

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:

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

```
1. 1.1. 1: for  $i = 1$  to  $m$  do
2:    $D_1(i) \leftarrow \frac{1}{m}$ 
3: end for
4: for  $t = 1$  to  $T$  do
5:    $h_t \leftarrow$  base classifier in  $H$  with small error  $\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$ 
6:    $Z_t \leftarrow (1 - \epsilon_t)e^{-\alpha} + \epsilon_t e^{\alpha}$ 
7:   for  $i = 1$  to  $m$  do
8:      $D_{t+1}(i) \leftarrow \frac{D_t(i) \exp(-\alpha y_i h_t(x_i))}{Z_t}$ 
9:   end for
10: end for
11:  $g \leftarrow \sum_{t=1}^T h_t$ 
12: 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:

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

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

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

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

$$Z_t \leq \left(\frac{1}{2} + \gamma\right)e^{-\alpha} + \left(\frac{1}{2} - \gamma\right)e^{\alpha} \quad (3)$$

$$= \sqrt{1 - 4\gamma^2} \quad (4)$$

所以有

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

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

(6)

设  $\epsilon_t$

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

记

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

满足  $\sum_{i=1}^m D_t(i) = 1$ 。

于是有：

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

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

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

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

$$h_t = \arg \min_{h \in H} \Pr_{i \sim D_t} [h_t(x_i) \neq y_i] \quad (11)$$

为确定步长  $\eta$ ，需求解  $\frac{dL(\alpha_{t-1} + \eta e_t)}{d\eta} = 0$ ，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  $\Phi$  的假设。

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

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

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( $S = ((x_1, y_1), \dots, (x_m, y_m))$ )

---

```

1: for  $i = 1$  to  $m$  do
2:    $D_1(i) \leftarrow \frac{1}{m}$ 
3: end for
4:  $g_0 \leftarrow 0$ 
5: for  $t = 1$  to  $T$  do
6:    $h_t \leftarrow$  base classifier in  $H$  with small error  $\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$ 
7:    $Z_t \leftarrow (1 - \epsilon_t)e^{-\alpha} + \epsilon_t e^{\alpha}$ 
8:   求解关于  $\eta$  的非线性方程  $\frac{dL(\alpha_{t-1} + \eta e_t)}{d\eta} = 0$  得解  $\alpha_t$ 
9:    $g_t \leftarrow g_{t-1} + \alpha_t h_t$ 
10:  for  $i = 1$  to  $m$  do
11:     $D_{t+1}(i) \leftarrow \frac{\Phi'(-y_i g_t(x_i))}{\sum_{i=1}^m \Phi'(-y_i g_t(x_i))}$ 
12:  end for
13: end for
14:  $g \leftarrow \sum_{t=1}^T \alpha_t h_t$ 
15: return  $h = \text{sgn}(g)$ 

```

---

Figure 1

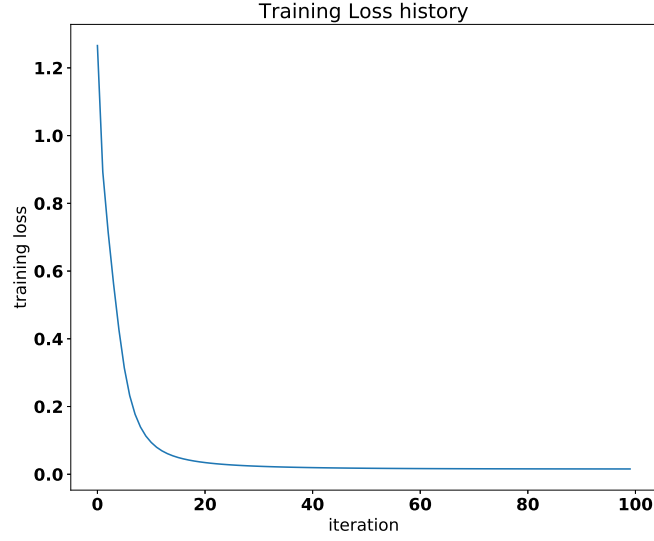


Table 1

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

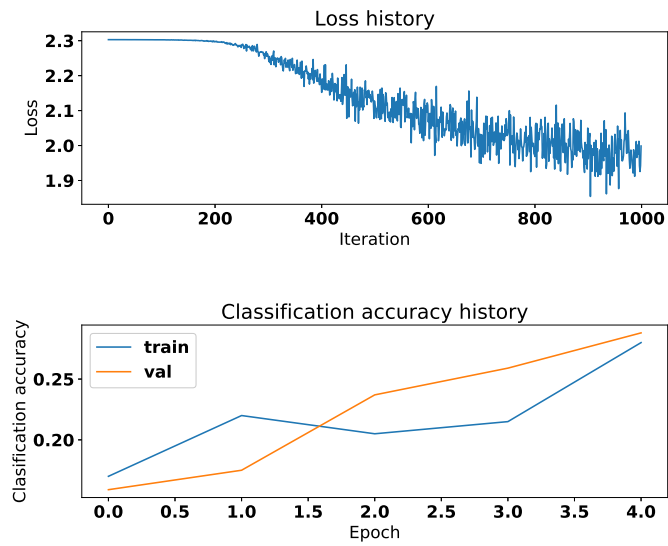


Figure 3



