Tsinghua University
机器学习基础
Fall 2017

第 4 次作业

赵丰 2017310711 January 4, 2018

- **Acknowledgments:** This coursework refers to textbook.
- **Collaborators:**
  - DIY

I use `enumerate` to generate answers for each question:

1. 1.1.

Algorithm 1 modifed ADABOOST($S = ((x_1, y_1), \dots, (x_m, y_m))$)

for $i = 1$ to m **do**
  $D_1(i) \leftarrow \frac{1}{m}$
end for
for $t = 1$ to T **do**
  $h_t \leftarrow$ base classifier in H with small error $\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$
  $Z_t \leftarrow (1-\epsilon_t)e^{-\alpha} + \epsilon_t e^{\alpha}$
  for $i = 1$ to m **do**
    $D_{t+1}(i) \leftarrow \frac{D_t(i)exp(-\alpha y_i h_t(x_i))}{Z_t}$
  end for
end for
$g \leftarrow \sum_{t=1}^{T} h_t$
return $h = \text{sgn}(g)$

设常数 $\alpha = \frac{1}{2}\log\frac{1+2\gamma}{1-2\gamma}$，原 adaboost 算法按上表修改：相应的证明过程中首先有：

$$D_{t+1}(i) = \frac{e^{-y_i g_t(i)\alpha}}{m\prod_{s=1}^{t} Z_s} \tag{1}$$

其中 $g_t = \sum_{s=1}^{t} h_s$ 注意到算法中 Z_t 的取值是使得 $\sum_{i=1}^{m} D_{t+1}(i) = 1$, 所以利用 $1_{u\leq 0} \leq exp(-u\alpha)$ 得训练误差

$$\hat{R}(h) \leq \prod_{t=1}^{T} Z_t \tag{2}$$

注意到 Z_t 是关于 ϵ_t 的增函数，根据题目中 $\epsilon_t \leq \frac{1}{2} - \gamma$ 得到

$$Z_t \leq (\frac{1}{2}+\gamma)e^{-\alpha} + (\frac{1}{2}-\gamma)e^{\alpha} \tag{3}$$
$$= \sqrt{1-4\gamma^2} \tag{4}$$

所以有

$$\hat{R}(h) \leq (1-4\gamma^2)^{\frac{T}{2}} \tag{5}$$

1.2. (a) 设已经有 $\boldsymbol{\alpha}_{t-1} = (\alpha_1, \ldots, \alpha_{t-1}, 0, \ldots, 0)$

$$\tag{6}$$

设 ϵ_t

$$\frac{dL(\boldsymbol{\alpha_{t-1}} + \eta \boldsymbol{e}_t)}{d\eta}|_{\eta=0} = \sum_{i=1}^{m} \Phi'(-y_i \sum_{j=1}^{t-1} \alpha_j h_j(x_i))(-y_i h_t(x_i)) \tag{7}$$

记

$$D_t(i) = \frac{\Phi'(-y_i \sum_{j=1}^{t-1} \alpha_j h_j(x_i))}{\sum_{i=1}^{m} \Phi'(-y_i \sum_{j=1}^{t-1} \alpha_j h_j(x_i))} \tag{8}$$

满足 $\sum_{i=1}^{m} D_t(i) = 1$。
于是有：

$$\frac{dL(\boldsymbol{\alpha_{t-1}} + \eta \boldsymbol{e}_t)}{d\eta}|_{\eta=0} = -\sum_{i=1}^{m} D_t(i) y_i h_t(x_i) [\sum_{i=1}^{m} \Phi'(-y_i \sum_{j=1}^{s-1} \alpha_j h_j(x_i))] \tag{9}$$

假设 $h_s (s \leq t-1)$ 已经取好，设 $\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$ 则将上式可化为

$$\frac{dL(\boldsymbol{\alpha_{t-1}} + \eta \boldsymbol{e}_t)}{d\eta}|_{\eta=0} = -(2\epsilon_t - 1) [\sum_{i=1}^{m} \Phi'(-y_i \sum_{j=1}^{s-1} \alpha_j h_j(x_i))] \tag{10}$$

