

Restricted Boltzmann Machine

The *learning rate* was set to 0.006, k to 10, *batch size* to 100, and training was performed over 1500 *epochs*. These parameters offered a good balance across all hidden neuron configurations. A lower learning rate improved performance for $M=4,8$ but had a negative impact on $M=1,2$. The number of *epochs* ensured $M=4,8$ accurately reproduced the target distribution, while k was selected to guarantee the system "forgot" prior states and avoided dependence on earlier dynamics. Reducing k degraded the quality of distribution approximation. The *batch size* allowed for an appropriate representation of the target training distribution, approximately $[1/4, 0, 0, 1/4, 0, 1/4, 1/4, 0]$.

For sampling, I initialized the visible states randomly, ran the dynamics for k steps, sampled the updated states, and repeated this process 10,000 times. I converted the 10,000 binary states to decimal indices, counted their frequencies, and computed the Kullback-Leibler divergence, as illustrated in Figure 1. The results validate the accuracy of the reproduced distribution, as shown in Table 1.

M (Hidden Neurons)	D_{KL}	D_{KL} -bound
1	0.69874	0.69315
2	0.17378	0.34657
4	0.00652	0.0
8	0.00243	0.0

Table 1: Comparison of D_{KL} and D_{KL} -bound for different values of M (hidden neurons). The plot of these values is represented in Figure 1.

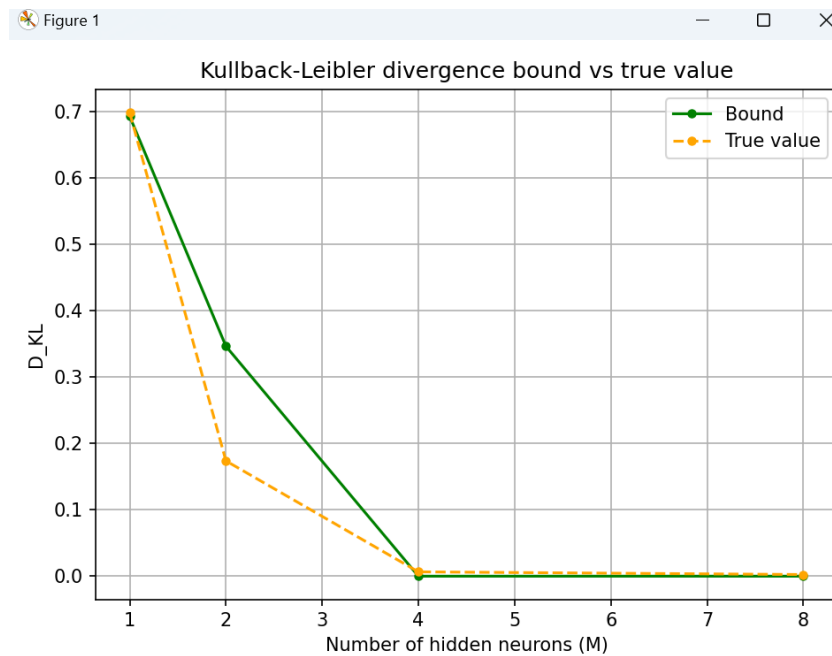


Figure 1: The figure represents the Kullback-Leibler divergence (D_{KL}) as a function of hidden neurons (M). The estimated upper bound (Green), and the value calculated after sampling (Orange).

Restricted Boltzmann Machine

Eric Blohm

October 7, 2024

Here is the Python code for the Restricted Boltzmann Machine:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def sample_patterns_equal_prob(batch_size, patterns):
6     pattern_indices = np.random.choice(len(
7         patterns), size=batch_size, p=[0.25, 0.25,
8         0.25, 0.25])
9     mini_batch = np.array([patterns[idx] for idx
10         in pattern_indices])
11     return mini_batch
12
13
14 def p_boltzmann(b):
15     denominator = 1+np.exp(-2*b)
16     return 1/denominator
17
18
19 def compute_delta_w(eta, b_i_h_0, v_0, b_i_h, v, w):
20     delta_w = w.copy()
21     for m in range(0, len(w)):
22         for n in range(0, len(w[0])):
23             term1 = np.tanh(b_i_h_0[m])*v_0[n]
24             term2 = np.tanh(b_i_h[m])*v[n]
25             delta_w[m][n] = eta*(term1 - term2)
26     return delta_w
27
28
29 def compute_delta_theta_v(eta, v_0, v, theta_v):
30     delta_theta_v = np.zeros(len(theta_v))
31     for n in range(0, len(theta_v)):
32         delta_theta_v[n] = -eta*(v_0[n]-v[n])
33     return delta_theta_v
```