```
a4/hyperparameter_tuning.py
1 from cs231n.classifiers.neural_net import TwoLayerNet
2 import numpy as np
3 input_size = 32 * 32 * 3
4 num_classes = 10
```

```

5
6 import logging
7
8 logging.basicConfig(level=logging.INFO,
9                     format='%(asctime)s %(filename)s[line:%(lineno)d]
10                          %(levelname)s %(message)s',
11                     datefmt='%a, %d %b %Y %H:%M:%S',
12                     filename='ht.log',
13                     filemode='w')
14
15 from cs231n.data_utils import load_CIFAR10
16 def get_CIFAR10_data(num_training=49000, num_validation=1000,
17                     num_test=1000):
18     """
19     Load the CIFAR-10 dataset from disk and perform preprocessing
20     to prepare
21     it for the two-layer neural net classifier. These are the same
22     steps as
23     we used for the SVM, but condensed to a single function.
24     """
25     # Load the raw CIFAR-10 data
26     cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
27     X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
28
29     # Subsample the data
30     mask = range(num_training, num_training + num_validation)
31     X_val = X_train[mask]
32     y_val = y_train[mask]
33     mask = range(num_training)
34     X_train = X_train[mask]
35     y_train = y_train[mask]
36     mask = range(num_test)
37     X_test = X_test[mask]
38     y_test = y_test[mask]
39
40     # Normalize the data: subtract the mean image
41     mean_image = np.mean(X_train, axis=0)
42     X_train -= mean_image
43     X_val -= mean_image
44     X_test -= mean_image
45
46     # Reshape data to rows
47     X_train = X_train.reshape(num_training, -1)
48     X_val = X_val.reshape(num_validation, -1)
49     X_test = X_test.reshape(num_test, -1)
50
51     return X_train, y_train, X_val, y_val, X_test, y_test
52
53 # Invoke the above function to get our data.
54 X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
55
56 # hyperparameter:

```

