Restricted Boltzmann Machine

Eric Blohm

October 7, 2024

Here is the Python code for the Restricted Boltzmann Machine:

```python
import numpy as np
import matplotlib.pyplot as plt


def sample_patterns_equal_prob(batch_size,patterns):
        pattern_indices = np.random.choice(len(
            patterns), size=batch_size, p=[0.25, 0.25,
            0.25, 0.25])
        mini_batch = np.array([patterns[idx] for idx
            in pattern_indices])
        return mini_batch


def p_boltzmann(b):
    denominator = 1+np.exp(-2*b)
    return 1/denominator


def compute_delta_w(eta, b_i_h_0, v_0, b_i_h, v, w):
    delta_w = w.copy()
    for m in range(0,len(w)):
        for n in range(0,len(w[0])):
            term1 = np.tanh(b_i_h_0[m])*v_0[n]
            term2 = np.tanh(b_i_h[m])*v[n]
            delta_w[m][n] = eta*(term1 - term2)
    return delta_w


def compute_delta_theta_v(eta, v_0, v, theta_v):
    delta_theta_v = np.zeros(len(theta_v))
    for n in range(0,len(theta_v)):
        delta_theta_v[n] = -eta*(v_0[n]-v[n])
    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,
            0])
46
47      ### Init ###
48      #patterns that have a 1/4 probability to be
            sampled, used for training
49      patterns = np.array([[-1,-1,-1],
50                           [-1,1,1],
51                           [1,-1,1],
52                           [1,1,-1]])
53
54      variance = 1/ np.maximum(3,M)
55      std = np.sqrt(variance)
56      w = np.random.normal(0, std, size=(M,3))
57
58      theta_v = np.zeros(3)
59      theta_h = np.zeros(M)
60      v = np.zeros(3)
61      h = np.zeros(M)
62      ############
63
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(
                batch_size,patterns)
79          ###########################################
80
81          for sample in mini_batch:
82              all_samples.append(sample)
83
84          for mu, pattern in enumerate(mini_batch):
85              v = pattern.copy()
86              v_0 = v.copy()
87              b_i_h_0 = np.zeros(M)
88
89              ### Calculate b_i^h(0) and update all
                    hidden neurons h_i(0) ###
90              for i in range(0,len(b_i_h_0)):
91                  b_i_h_0[i] = np.dot(w[i],v_0)-theta_h[
                        i]
92                  r = np.random.rand()
93                  p_B = p_boltzmann(b_i_h_0[i])
94                  if(r < p_B):
95                      h[i] = 1
96                  else:
97                      h[i] = -1
98              #
                    ##########################################################
99              b_i_h = b_i_h_0.copy()
100             b_j_v = np.zeros(3)
101             for step in range(0,k):
102                 ### update visible neurons ###
103                 for j in range(0,3):
104                     b_j_v[j] = np.dot(h, w[:,j]) -
                            theta_v[j]
105                     p_B_v = p_boltzmann(b_j_v[j])
106                     r =  np.random.rand()
107                     if(r < p_B_v):
108                         v[j] = 1
109                     else:
110                         v[j] = -1
111                 ###############################
112
113                 ### update hidden neurons ###
```

```
                for i in range(0,M):
                    b_i_h[i] = np.dot(w[i],v)-theta_h[
                        i]
                    p_B_h = p_boltzmann(b_i_h[i])
                    r =  np.random.rand()
                    if(r < p_B_h):
                        h[i] = 1
                    else:
                        h[i] = -1
                ############################
            ### calculate delta_w ###
            for m in range(0,M):
                for n in range(0,3):
                    delta_w[m][n] += eta* ( np.tanh(
                        b_i_h_0[m])*v_0[n] - np.tanh(
                        b_i_h[m])*v[n] )
            ##########################

            ### calculate delta_theta_v ###
            for n in range(0,3):
                delta_theta_v[n] -= eta*(v_0[n]-v[n])
            ###############################

            ## calculate delta_theta_h ###
            for m in range(0, M):
                delta_theta_h[m] -= eta*(np.tanh(
                    b_i_h_0[m])-np.tanh(b_i_h[m]))
            ##############################

        ### update values ###
        w += delta_w
        theta_h += delta_theta_h
        theta_v += delta_theta_v
        ####################

        ### Monitor energy function ###
        H = compute_energy_function(w, h, v , theta_v,
            theta_h)
        energies.append(H)

    tmp_freq = compute_frequencies(all_samples)
    print("training distribution: ", tmp_freq)

    ### Monitor energy function ###
    epoch_range = np.arange(epochs)
    plt.plot(epoch_range,energies,marker='o', color='
```

```
        green', markersize=1, linestyle='-')
    plt.title(f'Energy of training, M={M}')
    plt.xlabel('Epochs')
    plt.ylabel('Energy')
    plt.grid()
    plt.tight_layout()
    plt.show()

    return w, theta_h, theta_v

def D_kl_bound(M):
    #number of inputs
    N=3
    expression = 3 - int(np.log2(M+1)) - ((M+1)/(2**
        int(np.log2(M+1))))
    if M < (2**(N-1) - 1):
        return np.log(2)*expression

    elif M >= (2**(N-1) - 1):
        return np.log(2)*0


def D_kl(PD,PB):
    sum= 0
    for mu in range (0,len(PD)):
        if(PB[mu] > 0 and PD[mu] > 0):
            #print("PB^mu: ", PB[mu], ", ", "PD^mu: ",
                PD[mu])
            sum+= PD[mu]* np.log(PD[mu]/PB[mu])
    return sum

#convert binary number to decimal index, then
    increment.
def compute_frequencies(samples):
    n_patterns = 2**len(samples[0])
    PB = np.zeros(n_patterns)

    len_non_filled = 0

    for pattern in samples:
        #in case samples array is not full.
        if(np.array_equal(pattern, [0,0,0])):
            continue
        else:
            len_non_filled +=1