```

31
32
33 def compute_delta_theta_h(eta, b_i_h_0, b_i_h, theta_h
    ):
34     delta_theta_h = np.zeros(len(theta_h))
35     for m in range(0, len(theta_h)):
36         term1 = np.tanh(b_i_h_0[m])
37         term2 = np.tanh(b_i_h[m])
38         delta_theta_h[m] = -eta*(term1-term2)
39     return delta_theta_h
40
41
42
43 def train_RBM(M, eta, k, epochs, batch_size): #, patterns)
44
45     PD = np.array([0.25, 0, 0, 0.25, 0, 0.25, 0.25,
46                    0])
47
48     ### Init ###
49     #patterns that have a 1/4 probability to be
50     #sampled, used for training
51     patterns = np.array([[ -1, -1, -1],
52                          [ -1, 1, 1],
53                          [ 1, -1, 1],
54                          [ 1, 1, -1]])
55
56     variance = 1/ np.maximum(3, M)
57     std = np.sqrt(variance)
58     w = np.random.normal(0, std, size=(M, 3))
59
60     theta_v = np.zeros(3)
61     theta_h = np.zeros(M)
62     v = np.zeros(3)
63     h = np.zeros(M)
64     #####
65
66     energies = []
67     all_samples = []
68
69     for epoch in range(0, epochs):
70
71         ### Init weights and thresholds ###
72         delta_w = np.zeros((M, 3))
73         delta_theta_v = np.zeros(3)

```

```

74     delta_theta_h = np.zeros(M)
75     #####
76
77     ### Sample from patterns with equal prob ###
78     mini_batch = sample_patterns_equal_prob(
79         batch_size, patterns)
80     #####
81
82     for sample in mini_batch:
83         all_samples.append(sample)
84
85     for mu, pattern in enumerate(mini_batch):
86         v = pattern.copy()
87         v_0 = v.copy()
88         b_i_h_0 = np.zeros(M)
89
90         ### Calculate  $b_i^h(0)$  and update all
91         hidden neurons  $h_i(0)$  ###
92         for i in range(0, len(b_i_h_0)):
93             b_i_h_0[i] = np.dot(w[i], v_0) - theta_h[
94                 i]
95             r = np.random.rand()
96             p_B = p_boltzmann(b_i_h_0[i])
97             if(r < p_B):
98                 h[i] = 1
99             else:
100                 h[i] = -1
101             #
102             #####
103
104         b_i_h = b_i_h_0.copy()
105         b_j_v = np.zeros(3)
106         for step in range(0, k):
107             ### update visible neurons ###
108             for j in range(0, 3):
109                 b_j_v[j] = np.dot(h, w[:, j]) -
110                     theta_v[j]
111                 p_B_v = p_boltzmann(b_j_v[j])
112                 r = np.random.rand()
113                 if(r < p_B_v):
114                     v[j] = 1
115                 else:
116                     v[j] = -1
117             #####
118
119             ### update hidden neurons ###

```

```

114         for i in range(0,M):
115             b_i_h[i] = np.dot(w[i],v)-theta_h[
                i]
116             p_B_h = p_boltzmann(b_i_h[i])
117             r = np.random.rand()
118             if(r < p_B_h):
119                 h[i] = 1
120             else:
121                 h[i] = -1
122             #####
123             ### calculate delta_w ###
124             for m in range(0,M):
125                 for n in range(0,3):
126                     delta_w[m][n] += eta* ( np.tanh(
                        b_i_h_0[m])*v_0[n] - np.tanh(
                        b_i_h[m])*v[n] )
127             #####
128
129             ### calculate delta_theta_v ###
130             for n in range(0,3):
131                 delta_theta_v[n] -= eta*(v_0[n]-v[n])
132             #####
133
134             ## calculate delta_theta_h ###
135             for m in range(0, M):
136                 delta_theta_h[m] -= eta*(np.tanh(
                        b_i_h_0[m])-np.tanh(b_i_h[m]))
137             #####
138
139             ### update values ###
140             w += delta_w
141             theta_h += delta_theta_h
142             theta_v += delta_theta_v
143             #####
144
145             ### Monitor energy function ###
146             H = compute_energy_function(w, h, v , theta_v,
                theta_h)
147             energies.append(H)
148
149             tmp_freq = compute_frequencies(all_samples)
150             print("training distribution: ", tmp_freq)
151
152             ### Monitor energy function ###
153             epoch_range = np.arange(epochs)
154             plt.plot(epoch_range,energies,marker='o', color='

```