```

55 # hidden layer size -- increasing h_l_s will
56 # decrease the training error but there is over-fitting
    possibility.
57 # num_iters
58 # these two hyperparameters will increase computational cost
59 # learning_rate
60 # regularization parameter
61 # use grid search to tune the parameter
62 learning_rate_list=[1e-4,5*1e-4,25*1e-4,125*1e-4,625*1e-4]
63 reg_list=[0.05,0.1,0.5,0.9,1.5]
64 hidden_size_list=[50,70,90]
65 num_iteration_increment_list=[500,500,1000]
66 best_val_acc=0
67 logging.info("*****begin of hyperparameter tuning*****")
68 for i in learning_rate_list:
69     logging.info("for learning_rate=%.4f tuning "%i)
70     for j in reg_list:
71         for h in hidden_size_list:
72             net = TwoLayerNet(input_size, h, num_classes)
73             iteration_total=0
74             for k in num_iteration_increment_list:
75                 net.train(X_train, y_train, X_val, y_val,
76                           num_iters=k, batch_size=200,
77                           learning_rate=i, learning_rate_decay=0.95,
78                           reg=j, verbose=False)
79                 val_acc = (net.predict(X_val) == y_val).mean()
80                 iteration_total += k
81                 if(val_acc>best_val_acc):
82                     best_val_acc=val_acc
83                     logging.info("best net: acc=%.3f,for
                        learning_rate=%.4f,\
84                               reg=%.3f,num_iters=%d,hidden_size=%d"%(best_val_acc,i,j,iteration_total,h))
85 logging.info("*****end of hyperparameter tuning*****")

```

补全指定函数后的 neutral\_net.py 文件如下。

a4/cs231n/classifiers/neutral\_net.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 class TwoLayerNet(object):
6     """
7     A two-layer fully-connected neural network. The net has an input
        dimension of
8     N, a hidden layer dimension of H, and performs classification
        over C classes.
9     We train the network with a softmax loss function and L2
        regularization on the
10    weight matrices. The network uses a ReLU nonlinearity after the
        first fully
11    connected layer.
12

```

```

13 In other words, the network has the following architecture:
14
15 input - fully connected layer - ReLU - fully connected layer -
    softmax
16
17 The outputs of the second fully-connected layer are the scores
    for each class.
18 """
19
20 def __init__(self, input_size, hidden_size, output_size,
    std=1e-4):
21     """
22     Initialize the model. Weights are initialized to small random
        values and
23     biases are initialized to zero. Weights and biases are stored
        in the
24     variable self.params, which is a dictionary with the following
        keys:
25
26     W1: First layer weights; has shape (D, H)
27     b1: First layer biases; has shape (H,)
28     W2: Second layer weights; has shape (H, C)
29     b2: Second layer biases; has shape (C,)
30
31     Inputs:
32     - input_size: The dimension D of the input data.
33     - hidden_size: The number of neurons H in the hidden layer.
34     - output_size: The number of classes C.
35     """
36     self.params = {}
37     self.params['W1'] = std * np.random.randn(input_size,
        hidden_size)
38     self.params['b1'] = np.zeros(hidden_size)
39     self.params['W2'] = std * np.random.randn(hidden_size,
        output_size)
40     self.params['b2'] = np.zeros(output_size)
41
42 def _relu(self,X):
43     """
44     Implementation of RELU function for 1D numpy array
45
46     Inputs:
47     - X: 1D numpy array
48
49     Returns:
50     Y: 1D numpy array
51     """
52
53     # y = list(map(lambda x: max(x, 0), X))
54     return np.maximum(X,0)
55
56 def _softmax(self,X):
57     """
58     Implementation of softmax function for 1D numpy array

```

```

59     Inputs:
60     - X: 1D numpy array
61
62     Returns:
63     Y: 1D numpy array
64     """
65     tmp=np.exp(X)
66     return tmp/sum(tmp)
67
68
69 def loss(self, X, y=None, reg=0.0):
70     """
71     Compute the loss and gradients for a two layer fully connected
72     neural
73     network.
74
75     Inputs:
76     - X: Input data of shape (N, D). Each X[i] is a training sample.
77     - y: Vector of training labels. y[i] is the label for X[i], and
78         each y[i] is
79         an integer in the range 0 <= y[i] < C. This parameter is
80         optional; if it
81         is not passed then we only return scores, and if it is passed
82         then we
83         instead return the loss and gradients.
84     - reg: Regularization strength.
85
86     Returns:
87     If y is None, return a matrix scores of shape (N, C) where
88         scores[i, c] is
89         the score for class c on input X[i].
90
91     If y is not None, instead return a tuple of:
92     - loss: Loss (data loss and regularization loss) for this batch
93         of training
94         samples.
95     - grads: Dictionary mapping parameter names to gradients of
96         those parameters
97         with respect to the loss function; has the same keys as
98         self.params.
99     """
100     # Unpack variables from the params dictionary
101     W1, b1 = self.params['W1'], self.params['b1']
102     W2, b2 = self.params['W2'], self.params['b2']
103     N, D = X.shape
104
105     # Compute the forward pass
106     scores = np.zeros([N,len(b2)])
107     #####
108     # TODO: Perform the forward pass, computing the class scores
109     # for the input. #
110     # Store the result in the scores variable, which should be an
111     # array of #
112     # shape (N, C).