为使 L 在 $\boldsymbol{\alpha_{t-1}}$ 处的方向导数最小，取

$$h_t = \operatorname*{arg\,min}_{h \in H} \Pr_{i \sim D_t} [h_t(x_i) \neq y_i] \tag{11}$$

为确定步长 η，需求解 $\frac{dL(\boldsymbol{\alpha_{t-1}} + \eta \boldsymbol{e}_t)}{d\eta} = 0$，原 Adaboost 算法修改如表 2所示。

(b) (1,3) 不连续，(2) 不单调，只有 (4) 满足关于 Φ 的假设。

(c) 针对 $\Phi(u) = \log(1 + e^u)$, 求解 α_t 的非线性方程具有如下的形式

$$\sum_{i=1}^{m} \frac{y_i h_t(x_i)}{1 + \exp(-y_i g_t(x_i) - y_i h_t(x_i) \eta)} = 0 \tag{12}$$

上式可数值求解。

2. 对于 toy data，我们作出损失函数随迭代次数变化如图 1所示。

   针对数据，使用题目中的超参数我们得到约 0.29 的准确率，作出损失函数和训练、验证准确率随迭代次数变化的规律如图 2所示。

   对于两层全连接的神经网络，我们针对隐层数量，learning_rate, 正则化系数，迭代次数共四个超参数进行优化，我们采用对四个参数进行离散，在离散后的四维空间进行网格搜索的方法，最终优化得到最佳的参数如表 1所示。

   对于我们找到的最优参数，网络第一层权系数可视化的结果如图 3所示。

   寻找最佳参数的 python 代码如 hyperparameter_tuning.py 所示。

Algorithm 2 一般代价函数 Adaboost($S = ((x_1, y_1), \dots, (x_m, y_m))$)

for $i = 1$ to m **do**
    $D_1(i) \leftarrow \frac{1}{m}$
end for
$g_0 \leftarrow 0$
for $t = 1$ to T **do**
    $h_t \leftarrow$ base classifier in H with small error $\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$
    $Z_t \leftarrow (1 - \epsilon_t)e^{-\alpha} + \epsilon_t e^{\alpha}$
    求解关于 η 的非线性方程 $\frac{dL(\boldsymbol{\alpha}_{t-1} + \eta \boldsymbol{e}_t)}{d\eta} = 0$ 得解 α_t
    $g_t \leftarrow g_{t-1} + \alpha_t h_t$
    for $i = 1$ to m **do**
        $D_{t+1}(i) \leftarrow \frac{\Phi'(-y_i g_t(x_i))}{\sum_{i=1}^{m} \Phi'(-y_i g_t(x_i))}$
    end for
end for
$g \leftarrow \sum_{t=1}^{T} \alpha_t h_t$
return $h = \mathrm{sgn}(g)$

Figure 1

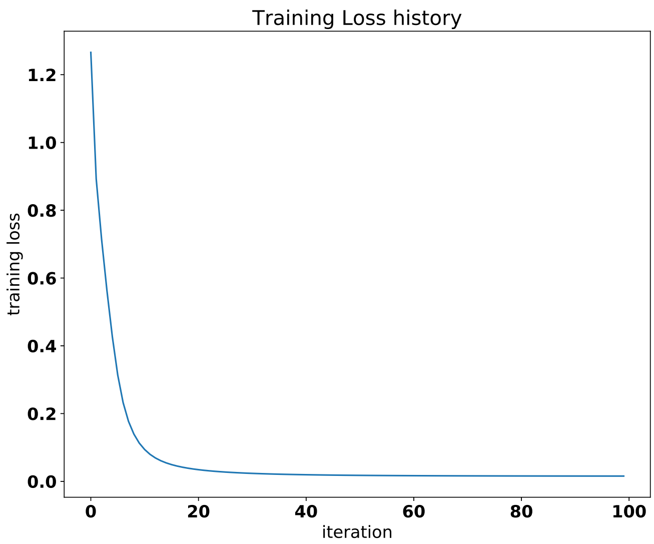


Table 1

隐层数量	70
learning_rate	0.0025
正则化系数	0.9
迭代次数	2000
Validation accuracy	0.491
Test accuracy	0.488