```

        green', markersize=1, linestyle='--')
155 plt.title(f'Energy of training, M={M}')
156 plt.xlabel('Epochs')
157 plt.ylabel('Energy')
158 plt.grid()
159 plt.tight_layout()
160 plt.show()
161
162 return w, theta_h, theta_v
163
164 def D_kl_bound(M):
165     #number of inputs
166     N=3
167     expression = 3 - int(np.log2(M+1)) - ((M+1)/(2**
        int(np.log2(M+1))))
168     if M < (2**(N-1) - 1):
169         return np.log(2)*expression
170
171     elif M >= (2**(N-1) - 1):
172         return np.log(2)*0
173
174
175 def D_kl(PD,PB):
176     sum= 0
177     for mu in range (0,len(PD)):
178         if(PB[mu] > 0 and PD[mu] > 0):
179             #print("PB^mu: ", PB[mu], ", ", "PD^mu: ",
                PD[mu])
180             sum+= PD[mu]* np.log(PD[mu]/PB[mu])
181     return sum
182
183 #convert binary number to decimal index, then
    increment.
184 def compute_frequencies(samples):
185     n_patterns = 2**len(samples[0])
186     PB = np.zeros(n_patterns)
187
188     len_non_filled = 0
189
190     for pattern in samples:
191         #in case samples array is not full.
192         if(np.array_equal(pattern, [0,0,0])):
193             continue
194         else:
195             len_non_filled +=1
196

```

```

197         #convert to 0 and 1 bits
198         binary_pattern = []
199         for bit in pattern:
200             if bit == 1:
201                 binary_pattern.append(1)
202             else:
203                 binary_pattern.append(0)
204
205         # convert binary pattern to decimal index
206         idx = 0
207         n_bits = len(samples[0])
208         for i,bit in enumerate(binary_pattern):
209             idx += bit * (2**(n_bits-i-1))
210         #print("Index: ", idx)
211         PB[idx] +=1
212     #normalize
213     PB = PB/len_non_filled
214     return PB
215
216
217 def compute_energy_function(w, h, v , theta_v, theta_h
218 ):
219     term1_sum1 = 0
220     for i in range(0,len(h)):
221         term1_sum2 = 0
222         for j in range(0,len(v)):
223             term1_sum2 += w[i][j]*h[i]*v[j]
224         term1_sum1 += term1_sum2
225
226     term2_sum = 0
227     for j in range(0,len(theta_v)):
228         term2_sum += theta_v[j]*v[j]
229
230     term3_sum = 0
231     for i in range(0,len(theta_h)):
232         term3_sum += theta_h[i]*h[i]
233
234     return -term1_sum1 + term2_sum + term3_sum
235
236 def main():
237     ### Init ###
238     # pattern = 0.25 for index 0,3,5,6, used when
239     # sampling
240     all_patterns = np.array([[ -1, -1, -1],
241                             [ -1, -1, 1],

```