```



```

103                                     #
104 #####
105 # python的numpy
106     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
107 first_layer_output=np.dot(X,W1)+b1
108 activated_result=self._relu(first_layer_output)
109 second_layer_output=np.dot(activated_result,W2)+b2
110 #softmax_output=self._softmax(second_layer_output)
111 scores=second_layer_output
112 #####
113 #
114                                     END OF YOUR CODE
115                                     #
116 #####
117
118 # If the targets are not given then jump out, we're done
119 if y is None:
120     return scores
121
122 # Compute the loss
123
124 #####
125 # TODO: Finish the forward pass, and compute the loss. This
126     should include #
127 # both the data loss and L2 regularization for W1 and W2. Store
128     the result #
129 # in the variable loss, which should be a scalar. Use the
130     Softmax #
131 # classifier loss. So that your results match ours, multiply
132     the #
133 # regularization loss by 0.5
134
135                                     #
136 #####
137 #for back propogation purpose we need save more
138 forward_g=np.exp(scores)
139 forward_h=np.sum(forward_g,axis=1)
140 L_loss=-1*scores[list(range(len(y))),y]+np.log(forward_h)
141 cross_entropy_loss=np.average(L_loss)
142 L2_regularization_loss=0.5*reg*(np.sum(W1*W1)+np.sum(W2*W2))
143 loss = cross_entropy_loss + L2_regularization_loss
144 #####
145 #
146                                     END OF YOUR CODE
147                                     #
148 #####
149
150 # Backward pass: compute gradients
151 grads = {}
152 backward_score=np.zeros(scores.shape)
153 backward_score[list(range(len(y))),y]=-1
154 backward_score+=forward_g/np.transpose(forward_h+np.zeros([len(b2),1]));
155 #####
156 # TODO: Compute the backward pass, computing the derivatives of
157     the weights #
158 # and biases. Store the results in the grads dictionary. For
159     example, #

```

```

146     # grads['W1'] should store the gradient on W1, and be a matrix
      of same size #
147     #####
148     # average over columns
149     grads['b2'] = np.average(backward_score,axis=0)
150     grads['W2'] = np.dot(activated_result.T,backward_score)/N
151     grads['W2'] +=reg*W2 # plus regularization part
152
153
154     backward_activated_result=np.dot(backward_score,W2.T)
155     backward_first_layer_output=backward_activated_result*(first_layer_output>0)
156     grads['b1']=np.average(backward_first_layer_output,axis=0)
157     grads['W1']=np.dot(X.T,backward_first_layer_output)/N
158     grads['W1'] +=reg*W1 # plus regularization part
159
160     #####
161     #                               END OF YOUR CODE
      #
162     #####
163
164     return loss, grads
165
166 def train(self, X, y, X_val, y_val,
167         learning_rate=1e-3, learning_rate_decay=0.95,
168         reg=1e-5, num_iters=100,
169         batch_size=200, verbose=False):
170     """
171     Train this neural network using stochastic gradient descent.
172
173     Inputs:
174     - X: A numpy array of shape (N, D) giving training data.
175     - y: A numpy array f shape (N,) giving training labels; y[i] =
      c means that
176       X[i] has label c, where 0 <= c < C.
177     - X_val: A numpy array of shape (N_val, D) giving validation
      data.
178     - y_val: A numpy array of shape (N_val,) giving validation
      labels.
179     - learning_rate: Scalar giving learning rate for optimization.
180     - learning_rate_decay: Scalar giving factor used to decay the
      learning rate
181       after each epoch.
182     - reg: Scalar giving regularization strength.
183     - num_iters: Number of steps to take when optimizing.
184     - batch_size: Number of training examples to use per step.
185     - verbose: boolean; if true print progress during optimization.
186     """
187     num_train = X.shape[0]
188     iterations_per_epoch = max(num_train / batch_size, 1)
189
190     # Use SGD to optimize the parameters in self.model
191     loss_history = []
192     train_acc_history = []
193     val_acc_history = []