Figure 2

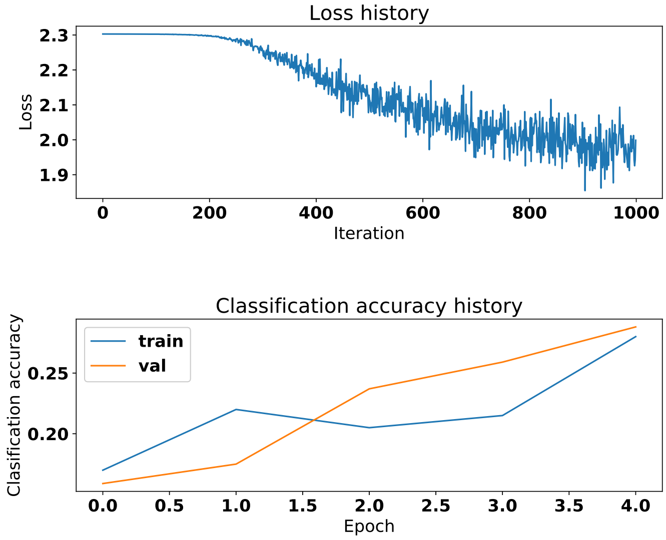
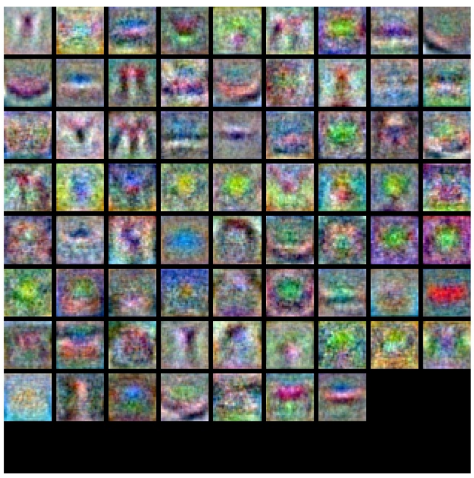


Figure 3



a4/hyperparameter_tuning.py

```
from cs231n.classifiers.neural_net import TwoLayerNet
import numpy as np
input_size = 32 * 32 * 3
num_classes = 10
```

```python

import logging

logging.basicConfig(level=logging.INFO,
                format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',
                datefmt='%a, %d %b %Y %H:%M:%S',
                filename='ht.log',
                filemode='w')

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

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


# hyperparameter:
```

```python
# hidden layer size -- increasing h_l_s will
# decrease the training error but there is over-fitting
    possibility.
# num_iters
# these two hyperparameters will increase computational cost
# learing_rate
# regularation parameter
# use grid search to tune the parameter
learning_rate_list=[1e-4,5*1e-4,25*1e-4,125*1e-4,625*1e-4]
reg_list=[0.05,0.1,0.5,0.9,1.5]
hidden_size_list=[50,70,90]
num_iteration_increment_list=[500,500,1000]
best_val_acc=0
logging.info("*****begin of hyperparameter tuning*****")
for i in learning_rate_list:
   logging.info("for learning_rate=%.4f tuning "%i)
   for j in reg_list:
      for h in hidden_size_list:
         net = TwoLayerNet(input_size, h, num_classes)
         iteration_total=0
         for k in num_iteration_increment_list:
            net.train(X_train, y_train, X_val, y_val,
               num_iters=k, batch_size=200,
               learning_rate=i, learning_rate_decay=0.95,
               reg=j, verbose=False)
            val_acc = (net.predict(X_val) == y_val).mean()
            iteration_total += k
            if(val_acc>best_val_acc):
               best_val_acc=val_acc
               logging.info("best net: acc=%.3f,for
                   learning_rate=%.4f,\
               reg=%.3f,num_iters=%d,hidden_size=%d"%(best_val_acc,i,j,iteration_total,h))
logging.info("*****end of hyperparameter tuning*****")
```

补全指定函数后的 neutral_net.py 文件如下。

a4/cs231n/classifiers/neural_net.py

```python
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.

