_pred = None

  ###########################################################################
  # TODO: Implement this function; it should be VERY simple!                #
  ###########################################################################
  hidden_lay = np.maximum(0, np.dot(X,self.params['W1'])+self.params['b1'])
  y_pred = np.argmax( np.dot(hidden_lay, self.params['W2']), axis=1)

  ###########################################################################
  #                              END OF YOUR CODE                           #
  ###########################################################################

  return y_pred
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70

- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100
- 101
- 102
- 103
- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111
- 112
- 113
- 114
- 115
- 116
- 117
- 118
- 119
- 120
- 121
- 122
- 123
- 124
- 125
- 126

- 127
- 128
- 129
- 130
- 131
- 132
- 133
- 134
- 135
- 136
- 137
- 138
- 139
- 140
- 141
- 142
- 143
- 144
- 145
- 146
- 147
- 148
- 149
- 150
- 151
- 152
- 153
- 154
- 155
- 156
- 157
- 158
- 159
- 160
- 161
- 162
- 163
- 164
- 165
- 166
- 167
- 168
- 169
- 170
- 171
- 172
- 173
- 174
- 175
- 176
- 177
- 178
- 179
- 180
- 181
- 182

- 183
- 184
- 185
- 186
- 187
- 188
- 189
- 190
- 191
- 192
- 193
- 194
- 195
- 196
- 197
- 198
- 199
- 200
- 201
- 202
- 203
- 204
- 205
- 206
- 207
- 208
- 209
- 210
- 211
- 212
- 213
- 214
- 215
- 216
- 217
- 218
- 219
- 220
- 221
- 222
- 223
- 224
- 225
- 226
- 227
- 228
- 229
- 230
- 231
- 232
- 233
- 234
- 235
- 236
- 237
- 238

- 239
- 240
- 241
- 242
- 243
- 244
- 245
- 246
- 247
- 248
- 249
- 250
- 251
- 252
- 253
- 254
- 255
- 256
- 257
- 258
- 259
- 260
- 261
- 262
- 263
- 264
- 265
- 266
- 267
- 268
- 269
- 270
- 271
- 272
- 273
- 274
- 275

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18

- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74

- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100
- 101
- 102
- 103
- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111
- 112
- 113
- 114
- 115
- 116
- 117
- 118
- 119
- 120
- 121
- 122
- 123
- 124
- 125
- 126
- 127
- 128
- 129
- 130

- 131
- 132
- 133
- 134
- 135
- 136
- 137
- 138
- 139
- 140
- 141
- 142
- 143
- 144
- 145
- 146
- 147
- 148
- 149
- 150
- 151
- 152
- 153
- 154
- 155
- 156
- 157
- 158
- 159
- 160
- 161
- 162
- 163
- 164
- 165
- 166
- 167
- 168
- 169
- 170
- 171
- 172
- 173
- 174
- 175
- 176
- 177
- 178
- 179
- 180
- 181
- 182
- 183
- 184
- 185
- 186

- 187
- 188
- 189
- 190
- 191
- 192
- 193
- 194
- 195
- 196
- 197
- 198
- 199
- 200
- 201
- 202
- 203
- 204
- 205
- 206
- 207
- 208
- 209
- 210
- 211
- 212
- 213
- 214
- 215
- 216
- 217
- 218
- 219
- 220
- 221
- 222
- 223
- 224
- 225
- 226
- 227
- 228
- 229
- 230
- 231
- 232
- 233
- 234
- 235
- 236
- 237
- 238
- 239
- 240
- 241
- 242

- 243
- 244
- 245
- 246
- 247
- 248
- 249
- 250
- 251
- 252
- 253
- 254
- 255
- 256
- 257
- 258
- 259
- 260
- 261
- 262
- 263
- 264
- 265
- 266
- 267
- 268
- 269
- 270
- 271
- 272
- 273
- 274
- 275

**Tune your hyperparameters**

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

**Tuning**. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

**Approximate results**. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

**Experiment**: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 52% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

```
# two_layer_net.ipynb

best_net = None # store the best model into this
best_stats = None
################################################################################
# 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.                          #
################################################################################
input_size = 32 * 32 * 3
hidden_size = 300
num_classes = 10

results = {}
best_val = -1
learning_rates = [1e-3, 1.2e-3, 1.4e-3, 1.6e-3, 1.8e-3]
regularization_strengths = [1e-4, 1e-3, 1e-2]

params = [(x,y) for x in learning_rates for y in regularization_strengths ]
for lrate, regular in params:
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    # Train the network
    stats = net.train(X_train, y_train, X_val, y_val,
                      num_iters=1600, batch_size=400,
                      learning_rate=lrate, learning_rate_decay=0.90,
                      reg=regular, verbose=False)

    # Predict on the validation set
    accuracy_train = (net.predict(X_train) == y_train).mean()
    accuracy_val = (net.predict(X_val) == y_val).mean()
    results[(lrate, regular)] = (accuracy_train, accuracy_val)
    if( best_val < accuracy_val ):
        best_val = accuracy_val
        best_net = net
        best_stats = stats
```

```
# 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

# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(best_stats['train_acc_history'], label='train',color='r')
plt.plot(best_stats['val_acc_history'], label='val',color='g')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.show()


################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28

- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69

- 70

lr 1.000000e-03 reg 1.000000e-04 train accuracy: 0.541551 val accuracy: 0.499000
lr 1.000000e-03 reg 1.000000e-03 train accuracy: 0.541694 val accuracy: 0.511000
lr 1.000000e-03 reg 1.000000e-02 train accuracy: 0.540898 val accuracy: 0.490000
lr 1.200000e-03 reg 1.000000e-04 train accuracy: 0.562041 val accuracy: 0.528000
lr 1.200000e-03 reg 1.000000e-03 train accuracy: 0.563653 val accuracy: 0.507000
lr 1.200000e-03 reg 1.000000e-02 train accuracy: 0.564184 val accuracy: 0.512000
lr 1.400000e-03 reg 1.000000e-04 train accuracy: 0.580857 val accuracy: 0.532000
lr 1.400000e-03 reg 1.000000e-03 train accuracy: 0.580857 val accuracy: 0.513000
lr 1.400000e-03 reg 1.000000e-02 train accuracy: 0.575245 val accuracy: 0.534000
lr 1.600000e-03 reg 1.000000e-04 train accuracy: 0.593347 val accuracy: 0.529000
lr 1.600000e-03 reg 1.000000e-03 train accuracy: 0.594857 val accuracy: 0.548000
lr 1.600000e-03 reg 1.000000e-02 train accuracy: 0.593878 val accuracy: 0.551000
lr 1.800000e-03 reg 1.000000e-04 train accuracy: 0.605306 val accuracy: 0.537000
lr 1.800000e-03 reg 1.000000e-03 train accuracy: 0.610000 val accuracy: 0.533000
lr 1.800000e-03 reg 1.000000e-02 train accuracy: 0.603204 val accuracy: 0.546000
best validation accuracy achieved during cross-validation: 0.551000
Test accuracy: 0.542