```

```

194 t_batch_size=min(num_train,batch_size)
195 for it in range(num_iters):
196     index_choi= np.random.choice(num_train,t_batch_size,False)
197     X_batch = X[index_choi,:] # for bebugging purpose only !
198     y_batch = y[index_choi]
199
200     #####
201     # TODO: Create a random minibatch of training data and
202     # labels, storing #
203     # them in X_batch and y_batch respectively.
204     #
205     #####
206     # the mini-batch contains only a single example.
207     # This process is called Stochastic Gradient Descent (SGD)
208     # (or also sometimes on-line gradient descent)
209     pass
210     #####
211     #
212     # END OF YOUR CODE
213     #
214     #####
215     # Compute loss and gradients using the current minibatch
216
217     loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
218     loss_history.append(loss)
219
220     #####
221     # TODO: Use the gradients in the grads dictionary to update
222     # the #
223     # parameters of the network (stored in the dictionary
224     # self.params) #
225     # using stochastic gradient descent. You'll need to use the
226     # gradients #
227     # stored in the grads dictionary defined above.
228     #
229     #####
230     self.params['W1'] -= learning_rate*grads['W1']
231     self.params['W2'] -= learning_rate*grads['W2']
232     self.params['b1'] -= learning_rate*grads['b1']
233     self.params['b2'] -= learning_rate*grads['b2']
234
235     #####
236     #
237     # END OF YOUR CODE
238     #
239     #####
240
241     if verbose and it % 100 == 0:
242         print('iteration %d / %d: loss %f' % (it, num_iters, loss))
243
244     # Every epoch, check train and val accuracy and decay
245     # learning rate.
246     if it % iterations_per_epoch == 0:
247         # Check accuracy
248         train_acc = (self.predict(X_batch) == y_batch).mean()