```

```python
    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 _relu(self,X):
        """
        Implementation of RELU function for 1D numpy array

        Inputs:
        - X: 1D numpy array

        Returns:
        Y: 1D numpy array
        """

       # y = list(map(lambda x: max(x, 0), X))
        return np.maximum(X,0)

    def _softmax(self,X):
        """
        Implementation of softmax function for 1D numpy array
```

```

    Inputs:
    - X: 1D numpy array

    Returns:
    Y: 1D numpy array
    """
    tmp=np.exp(X)
    return tmp/sum(tmp)

  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 = np.zeros([N,len(b2)])
    #############################################################################
    # 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).
```

```
                                                #
    #############################################################################
    # python的numpy array运算有个broadcast功能，它会自动扩维。不放心的话，你可以查一下broadcast机制
    first_layer_output=np.dot(X,W1)+b1
    activated_result=self._relu(first_layer_output)
    second_layer_output=np.dot(activated_result,W2)+b2
    #softmax_output=self._softmax(second_layer_output)
    scores=second_layer_output
    #############################################################################
    #                              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

    #############################################################################
    # 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                                                #
    #############################################################################
    #for back propogation purpose we need save more
    forward_g=np.exp(scores)
    forward_h=np.sum(forward_g,axis=1)
    L_loss=-1*scores[list(range(len(y))),y]+np.log(forward_h)
    cross_entropy_loss=np.average(L_loss)
    L2_regularization_loss=0.5*reg*(np.sum(W1*W1)+np.sum(W2*W2))
    loss = cross_entropy_loss + L2_regularization_loss
    #############################################################################
    #                              END OF YOUR CODE                             #
    #############################################################################

    # Backward pass: compute gradients
    grads = {}
    backward_score=np.zeros(scores.shape)
    backward_score[list(range(len(y))),y]=-1
    backward_score+=forward_g/np.transpose(forward_h+np.zeros([len(b2),1]));
    #############################################################################
    # 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 #
    #############################################################################
    # average over columns
    grads['b2'] = np.average(backward_score,axis=0)
    grads['W2'] = np.dot(activated_result.T,backward_score)/N
    grads['W2'] +=reg*W2 # plus regularization part


    backward_activated_result=np.dot(backward_score,W2.T)
    backward_first_layer_output=backward_activated_result*(first_layer_output>0)
    grads['b1']=np.average(backward_first_layer_output,axis=0)
    grads['W1']=np.dot(X.T,backward_first_layer_output)/N
    grads['W1'] +=reg*W1 # plus regularization part

    #############################################################################
    #                              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 = []
```

```python
    t_batch_size=min(num_train,batch_size)
    for it in range(num_iters):
      index_choiced=np.random.choice(num_train,t_batch_size,False)
      X_batch = X[index_choiced,:] # for bebugging purpose only !
      y_batch = y[index_choiced]

      #########################################################################
      # TODO: Create a random minibatch of training data and labels, storing #
      # them in X_batch and y_batch respectively.                             #
      #########################################################################
      # the mini-batch contains only a single example.
      # This process is called Stochastic Gradient Descent (SGD) (or also sometimes on-line gradient descent)
      pass
      #########################################################################
      #                             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.                         #
      #########################################################################
      self.params['W1'] -= learning_rate*grads['W1']
      self.params['W2'] -= learning_rate*grads['W2']
      self.params['b1'] -= learning_rate*grads['b1']
      self.params['b2'] -= learning_rate*grads['b2']

      #########################################################################
      #                             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()
```

```python
      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.
    """
    first_layer_output=np.dot(X,self.params['W1'])+self.params['b1']
    activated_result=self._relu(first_layer_output)
    second_layer_output=np.dot(activated_result,self.params['W2'])+self.params['b2']
    softmax_output=self._softmax(second_layer_output)
    y_pred = np.argmax(softmax_output,axis=1)

    ###########################################################################
    # TODO: Implement this function; it should be VERY simple!                #
    ###########################################################################
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################

    return y_pred
```