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.

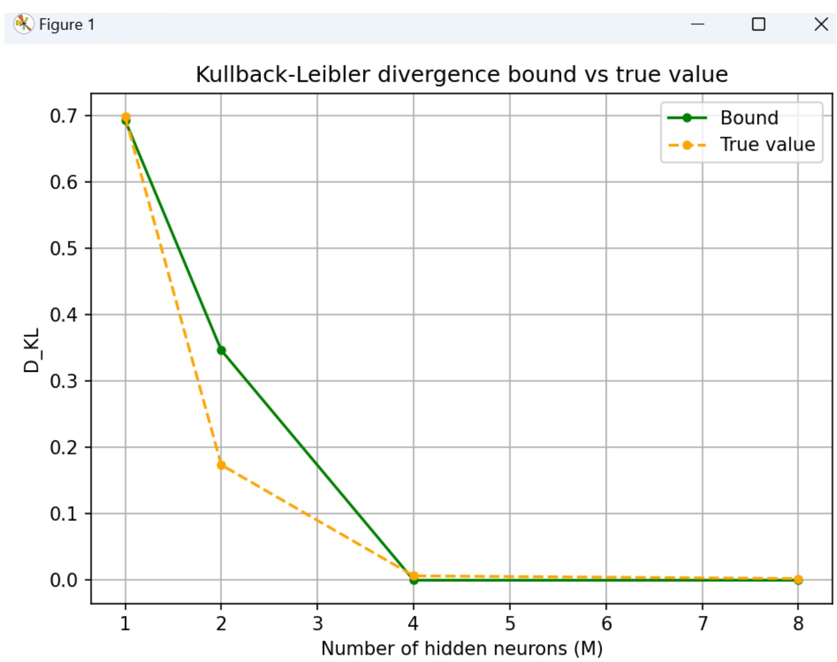

Figure 1: The figure represents the Kullback-Leibler divergence (D_{KL}) as a function of hidden neurons (M). The estimated upper bound (Green), and the value calculated after sampling (Orange).

Restricted Boltzmann Machine

Eric Blohm

October 7, 2024

Here is the Python code for the Restricted Boltzmann Machine:

```
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
```

```python


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)
```

```python
        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)

    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

```

```python
        #convert to 0 and 1 bits
        binary_pattern = []
        for bit in pattern:
            if bit == 1:
                binary_pattern.append(1)
            else:
                binary_pattern.append(0)

        # convert binary pattern to decimal index
        idx = 0
        n_bits = len(samples[0])
        for i,bit in enumerate(binary_pattern):
            idx += bit * (2**(n_bits-i-1))
        #print("Index: ", idx)
        PB[idx] +=1
    #normalize
    PB = PB/len_non_filled
    return PB


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

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],
```

```python
                                [-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)

```

```python
        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()
```

Perceptron with one hidden layer

Eric Blohm

October 1, 2024

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

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

#data has the input in the first two elements and output on the third.
def GetInputOutput(data):
    input = []
    output = []
    for row in data:
        input.append([row[0],row[1]])
        output.append(row[2])
    return np.array(input),np.array(output)


def getCSV(file):
    df = pd.read_csv(file,header=None)
    data_list = df.values.tolist()
    input,output = GetInputOutput(data_list)
    return input,output


def init_w_theta(M):
    # between input and hidden
    # 1/2 from the number of inputs
    variance = 1/2
    standard_dev = np.sqrt(variance)
    theta_j = np.zeros(M)
    w_jk = np.random.normal(0, standard_dev, size=(M,2))

    # between hidden and output
    variance_h = 1/M
```

```python
    standard_dev_h = np.sqrt(variance_h)
    theta = 0
    w_j = np.random.normal(0, standard_dev_h, size=(M))
    return w_jk, w_j, theta_j, theta


def compute_hidden_output(w_jk,theta_j,input):
    b_j = np.dot(w_jk, input.T) - theta_j.T
    return np.tanh(b_j)


def compute_network_output(w_j,theta,hidden_output,mu):
    sum = 0
    for j in range(0,len(w_j)):
        sum += w_j[j]*hidden_output[mu][j]
    B_i = sum - theta
    return np.tanh(B_i)


def back_prop(output_error,w_j,hidden_output,hidden_error):
    for m in range(0,len(w_j)):
        hidden_error[m] = output_error * w_j[m]* (1-hidden_output[m]**2)
    return hidden_error


def get_delta_w(input,hidden_output,hidden_error,output_error,eta, mini_batch,M):
    delta_w_j = np.zeros(M)
    #Delta_m, V_n. m = 1 only 1 output per pattern
    for n in range(0,len(hidden_output[0])):
        for mu in range(0,mini_batch):
            delta_w_j[n]+= output_error[mu]*hidden_output[mu][n]


    delta_w_jk = np.zeros((M,len(input[0])))
    #m is every hidden neuron.
    for m in range(0,len(hidden_error[0])):
        #n is x_1 and x_2
        for n in range(0,len(input[0])):
            for mu in range(0,mini_batch):
                delta_w_jk[m][n] += hidden_error[mu][m
```