```

```

238     val_acc = (self.predict(X_val) == y_val).mean()
239     train_acc_history.append(train_acc)
240     val_acc_history.append(val_acc)
241
242     # Decay learning rate
243     learning_rate *= learning_rate_decay
244
245     return {
246         'loss_history': loss_history,
247         'train_acc_history': train_acc_history,
248         'val_acc_history': val_acc_history,
249     }
250
251 def predict(self, X):
252     """
253     Use the trained weights of this two-layer network to predict
254     labels for
255     data points. For each data point we predict scores for each of
256     the C
257     classes, and assign each data point to the class with the
258     highest score.
259
260     Inputs:
261     - X: A numpy array of shape (N, D) giving N D-dimensional data
262         points to
263         classify.
264
265     Returns:
266     - y_pred: A numpy array of shape (N,) giving predicted labels
267         for each of
268         the elements of X. For all i, y_pred[i] = c means that X[i]
269         is predicted
270         to have class c, where 0 <= c < C.
271     """
272     first_layer_output=np.dot(X,self.params['W1'])+self.params['b1']
273     activated_result=self._relu(first_layer_output)
274     second_layer_output=np.dot(activated_result,self.params['W2'])+self.params['b2']
275     softmax_output=self._softmax(second_layer_output)
276     y_pred = np.argmax(softmax_output,axis=1)
277
278     #####
279     # TODO: Implement this function; it should be VERY simple!
280     #
281     #####
282     #
283     #####
284     #
285     #####
286     #
287     #####
288     #
289     #####
290     #
291     #####
292     #
293     #####
294     #
295     #####
296     #
297     #####
298     #
299     #####
300     #
301     #####
302     #
303     #####
304     #
305     #####
306     #
307     #####
308     #
309     #####
310     #
311     #####
312     #
313     #####
314     #
315     #####
316     #
317     #####
318     #
319     #####
320     #
321     #####
322     #
323     #####
324     #
325     #####
326     #
327     #####
328     #
329     #####
330     #
331     #####
332     #
333     #####
334     #
335     #####
336     #
337     #####
338     #
339     #####
340     #
341     #####
342     #
343     #####
344     #
345     #####
346     #
347     #####
348     #
349     #####
350     #
351     #####
352     #
353     #####
354     #
355     #####
356     #
357     #####
358     #
359     #####
360     #
361     #####
362     #
363     #####
364     #
365     #####
366     #
367     #####
368     #
369     #####
370     #
371     #####
372     #
373     #####
374     #
375     #####
376     #
377     #####
378     #
379     #####
380     #
381     #####
382     #
383     #####
384     #
385     #####
386     #
387     #####
388     #
389     #####
390     #
391     #####
392     #
393     #####
394     #
395     #####
396     #
397     #####
398     #
399     #####
400     #
401     #####
402     #
403     #####
404     #
405     #####
406     #
407     #####
408     #
409     #####
410     #
411     #####
412     #
413     #####
414     #
415     #####
416     #
417     #####
418     #
419     #####
420     #
421     #####
422     #
423     #####
424     #
425     #####
426     #
427     #####
428     #
429     #####
430     #
431     #####
432     #
433     #####
434     #
435     #####
436     #
437     #####
438     #
439     #####
440     #
441     #####
442     #
443     #####
444     #
445     #####
446     #
447     #####
448     #
449     #####
450     #
451     #####
452     #
453     #####
454     #
455     #####
456     #
457     #####
458     #
459     #####
460     #
461     #####
462     #
463     #####
464     #
465     #####
466     #
467     #####
468     #
469     #####
470     #
471     #####
472     #
473     #####
474     #
475     #####
476     #
477     #####
478     #
479     #####
480     #
481     #####
482     #
483     #####
484     #
485     #####
486     #
487     #####
488     #
489     #####
490     #
491     #####
492     #
493     #####
494     #
495     #####
496     #
497     #####
498     #
499     #####
500     #
501     #####
502     #
503     #####
504     #
505     #####
506     #
507     #####
508     #
509     #####
510     #
511     #####
512     #
513     #####
514     #
515     #####
516     #
517     #####
518     #
519     #####
520     #
521     #####
522     #
523     #####
524     #
525     #####
526     #
527     #####
528     #
529     #####
530     #
531     #####
532     #
533     #####
534     #
535     #####
536     #
537     #####
538     #
539     #####
540     #
541     #####
542     #
543     #####
544     #
545     #####
546     #
547     #####
548     #
549     #####
550     #
551     #####
552     #
553     #####
554     #
555     #####
556     #
557     #####
558     #
559     #####
560     #
561     #####
562     #
563     #####
564     #
565     #####
566     #
567     #####
568     #
569     #####
570     #
571     #####
572     #
573     #####
574     #
575     #####
576     #