```

241         [-1, 1, -1],
242         [-1, 1, 1],
243         [1, -1, -1],
244         [1, -1, 1],
245         [1, 1, -1],
246         [1, 1, 1]])
247     M_values = [1, 2, 4, 8]
248     d_kl_bound_values = []
249     d_kl_values = []
250     eta = 0.006
251     k = 10
252     batch_size = 100
253     epochs = 1500
254
255     # sample using the dynamincs in the CD_k algorithm
256     #
257     num_iterations = 10000
258     max_T = 10
259     print(f"Sampling: num_iterations={num_iterations},
260           T={max_T}")
261     #####
262     for M in M_values:
263         print("-----")
264         print(f"Training configuration: M={M} | eta={
265               eta} | k={k} | batch_size={batch_size} |
266               epochs={epochs}")
267
268         w, theta_h, theta_v = train_RBM(M, eta, k, epochs
269                                         , batch_size)
270         print(f"\n w={w} | theta_h={theta_h} |
271               theta_v={theta_v}")
272
273         PD = np.array([0.25, 0, 0, 0.25, 0, 0.25,
274                        0.25, 0])
275
276         h = np.zeros(M)
277
278         samples = np.zeros((num_iterations, 3))
279
280         for step in range(0, num_iterations):
281             v = all_patterns[np.random.randint(
282                             all_patterns.shape[0])].copy()
283             b_j_v = np.zeros(len(v))
284             b_i_h = np.zeros(len(h))

```



```

279         for T in range(0,max_T):
280             ### update hidden neurons ###
281             for i in range(0,M):
282                 b_i_h[i] = np.dot(w[i],v)-theta_h[
                    i]
283                 p_B_h = p_boltzmann(b_i_h[i])
284                 r = np.random.rand()
285                 if(r < p_B_h):
286                     h[i] = 1
287                 else:
288                     h[i] = -1
289             #####
290
291             ### update visible neurons ###
292             for j in range(0,3):
293                 b_j_v[j] = np.dot(h,w[:,j]-theta_v
                    [j])
294                 p_B_v = p_boltzmann(b_j_v[j])
295                 r = np.random.rand()
296                 if(r < p_B_v):
297                     v[j] = 1
298                 else:
299                     v[j] = -1
300             #####
301
302
303         samples[step] = v
304
305         #Implement early stopping since i need
            different "num_iterations" for
            different M. modulo 10 for performance
            reasons
306         if(step % 10 == 0):
307             tmp = compute_frequencies(samples)
308             print(f"runtime PD: {tmp}, iteration:
                {step}")
309
310
311         print(f"Results:\nM={M}")
312         PB = compute_frequencies(samples)
313         print("PB: ", PB, "sum =", np.sum(PB))
314         print("PD: ", PD)
315
316         d_kl_bound = D_kl_bound(M)
317         d_kl_bound_values.append(d_kl_bound)
318

```

```

319         d_kl = D_kl(PD,PB)
320         d_kl_values.append(d_kl)
321         print(f"D_KL: {d_kl}, D_KL_bound: {d_kl_bound}
322               ")
323         print("-----")
324     print("D_KL: " ,d_kl_values," ", "D_KL_bound: ",
325           d_kl_bound_values)
326     plt.plot(M_values,d_kl_bound_values,marker='o',
327              color='green', markersize=4, linestyle='-',
328              label='Bound')
329     plt.plot(M_values,d_kl_values,marker='o', color='
330              orange', markersize=4, linestyle='--',label='
331              True value')
332     plt.title('Kullback-Leibler divergence bound vs
333              true value')
334     plt.xlabel('Number of hidden neurons (M)')
335     plt.ylabel('D_KL')
336     plt.grid()
337     plt.legend()
338     plt.tight_layout()
339     plt.show()
340
341 if __name__ == "__main__":
342     main()

```

Perceptron with one hidden layer

Eric Blohm

October 1, 2024

Here is the Python code for the perception with one hidden layer:

```
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 #data has the input in the first two elements and
  #output on the third.
6 def GetInputOutput(data):
7     input = []
8     output = []
9     for row in data:
10         input.append([row[0],row[1]])
11         output.append(row[2])
12     return np.array(input),np.array(output)
13
14
15 def getCSV(file):
16     df = pd.read_csv(file,header=None)
17     data_list = df.values.tolist()
18     input,output = GetInputOutput(data_list)
19     return input,output
20
21
22 def init_w_theta(M):
23     # between input and hidden
24     # 1/2 from the number of inputs
25     variance = 1/2
26     standard_dev = np.sqrt(variance)
27     theta_j = np.zeros(M)
28     w_jk = np.random.normal(0, standard_dev, size=(M
29         ,2))
30
31     # between hidden and output
32     variance_h = 1/M
```