```

```
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     term1_sum1 = 0
219     for i in range(0,len(h)):
220         term1_sum2 = 0
221         for j in range(0,len(v)):
222             term1_sum2 += w[i][j]*h[i]*v[j]
223         term1_sum1 += term1_sum2
224 
225     term2_sum = 0
226     for j in range(0,len(theta_v)):
227         term2_sum += theta_v[j]*v[j]
228 
229     term3_sum = 0
230     for i in range(0,len(theta_h)):
231         term3_sum += theta_h[i]*h[i]
232 
233     return -term1_sum1 + term2_sum + term3_sum
234 
235 def main():
236 
237     ### Init ###
238     # pattern = 0.25 for index 0,3,5,6, used when
            sampling
239     all_patterns = np.array([[-1,-1,-1],
240                              [-1,-1, 1],
```

```
                                [-1,  1,-1],
                                [-1,  1,  1],
                                [1,  -1,-1],
                                [1,  -1,  1],
                                [1,   1,-1],
                                [1,   1,  1]])
    M_values = [1, 2, 4, 8]
    d_kl_bound_values = []
    d_kl_values = []
    eta = 0.006
    k = 10
    batch_size = 100
    epochs = 1500

    # sample using the dynamincs in the CD_k algorithm
        #
    num_iterations = 10000
    max_T = 10
    print(f"Sampling: num_iterations={num_iterations},
        T={max_T}")
    ##########

    for M in M_values:
        print("_________________")
        print(f"Training configuration: M={M} | eta={
            eta} | k={k} | batch_size={batch_size} |
            epochs={epochs}")

        w, theta_h, theta_v = train_RBM(M,eta,k,epochs
            ,batch_size)
        print(f"\n  w={w} | theta_h={theta_h} |
            theta_v={theta_v}")

        PD = np.array([0.25, 0, 0, 0.25, 0, 0.25,
            0.25, 0])

        h = np.zeros(M)

        samples = np.zeros((num_iterations,3))

        for step in range(0,num_iterations):
            v = all_patterns[np.random.randint(
                all_patterns.shape[0])].copy()
            b_j_v = np.zeros(len(v))
            b_i_h = np.zeros(len(h))

```

```
            for T in range(0,max_T):
                 ### update hidden neurons ###
                for i in range(0,M):
                    b_i_h[i] = np.dot(w[i],v)-theta_h[
                       i]
                    p_B_h = p_boltzmann(b_i_h[i])
                    r = np.random.rand()
                    if(r < p_B_h):
                        h[i] = 1
                    else:
                        h[i] = -1
                ############################

                ### update visible neurons ###
                for j in range(0,3):
                    b_j_v[j] = np.dot(h,w[:,j]-theta_v
                       [j])
                    p_B_v = p_boltzmann(b_j_v[j])
                    r =  np.random.rand()
                    if(r < p_B_v):
                        v[j] = 1
                    else:
                        v[j] = -1
                ##############################


            samples[step] = v

            #Implement early stopping since i need
               different "num_iterations" for
               different M. modulo 10 for performance
               reasons
            if(step % 10 == 0):
                tmp = compute_frequencies(samples)
                print(f"runtime PD: {tmp}, iteration:
                   {step}")


        print(f"Results:\nM={M}")
        PB = compute_frequencies(samples)
        print("PB: ", PB, "sum =", np.sum(PB))
        print("PD: ", PD)

        d_kl_bound = D_kl_bound(M)
        d_kl_bound_values.append(d_kl_bound)
```

```
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             print("__________________")
323
324         print("D_KL: " ,d_kl_values,", ",  "D_KL_bound: ",
                d_kl_bound_values)
325
326         plt.plot(M_values,d_kl_bound_values,marker='o',
                color='green', markersize=4, linestyle='-',
                label='Bound')
327         plt.plot(M_values,d_kl_values,marker='o', color='
                orange', markersize=4, linestyle='--',label='
                True value')
328         plt.title('Kullback-Leibler divergence bound vs
                true value')
329         plt.xlabel('Number of hidden neurons (M)')
330         plt.ylabel('D_KL')
331         plt.grid()
332         plt.legend()
333         plt.tight_layout()
334         plt.show()
335
336
337
338 if __name__ == "__main__":
339         main()
```