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
```

```


def compute_delta_theta_h(eta, b_i_h_0, b_i_h, theta_h):
    delta_theta_h = np.zeros(len(theta_h))
    for m in range(0,len(theta_h)):
        term1 = np.tanh(b_i_h_0[m])
        term2 = np.tanh(b_i_h[m])
        delta_theta_h[m] = -eta*(term1-term2)
    return delta_theta_h



def train_RBM(M,eta,k,epochs,batch_size): #,patterns)

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

    ### Init ###
    #patterns that have a 1/4 probability to be sampled, used for training
    patterns = np.array([[-1,-1,-1],
                         [-1,1,1],
                         [1,-1,1],
                         [1,1,-1]])

    variance = 1/ np.maximum(3,M)
    std = np.sqrt(variance)
    w = np.random.normal(0, std, size=(M,3))

    theta_v = np.zeros(3)
    theta_h = np.zeros(M)
    v = np.zeros(3)
    h = np.zeros(M)
    ############

    energies = []
    all_samples = []

    for epoch in range(0,epochs):

        ### Init weights and thresholds ###
        delta_w = np.zeros((M,3))
        delta_theta_v = np.zeros(3)
        delta_theta_h = np.zeros(M)
        ##################################
```

```
        ### Sample from patterns with equal prob ###
        mini_batch = sample_patterns_equal_prob(batch_size,patterns)
        ############################################

        for sample in mini_batch:
            all_samples.append(sample)

        for mu, pattern in enumerate(mini_batch):
            v = pattern.copy()
            v_0 = v.copy()
            b_i_h_0 = np.zeros(M)

            ### Calculate b_i^h(0) and update all hidden neurons h_i(0) ###
            for i in range(0,len(b_i_h_0)):
                b_i_h_0[i] = np.dot(w[i],v_0)-theta_h[i]
                r = np.random.rand()
                p_B = p_boltzmann(b_i_h_0[i])
                if(r < p_B):
                    h[i] = 1
                else:
                    h[i] = -1
            # #############################################################
            b_i_h = b_i_h_0.copy()
            b_j_v = np.zeros(3)
            for step in range(0,k):
                ### 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
                ##############################

                ### 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)

    """
    ### 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


def compute_energy_function(w, h, v , theta_v, theta_h):
    term1_sum1 = 0
    for i in range(0,len(h)):
        term1_sum2 = 0
        for j in range(0,len(v)):
            term1_sum2 += w[i][j]*h[i]*v[j]
        term1_sum1 += term1_sum2

    term2_sum = 0
    for j in range(0,len(theta_v)):
        term2_sum += theta_v[j]*v[j]

    term3_sum = 0
    for i in range(0,len(theta_h)):
        term3_sum += theta_h[i]*h[i]

    return -term1_sum1 + term2_sum + term3_sum
```

```python

def main():

    ### Init ###
    # pattern = 0.25 for index 0,3,5,6, used when sampling
    all_patterns = np.array([[-1,-1,-1],
                             [-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 = 0.004
    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])
        PB = np.zeros(8)

        h = np.zeros(M)
```

```python

        for step in range(0,num_iterations):
            v = all_patterns[np.random.randint(all_patterns.shape[0])].copy()
            b_j_v = np.zeros(3)
            b_i_h = np.zeros(M)

            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
                ##############################

            for idx in range(0,len(PB)):
                if(np.array_equal(v,all_patterns[idx])):
                    PB[idx]+=1

            if(step % 10 == 0 and step >0):
                tmp = PB/step#compute_frequencies(samples)
                print(f"runtime PD: {tmp}, iteration: {step}")


        print(f"Results:\nM={M}")
        PB= PB/num_iterations
```

```
        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)

        d_kl = D_kl(PD,PB)
        d_kl_values.append(d_kl)
        print(f"D_KL: {d_kl}, D_KL_bound: {d_kl_bound}")
        print("_________________")

    print("D_KL: " ,d_kl_values,", ",  "D_KL_bound: ", d_kl_bound_values)

    plt.plot(M_values,d_kl_bound_values,marker='o', color='green', markersize=4, linestyle='-', label='Bound')
    plt.plot(M_values,d_kl_values,marker='o', color='orange', markersize=4, linestyle='--',label='True value')
    plt.title('Kullback-Leibler divergence bound vs true value')
    plt.xlabel('Number of hidden neurons (M)')
    plt.ylabel('D_KL')
    plt.grid()
    plt.legend()
    plt.tight_layout()
    plt.show()



if __name__ == "__main__":
    main()
```