```

32     standard_dev_h = np.sqrt(variance_h)
33     theta = 0
34     w_j = np.random.normal(0, standard_dev_h, size=(M)
35         )
36     return w_jk, w_j, theta_j, theta
37
38 def compute_hidden_output(w_jk, theta_j, input):
39     b_j = np.dot(w_jk, input.T) - theta_j.T
40     return np.tanh(b_j)
41
42
43 def compute_network_output(w_j, theta, hidden_output, mu)
44 :
45     sum = 0
46     for j in range(0, len(w_j)):
47         sum += w_j[j]*hidden_output[mu][j]
48     B_i = sum - theta
49     return np.tanh(B_i)
50
51 def back_prop(output_error, w_j, hidden_output,
52     hidden_error):
53     for m in range(0, len(w_j)):
54         hidden_error[m] = output_error * w_j[m]* (1-
55             hidden_output[m]**2)
56     return hidden_error
57
58 def get_delta_w(input, hidden_output, hidden_error,
59     output_error, eta, mini_batch, M):
60     delta_w_j = np.zeros(M)
61     #Delta_m, V_n. m = 1 only 1 output per pattern
62     for n in range(0, len(hidden_output[0])):
63         for mu in range(0, mini_batch):
64             delta_w_j[n] += output_error[mu]*
65                 hidden_output[mu][n]
66
67
68     delta_w_jk = np.zeros((M, len(input[0])))
69     #m is every hidden neuron.
70     for m in range(0, len(hidden_error[0])):
71         #n is x_1 and x_2
72         for n in range(0, len(input[0])):
73             for mu in range(0, mini_batch):
74                 delta_w_jk[m][n] += hidden_error[mu][m]

```

```

72         ]*input[mu][n]
73     return eta*delta_w_jk, eta*delta_w_j
74
75
76 def get_delta_theta(output_error,hidden_error,eta,
77 mini_batch,M):
78     delta_theta = 0
79     #m = 1
80     for mu in range(0,mini_batch):
81         delta_theta += output_error[mu]
82
83     delta_theta_j = np.zeros(M)
84     for m in range(0,len(hidden_error[0])):
85         for mu in range(0,mini_batch):
86             delta_theta_j[m] += hidden_error[mu][m]
87
88     return -eta*delta_theta_j, -eta*delta_theta
89
90 def compute_classification_error(output,target):
91     sum = 0
92     for mu in range(0,len(target)):
93         sum+= np.abs((np.sign(output[mu])-target[mu]))
94     return (1/(2*len(target)))*sum
95
96
97 def compute_energy_function(output,target):
98     sum = 0
99     for mu in range(0,len(target)):
100         sum += (target[mu]-output[mu])**2
101     return 0.5*sum
102
103
104 def save_values(weights_jk,weights_j,threshold_1,
105 threshold_2):
106     df = pd.DataFrame(weights_jk)
107     df.to_csv('w1.csv', index=False,header=False)
108
109     df = pd.DataFrame(weights_j)
110     df.to_csv('w2.csv',index=False, header=False)
111
112     df = pd.DataFrame(threshold_1)
113     df.to_csv('t1.csv',index=False, header=False)
114
115     df = pd.DataFrame([threshold_2])

```

```

115         df.to_csv('t2.csv',index=False, header=False)
116
117
118     def main():
119
120         ##### configuration #####
121         M = 10
122         epochsMax = 500
123         batch_size = 64
124         eta = 0.01
125         #####
126
127         ### Retrieve data ###
128         input,target = getCSV('training_set.csv')
129         input_validation,target_validation = getCSV('
            validation_set.csv')
130         #####
131
132         ##### Center and normalize data #####
133         input_mean = np.mean(input, axis=0)
134         input_std = np.std(input, axis=0)
135         ## Normalize based in training metrics
136         input = (input - input_mean) / input_std
137         input_validation = (input_validation - input_mean)
            / input_std
138         #####
139
140         ### initialize weights and thresholds ###
141         w_jk,w_j,theta_j,theta = init_w_theta(M)
142         #####
143
144
145         ## used for plotting ##
146         c_train_list = np.zeros(epochsMax)
147         c_validate_list = np.zeros(epochsMax)
148         #####
149
150         for epoch in range(0,epochsMax):
151             ### Shuffle the input data and targets ###
152             indices = np.arange(len(input))
153             np.random.shuffle(indices)
154             input = input[indices]
155             target = target[indices]
156             #####
157
158             ##### Create mini batches #####

```