577     #####
578     #
579     #####
580     #
581     #####
582     #
583     #####
584     #
585     #####
586     #
587     #####
588     #
589     #####
590     #
591     #####
592     #
593     #####
594     #
595     #####
596     #
597     #####
598     #
599     #####
600     #
601     #####
602     #
603     #####
604     #
605     #####
606     #
607     #####
608     #
609     #####
610     #
611     #####
612     #
613     #####
614     #
615     #####
616     #
617     #####
618     #
619     #####
620     #
621     #####
622     #
623     #####
624     #
625     #####
626     #
627     #####
628     #
629     #####
630     #
631     #####
632     #
633     #####
634     #
635     #####
636     #
637     #####
638     #
639     #####
640     #
641     #####
642     #
643     #####
644     #
645     #####
646     #
647     #####
648     #
649     #####
650     #
651     #####
652     #
653     #####
654     #
655     #####
656     #
657     #####
658     #
659     #####
660     #
661     #####
662     #
663     #####
664     #
665     #####
666     #
667     #####
668     #
669     #####
670     #
671     #####
672     #
673     #####
674     #
675     #####
676     #
677     #####
678     #
679     #####
680     #
681     #####
682     #
683     #####
684     #
685     #####
686     #
687     #####
688     #
689     #####
690     #
691     #####
692     #
693     #####
694     #
695     #####
696     #
697     #####
698     #
699     #####
700     #
701     #####
702     #
703     #####
704     #
705     #####
706     #
707     #####
708     #
709     #####
710     #
711     #####
712     #
713     #####
714     #
715     #####
716     #
717     #####
718     #
719     #####
720     #
721     #####
722     #
723     #####
724     #
725     #####
726     #
727     #####
728     #
729     #####
730     #
731     #####
732     #
733     #####
734     #
735     #####
736     #
737     #####
738     #
739     #####
740     #
741     #####
742     #
743     #####
744     #
745     #####
746     #
747     #####
748     #
749     #####
750     #
751     #####
752     #
753     #####
754     #
755     #####
756     #
757     #####
758     #
759     #####
760     #
761     #####
762     #
763     #####
764     #
765     #####
766     #
767     #####
768     #
769     #####
770     #
771     #####
772     #
773     #####
774     #
775     #####
776     #
777     #####
778     #
779     #####
780     #
781     #####
782     #
783     #####
784     #
785     #####
786     #
787     #####
788     #
789     #####
790     #
791     #####
792     #
793     #####
794     #
795     #####
796     #
797     #####
798     #
799     #####
800     #
801     #####
802     #
803     #####
804     #
805     #####
806     #
807     #####
808     #
809     #####
810     #
811     #####
812     #
813     #####
814     #
815     #####
816     #
817     #####
818     #
819     #####
820     #
821     #####
822     #
823     #####
824     #
825     #####
826     #
827     #####
828     #
829     #####
830     #
831     #####
832     #
833     #####
834     #
835     #####
836     #
837     #####
838     #
839     #####
840     #
841     #####
842     #
843     #####
844     #
845     #####
846     #
847     #####
848     #
849     #####
850     #
851     #####
852     #
853     #####
854     #
855     #####
856     #
857     #####
858     #
859     #####
860     #
861     #####
862     #
863     #####
864     #
865     #####
866     #
867     #####
868     #
869     #####
870     #
871     #####
872     #
873     #####
874     #
875     #####
876     #
877     #####
878     #
879     #####
880     #
881     #####
882     #
883     #####
884     #
885     #####
886     #
887     #####
888     #
889     #####
890     #
891     #####
892     #
893     #####
894     #
895     #####
896     #
897     #####
898     #
899     #####
900     #
901     #####
902     #
903     #####
904     #
905     #####
906     #
907     #####
908     #
909     #####
910     #
911     #####
912     #
913     #####
914     #
915     #####
916     #
917     #####
918     #
919     #####
920     #
921     #####
922     #
923     #####
924     #
925     #####
926     #
927     #####
928     #
929     #####
930     #
931     #####
932     #
933     #####
934     #
935     #####
936     #
937     #####
938     #
939     #####
940     #
941     #####
942     #
943     #####
944     #
945     #####
946     #
947     #####
948     #
949     #####
950     #
951     #####
952     #
953     #####
954     #
955     #####
956     #
957     #####
958     #
959     #####
960     #
961     #####
962     #
963     #####
964     #
965     #####
966     #
967     #####
968     #
969     #####
970     #
971     #####
972     #
973     #####
974     #
975     #####
976     #
977     #####
978     #
979     #####
980     #
981     #####
982     #
983     #####
984     #
985     #####
986     #
987     #####
988     #
989     #####
990     #
991     #####
992     #
993     #####
994     #
995     #####
996     #
997     #####
998     #
999     #####
1000    #

```