```
]*input[mu][n]

    return eta*delta_w_jk, eta*delta_w_j


def get_delta_theta(output_error,hidden_error,eta,mini_batch,M):
    delta_theta = 0
    #m = 1
    for mu in range(0,mini_batch):
        delta_theta += output_error[mu]

    delta_theta_j = np.zeros(M)
    for m in range(0,len(hidden_error[0])):
        for mu in range(0,mini_batch):
            delta_theta_j[m] += hidden_error[mu][m]

    return -eta*delta_theta_j, -eta*delta_theta


def compute_classification_error(output,target):
    sum = 0
    for mu in range(0,len(target)):
        sum+= np.abs((np.sign(output[mu])-target[mu]))
    return (1/(2*len(target)))*sum


def compute_energy_function(output,target):
    sum = 0
    for mu in range(0,len(target)):
        sum += (target[mu]-output[mu])**2
    return 0.5*sum


def save_values(weights_jk,weights_j,threshold_1,threshold_2):
    df = pd.DataFrame(weights_jk)
    df.to_csv('w1.csv', index=False,header=False)

    df = pd.DataFrame(weights_j)
    df.to_csv('w2.csv',index=False, header=False)

    df = pd.DataFrame(threshold_1)
    df.to_csv('t1.csv',index=False, header=False)

    df = pd.DataFrame([threshold_2])
```

```python
    df.to_csv('t2.csv',index=False, header=False)


def main():

    ##### configuration #####
    M = 10
    epochsMax = 500
    batch_size = 64
    eta = 0.01
    #########################

    ### Retrieve data ###
    input,target = getCSV('training_set.csv')
    input_validation,target_validation = getCSV('validation_set.csv')
    ###################

    #### Center and normalize data ####
    input_mean = np.mean(input, axis=0)
    input_std = np.std(input, axis=0)
    ## Normalize based in training metrics
    input = (input - input_mean) / input_std
    input_validation = (input_validation - input_mean) / input_std
    #########################################

    ### initialize weights and thresholds ###
    w_jk,w_j,theta_j,theta = init_w_theta(M)
    #########################################


    ## used for plotting ##
    c_train_list = np.zeros(epochsMax)
    c_validate_list = np.zeros(epochsMax)
    ######################

    for epoch in range(0,epochsMax):
        ### Shuffle the input data and targets ###
        indices = np.arange(len(input))
        np.random.shuffle(indices)
        input = input[indices]
        target = target[indices]
        ##########################################

        #### Create mini batches #####
```

```python
        for start in range(0, len(input), batch_size):
            end = start + batch_size
            mini_batch = input[start:end]
            target_batch = target[start:end]

            ### Initialize outputs for the mini-batch ###
            hidden_output = np.zeros((len(mini_batch), M))
            output = np.zeros(len(mini_batch))
            output_error = np.zeros(len(mini_batch))
            hidden_error = np.zeros((len(mini_batch),M))

            #### for each pattern in mini batch  ####
            for mu in range(0,len(mini_batch)):
                #only one layer
                ##### Feed forward #####
                hidden_output[mu] = compute_hidden_output(w_jk,theta_j,mini_batch[mu])
                output[mu] = compute_network_output(w_j,theta,hidden_output,mu)
                ########################

                ##### back propagation #####
                output_error[mu] = (target_batch[mu]-output[mu])*(1-output[mu]**2)
                for m in range(0,len(w_j)):
                    hidden_error[mu][m] = output_error[mu] * w_j[m]* (1-hidden_output[mu][m]**2)
                ############################

            ##### Update weights #####
            #print(f"\n-Weights_jk before update: {w_jk}, \nWeights_j before update: {w_j}")
            delta_w_jk, delta_w_j = get_delta_w(mini_batch,hidden_output,hidden_error,output_error,eta, len(mini_batch),M)
            w_jk+= delta_w_jk
            w_j += delta_w_j

            delta_theta_j,delta_theta = get_delta_theta(output_error,
```

```
                hidden_error,eta, len(mini_batch),M)
            theta_j+= delta_theta_j
            theta   += delta_theta
            #########################

        ### validate during training and early stop ###
        hidden_output_validate = np.zeros((len(input_validation),M))
        output_validate = np.zeros(len(input_validation))
        for mu in range(0,len(input_validation)):
            hidden_output_validate[mu] = compute_hidden_output(w_jk,theta_j,input_validation[mu])
            output_validate[mu] = compute_network_output(w_j,theta,hidden_output_validate,mu)
        # #############################################

        ## Compute classification error and energy function. ##
        c = compute_classification_error(output_validate,target_validation)
        H_validate = compute_energy_function(output_validate,target_validation)
        print("C:", c*100, ", Energy function: ", H_validate)
        c_validate_list[epoch] = c*100
        if (c < 0.12):
            break
        # #############################################

    #if stopped early, retrieve all no negative elements
    c_validate_list = c_validate_list[c_validate_list > 0]

    ## create list for plotting ##
    epochs = np.arange(len(c_validate_list))

    ## plot Validation classification error ##
```

```python
    plt.plot(epochs, c_validate_list, marker='o', color='green', markersize=4, linestyle='-')
    plt.title('Validation Classification Error')
    plt.xlabel('Epochs')
    plt.ylabel('c_validate')
    plt.grid()

    plt.tight_layout()
    plt.show()
    # #######################################################

    ### Validate network after training ###
    hidden_output_validate = np.zeros((len(input_validation),M))
    output_validate = np.zeros(len(input_validation))
    for mu in range(0,len(input_validation)):
        hidden_output_validate[mu] = compute_hidden_output(w_jk,theta_j,input_validation[mu])
        output_validate[mu] = compute_network_output(w_j,theta,hidden_output_validate,mu)
    c = compute_classification_error(output_validate, target_validation)
    H_validate = compute_energy_function(output_validate,target_validation)
    print("C:", c*100, ", Energy function: ", H_validate)
    #######################################

    ### save the weights and thresholds to csv files ###
    #save_values(w_jk,w_j,theta_j,theta)
    # ###################################################


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