```

159     for start in range(0, len(input), batch_size):
160         end = start + batch_size
161         mini_batch = input[start:end]
162         target_batch = target[start:end]
163
164         ### Initialize outputs for the mini-batch
165         ###
166         hidden_output = np.zeros((len(mini_batch),
167                                   M))
168         output = np.zeros(len(mini_batch))
169         output_error = np.zeros(len(mini_batch))
170         hidden_error = np.zeros((len(mini_batch), M))
171
172         ##### for each pattern in mini batch #####
173         for mu in range(0, len(mini_batch)):
174             #only one layer
175             ##### Feed forward #####
176             hidden_output[mu] =
177                 compute_hidden_output(w_jk, theta_j,
178                                       mini_batch[mu])
179             output[mu] = compute_network_output(
180                 w_j, theta, hidden_output, mu)
181             #####
182
183             ##### back propagation #####
184             output_error[mu] = (target_batch[mu] -
185                                 output[mu])*(1-output[mu]**2)
186             for m in range(0, len(w_j)):
187                 hidden_error[mu][m] = output_error
188                     [mu] * w_j[m] * (1-hidden_output
189                     [mu][m]**2)
190             #####
191
192             ##### Update weights #####
193             #print(f"\n-Weights_jk before update: {
194                   w_jk}, \nWeights_j before update: {w_j
195                   }")
196             delta_w_jk, delta_w_j = get_delta_w(
197                 mini_batch, hidden_output, hidden_error,
198                 output_error, eta, len(mini_batch), M)
199             w_jk += delta_w_jk
200             w_j += delta_w_j
201
202             delta_theta_j, delta_theta =
203                 get_delta_theta(output_error,

```

```

191         hidden_error, eta, len(mini_batch), M)
192         theta_j += delta_theta_j
193         theta += delta_theta
194         #####
195     ### validate during training and early stop
196     ###
197     hidden_output_validate = np.zeros((len(
198         input_validation), M))
199     output_validate = np.zeros(len(
200         input_validation))
201     for mu in range(0, len(input_validation)):
202         hidden_output_validate[mu] =
203             compute_hidden_output(w_jk, theta_j,
204                 input_validation[mu])
205         output_validate[mu] =
206             compute_network_output(w_j, theta,
207                 hidden_output_validate, mu)
208     #
209     #####
210
211     ## Compute classification error and energy
212     function. ##
213     c = compute_classification_error(
214         output_validate, target_validation)
215     H_validate = compute_energy_function(
216         output_validate, target_validation)
217     print("C:", c*100, ", Energy function: ",
218         H_validate)
219     c_validate_list[epoch] = c*100
220     if (c < 0.12):
221         break
222     #
223     #####
224
225     #if stopped early, retrieve all no negative
226     elements
227     c_validate_list = c_validate_list[c_validate_list
228         > 0]
229
230     ## create list for plotting ##
231     epochs = np.arange(len(c_validate_list))
232
233     ## plot Validation classification error ##

```



```

219 plt.plot(epochs, c_validate_list, marker='o',
           color='green', markersize=4, linestyle='--')
220 plt.title('Validation Classification Error')
221 plt.xlabel('Epochs')
222 plt.ylabel('c_validate')
223 plt.grid()
224
225 plt.tight_layout()
226 plt.show()
227 #
           #####
228
229 ### Validate network after training ###
230 hidden_output_validate = np.zeros((len(
           input_validation),M))
231 output_validate = np.zeros(len(input_validation))
232 for mu in range(0,len(input_validation)):
233     hidden_output_validate[mu] =
           compute_hidden_output(w_jk,theta_j,
           input_validation[mu])
234     output_validate[mu] = compute_network_output(
           w_j,theta,hidden_output_validate,mu)
235 c = compute_classification_error(output_validate,
           target_validation)
236 H_validate = compute_energy_function(
           output_validate,target_validation)
237 print("C:", c*100, ", Energy function: ",
           H_validate)
238 #####
239
240 ### save the weights and thresholds to csv files
           ###
241 #save_values(w_jk,w_j,theta_j,theta)
242 #
           #####
243
244
245 if __name__ == "__main__":
246     main()

```